The Optimisation of Gradient Descent

Mateusz Syrek
University of Warwick

Novel publications proposing algorithms aimed at optimising the gradient descent of machine learning models often claim state-of-the-art performance when compared against older baseline algorithms such as Stochastic Gradient Descent and Adam, omitting a comparison against newer proposals. This project looked at exploring a wide range of such optimisation algorithms to compare them against each other, looking specifically at the early stages of training such as the first 40 epochs in the case of an image classification model. It was discovered that best performing algorithms appeared to generalise well across different network architectures and machine learning tasks with the KFAC and Ranger algorithms performing the best, obtaining an improvement of at least 4% accuracy over the baselines across different network architectures with even more significant improvements in the test perplexity of a language model. Beyond the generalisation ability of the model, well performing algorithms appeared to reach relatively flat regions of the loss surface with the largest Hessian eigenvalues reaching magnitudes of around 100, meanwhile lower test performance was indicated by large outlying eigenvalues of over 2000. Within the region of around 100 there seemed to be little correlation between surface sharpness and model performance, suggesting that there exists a threshold within which sharpness of the loss surface proves to have little influence over the model's generalisation.

Contents

1 Introduction

The past two decades have seen Machine Learning (ML) rise to the forefront of research in computer science and other fields such as medicine and finance. During this time, we have witnessed a revolution, owing largely to the advancements in hardware which have facilitated ever larger and more complex algorithms resulting in new 'state-of-the-art' models on a near year-by-year basis.

As new horizons are explored, human ambitions are likely to flare, with a desire to tackle and solve ever more difficult problems. This natural progression is well illustrated by the comparison of two different image classification datasets proposed roughly 10 years apart. In the 10 years, this pattern of continuous improvement has seen ML models go from attempting to classify grey-scale images of single digits [1] with 60,000 examples, to classifying full colour images to any of the 1000 classes with over 14 million examples in the ImageNet dataset [2]. With increased automation in the modern world, the datasets used in ML are certain to continue growing in size.

ML tasks utilise a model which by itself is a blank book, requiring a period of learning to tackle the task at hand. The increase in the complexity of ML tasks has required the development of ever more complex models, to a point where new tasks have outpaced advances in hardware leading to ever longer training times, with some requiring months of training. This itself has resulted in a paradigm shift within ML, turning towards parallel computation to reduce such immoderate time requirements. One major change has seen neural networks trained on graphical processor units (GPUs), which possess a greater number of cores allowing for faster parallel computation, proving excellent at accelerating matrix and tensor operations. This has been taken even further, with the use of computing clusters looking to utilise numerous machines, themselves possessing numerous GPUs, to allow for the effective training of large models. However, such clusters are typically very costly and hence prohibitive to many. As a result, researchers have begun to increasingly turn towards optimisation methods in an attempt to reduce training times with limited computing resources.

Many of the algorithms used for training ML models had their foundations formulated in the last century, with methods such as Stochastic Gradient Descent [3] dating back to 1951. Gradient-based algorithms are the leading methods which, as their name suggests, look to utilise gradient information during the training process. The recent drive for optimisation has seen these algorithms modified using methods from the field of Numerical Optimisation, with further work going into producing efficient implementations.

This project concerns itself with exploring numerous optimisation algorithms. In general, new publications in the field claim 'state-of-the-art' performance when comparing themselves against the established baselines and often omit a comparison against similar newly proposed methods. This allows an opportunity for a work to look more closely at these newer algorithms, endeavouring to compare them against each other and put their performance in a better perspective.

Some of the key goals of the paper can be summarised as,

- Provide a comprehensive comparison, both theoretical and empirical, across newly proposed optimisation algorithms and older baselines.
- Investigate various features of the optimisation process, such as the loss landscape, during the training of various ML models.
- Focus largely on the initial and most crucial stages of training during the first 40 epochs from initialisation, providing a unique insight into the behaviour of training

algorithms, unlike many studies which look towards the fully trained model.

The project concluded that while most algorithms perform well against the baseline, there exist significant variability across the different methods, with the KFAC algorithm which uses natural gradient information proving superior in both test performance and hyperparameter robustness compared to others. Furthermore, some of the algorithms were discovered to possess computational overhead not made apparent by their theoretical analysis, owing largely due to the implementation of modern ML frameworks which consequently should be considered more closely in novel publications.

2 Background

2.1 The Machine Learning Problem

The ultimate aim of any ML model is good performance on previously unseen data, also referred to as generalisation. Training a ML model requires a dataset which is partitioned into a training and testing set, with a further evaluation set sometimes used. The model then attempts to learn the underlying distribution of the data by only looking at samples from the training set. This training process is formulated as a minimisation problem, with the objective being the loss function L which measures how wrong the model is in its current predictions. The training and test distributions, while sampled from the same underlying data distribution, are not identical but rather see slight shifts in distribution space, giving rise to interesting phenomena described in optimisation literature, such as the generalisation gap between sharp and flat minima [4, 5], along with the more widely known phenomenon of overfitting [6].

Defining this more formally as a supervised learning problem, the ML model produces a predictor function $f : X \to Y$, with internal parameters θ, which attempts to learn a mapping from the input space $X \in \mathbb{R}^{n \times d}$, to the output space $Y \in \mathbb{R}^n$ according to the distribution of a dataset consisting of n data-target pairs $\{x, y\}$. Such predictor functions are task and model dependent, they generally range from simple linear functions used in regression models (1) to more complex, non-linear functions as seen in neural networks.

$$f(\theta, x) = \theta^T x \tag{1}$$

Data is typically handled in terms of multi-dimensional matrices known as tensors, which in the case of the dataset mentioned above are defined as $x \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$, with n being the number of data samples within the dataset and d being the implicit dimensionality of individual data points.

The training process amounts to minimising the Expected Risk (2), however, at the time of training the model does not know the underlying data distribution and has no access to dP. Instead, the model looks to minimise an approximation of the expected risk known as the Empirical Risk (3) which equals the expected risk in expectation [50, 51].

$$R(\theta) = \int L(f(\theta, x), y)\ dP \tag{2}$$

$$R_{emp}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(f(\theta, x_i), y_i) \tag{3}$$

In computing the Empirical Risk, the model uses the loss function $L : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ to measure how wrong its current prediction for a set of inputs $\hat{y}$ is, when compared to the true value y. Like the predictor function, the loss function is task dependent, with

Cross Entropy Loss (4) and Mean Squared Error (5) commonly used in classification and regression tasks respectively.

$$L(\hat{y}, y) = \sum_{i=1}^{d} y_i \log(\hat{y}_i) \tag{4}$$

$$L(\hat{y}, y) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{5}$$

2.2 The Minimisation Problem

In almost all cases, the task of minimising the empirical risk is done iteratively, as analytic solutions prove intractable for larger problems. Today, virtually all ML models use gradient-based optimisation methods for training, with many variants of the gradient descent (GD) algorithm. GD proceeds by making iterative updates to the model's parameters according to the gradient of the loss function w.r.t these parameters $\nabla_\theta L$. The updates proceed along the direction opposite to the gradient of the loss function at each point, taking a path down the curvature of the loss surface towards the minimum, with the size of each step controlled by the parameter α.

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(f(\theta_t, x), y) \tag{6}$$

This type of GD is often referred to as Full-Batch GD, as at each iteration it utilises the entire dataset in computing the gradient. Such full-batch methods scale with the size of the dataset $\mathcal{O}(n)$ and prove impractical for larger problems. With modern datasets often growing to millions of data samples, this algorithm would prove of little use in ML today.

As a result of this most models use the Stochastic Gradient Descent (SGD) [3], which at each iteration computes the gradient of the loss function w.r.t a single datapoint, sampled stochastically from the dataset (7) where the complexity of each iteration is now constant $\mathcal{O}(1)$, in the size of the dataset. Each iteration of SGD is now a stochastic process, meaning that the computed gradient equals the true gradient only in expectation (8). Any individual update may take steps which at times diverge from the optimal descent path, introducing variance into the gradient estimation.

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(f(\theta_t, x_i), y_i) \tag{7}$$

$$\nabla_\theta L(f(\theta_t, x), y) = \mathbb{E}\big[\nabla_\theta L(f(\theta_t, x_i), y_i)\big] \tag{8}$$

The introduction of variance proves detrimental for the algorithm's rate of convergence, in fact SGD possesses a sub-linear rate of convergence, compared to the full-batch method's linear rate. This rate can be improved through a compromise between the two methods, involving a mini-batch of data at each iteration. The size of this mini-batch is typically much smaller than the size of the dataset, with the convention being a power of 2 as this facilitates better hardware utilisation [7]. Mini-batching is able to reduce variance while maintaining a scalable algorithm, applicable to large datasets.

2.3 The Convergence Problem

During training, the optimisation algorithm produces a convergent series with the loss value at iteration t given by L_t and an optimum L^*. There exist three fundamental classes of convergence rates [8] which are considered in optimisation literature, these are visualised in figure 1.

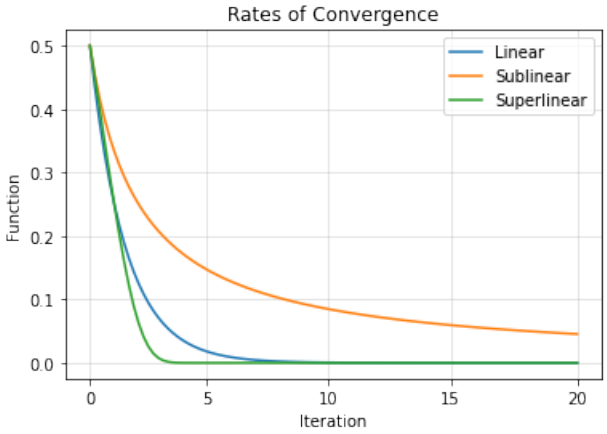


Figure 1: The different rates of convergence.

- **Sublinear Convergence** A sequence of loss values is said to converge at a sublinear rate if the following limit holds.

$$\lim_{t\to\infty} \frac{|L_{t+1} - L^*|}{|L_t - L^*|} = 1$$

  This is the slowest rate of the three classes considered here, it is also the rate achieved by single-sample SGD. In general, most acceleration algorithms aim to improve the algorithm's convergence beyond this rate.

- **Linear Convergence** A sequence of loss values is said to converge at a linear rate if the following limit holds.

$$\lim_{t\to\infty} \frac{|L_{t+1} - L^*|}{|L_t - L^*|} = \alpha \ , \quad \alpha \in (0, 1)$$

  Note that $(0, 1)$ is an open interval. This is the rate achieved by full-batch GD, with most acceleration algorithms aiming to achieve this rate of convergence.

- **Superlinear Convergence** A sequence of loss values is said to converge at a superlinear rate if the following limit holds.

$$\lim_{t\to\infty} \frac{|L_{t+1} - L^*|}{|L_t - L^*|} = 0$$

  This is the rate achieved by some second order algorithms on quadratic objective functions. Beyond this, it is a rate achieved only in theory as most algorithm use some form of approximation, slowing down convergence.

Convergence rates alone should not be used as a metric on which to evaluate optimisation algorithms, as they overlook factors such as the complexity of each iteration and the performance of the final model. Instead, they can be used as an indicator of performance to be used in conjunction with further analysis.

2.4 The Case for Convexity

The principle of convexity is essential in establishing the convergence proofs of GD algorithms. This section will begin to present some more involved proofs which may be too lengthy to include in the main body of the paper, in which case the reader will be directed towards the appendix or other relevant literature.

2.4.1 Convex Sets

In describing convexity, it is best to begin with the case of the convex set which contains the data used by the function. A convex set is defined to be a subset C of the corresponding vector space wherein every line segment connecting any two points in the set is itself contained within the set. More formally, a set $C \in \mathbb{R}^n$ is defined as convex if (9) holds.

$$\forall x, y \in C \ : \ \alpha x + (1-\alpha)y \in C, \ \forall \alpha \in [0,1] \tag{9}$$

Furthermore, strict convexity is achieved when all points except the end points are contained within the set, this corresponds to the interval on the RHS of (9) being open, $[0,1] \to (0,1)$.

2.4.2 Convex Functions

The concept behind convexity in functions is similar to that of convex sets. A convex function has every line segment connecting any of its two points lie entirely on or above the function line. Formally, given a convex set $C \in \mathbb{R}$, a function $f : C \to \mathbb{R}$ is said to be convex if (10) holds [9].

$$\forall x, y \in C \ : \ f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y), \ \forall \alpha \in [0,1] \tag{10}$$

Similarly to a convex set, strictly convex functions have all line segments lie entirely above the function line, this amounts to swapping the inequality in (10) for a strict inequality.

Applying GD algorithms assumes the objective function to be differentiable, this allows for additional convexity conditions to be derived from (10) (see appendix A for a full derivation). Given that f is differentiable, (10) can be expanded to provide a first- and second-order condition of convexity (11), (12).

$$\forall x, y \in C \ : \ f(y) \geq f(x) + \nabla f(x)^T (y - x) \tag{11}$$

$$\forall x \in C \ : \ \nabla^2 f(x) \geq 0 \tag{12}$$

The RHS of the first-order condition (11) is equivalent to a first-order Taylor expansion at x, which provides a global under estimator for the function. This gives rise to one of the key tenets of convexity, **local gradient information can provide global function information**. A local optimality condition $\nabla f(x) = 0$ can be extended to entail global optimality of a convex function. For a convex function f, given the local optimality and the first-order condition (11), it must be the case that,

$$\forall y \in C \ : \ f(y) \geq f(x)$$

This means that any locally optimal point is also the global optimum. This feature is dependent upon the conditions of convexity and hence breaks down for non-convex functions.

The aforementioned conditions make the optimisation of convex functions far simpler and consequently faster than for non-convex functions, as the optimisation algorithm must only find the point at which the gradient is 0. In general however, the convex conditions (11), (12) only hold for simple ML models which use linear discriminant functions, these include Linear and Logistic Regression models. However, most complex ML models utilise non-linearities which break the global convexity conditions, one class of models which do this are neural networks whose loss surface proves to be highly non-convex.

2.4.3 Lipschitz Continuity

A further assumption widely utilised in the theoretical analysis of GD is that of Lipschitz Continuity, which is stated formally below.

A differentiable function $f : \mathbb{R}^a \to \mathbb{R}^b$ is defined to be Lipschitz Continuous at a point $x \in \mathbb{R}^a$ if there exists a constant L such that the following holds

$$\forall\ y \in \mathbb{R}^a\ :\ ||\nabla f(y) - \nabla f(x)|| \leq L||y - x||$$

Usually utilised alongside convexity, this assumption ensures that the function vector does not rapidly change with the parameters, ensuring a 'smooth' gradient. Sometimes this condition is also referred to as 'L-Smoothness'.

2.5 Non-Convex Analysis

With essentially all state-of-the-art models consisting at least in some part of deep neural networks, any optimisation algorithm which aims to accelerate the procession of gradient descent must perform well in non-convex environments. While convex analysis of GD algorithms is covered extensively in optimisation literature, the non-convex case has seen much less attention and there remains a gap in our understanding which separates the theoretical convergence of GD and its remarkable performance in training neural networks.

The non-convex case is markedly more difficult for GD algorithms. Unlike the convex case which simply involved searching for the region where $|\nabla L| = 0$ indicating the global minimum, non-convexity introduces two further critical points which can entrap GD algorithms hindering convergence.

2.5.1 Local Minima

The first of these are local minima. These consist of points where $|\nabla L| = 0$, surrounded by positive curvature. In most functions, local minima exist at higher function values than the global, however the validity of this in the case of neural networks remains a topic of great contention within the ML community, with arguments suggesting that such local minima within NNs are not bad [10, 11, 12] and hence do not hinder the performance of the trained model. However, such analysis often requires simplifying assumptions and conditions which apply only to certain ML problems, furthermore [13] was able to create bad minima within networks using synthetic datasets, dispelling the idea that such minima cannot exist. Regardless of the optimality of local minima, certain GD algorithms remain capable of generalising well across various deep networks, with no clear correlation to the optimality of local minima. Instead, there exist other factors which have been shown to exert greater influence over the generalisation performance of the model.

2.5.2 Saddle Points

Perhaps more detrimental to training is the presence of saddle points. These are points where the maximum of one dimension coincides with the minimum of another, hence resulting in a flat-like region of low gradient. In high-dimensional parameter spaces, saddle points can be identified using the Hessian matrix of second-order derivatives, $H \in \mathbb{R}^{d \times d}$ with d corresponding to the dimensionality of the parameter space. An indefinite Hessian, containing both positive and negative eigenvalues indicates both positive and negative curvature as is seen at a saddle point. In large networks with $d > 10^8$, it

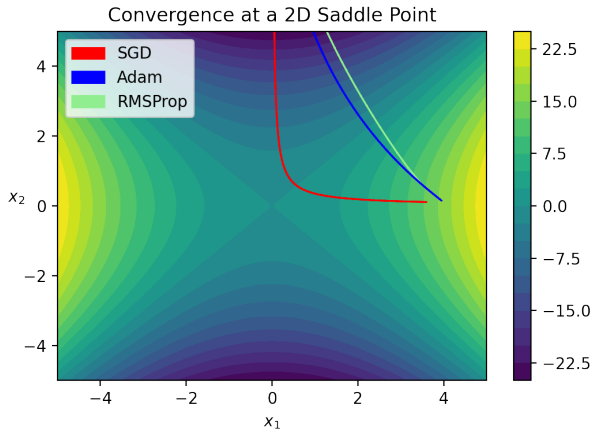


Figure 2: The convergence of different GD algorithms, passing through a 2D saddle point.

is increasingly less likely that a Hessian will be composed of entirely positive or negative eigenvalues, resulting in the proliferation of saddle points inside high-dimensional parameter spaces. Saddle points directly affect the descent path of individual GD algorithms, with different algorithms diverging at these points [14, 15] and leading to widely different final minima. This can be seen in figure 2, where each of algorithm takes a different path. This effect is further amplified for high-dimensional saddle points, more prevalent inside deep neural networks.

2.6 Convergence of Gradient Descent

With a brief introduction to the principles of convexity, it is now possible to theoretically derive the convergence proof for GD.

Let us consider a differentiable objective function $f : \mathbb{R}^d \to \mathbb{R}$ which is also convex and satisfies the Lipschitz Continuity condition with a Lipschitz constant L. Taking the quadratic expansion of the function at y to be,

$$f(y) = f(x) + f(x)^T(y - x) + \frac{1}{2}H||y - x||^2$$

However, considering only first-order GD, the expression has no access to the Hessian H and instead replace the H by an approximation $\frac{1}{t}I$, where I is an identity matrix and t is the step size used in the algorithm. Furthermore, take $f(y)$ to be the next step in the GD algorithm which must be less than or equal to the current step value, allowing for the inclusion of the inequality below.

$$f(y) \;\leq\; f(x) + f(x)^T(y - x) + \frac{1}{2t}||y - x||^2 \tag{13}$$

Set $y = x^+ = x - t\nabla f(x)$ to be the next iterate in the series, after rearranging this can be substituted into (13), obtaining (14) below.

$$f(x^+) \;\leq\; f(x) - \frac{t}{2}||\nabla f(x)||^2 \tag{14}$$

Next, the $f(x)$ in (14) can be substituted for the first-order convexity condition (11) around the optimal point $f(x^*)$, which when rearranged allows for its substitution into

(14) as

$$f(x) \;\leq\; f(x^*) + \nabla f(x)^T (x - x^*)$$

consequently giving the following expression,

$$f(x^+) \;\leq\; f(x^*) + \nabla f(x)^T (x - x^*) - \frac{t}{2} ||\nabla f(x)||^2$$

This can be expanded further by substituting the value of each gradient $\nabla f(x)$ for the parameter values according to the GD update rule.

$$f(x^+) \;\leq\; f(x^*) + \frac{1}{2t}(||x - x^*||^2 - ||x^+ - x^*||^2) \tag{15}$$

Taking $x = x_0$ to be the initial iteration and $x^+ = x_k$ to be any later iteration after k steps, both sides are summed over k iterations and $f(x^*)$ is moved to the LHS.

$$\begin{aligned} \sum_{i=1}^{k} (f(x_i) - f(x^*)) \quad &\leq \quad \frac{1}{2t}(||x_0 - x^*||^2 - ||x_k - x^*||^2) \\ &\leq \quad \frac{1}{2t}||x_0 - x^*||^2 \end{aligned}$$

Finally, note that any $f(x_k)$ after k iterations is non-decreasing hence the final expression may be written as (16).

$$f(x_k) - f(x^*) \;\leq\; \frac{1}{k} \sum_{i=1}^{k} \big(f(x_i) - f(x^*)\big) \;\leq\; \frac{1}{2tk} ||x_0 - x^*||^2 \tag{16}$$

And so finally, from (16) the linear convergence rate observed for GD is obtained, $\mathcal{O}(\frac{1}{k})$ with k being the number of iterations. □

2.7 Convergence of Stochastic Gradient Descent

The theoretical convergence of GD omits the computational overhead involved in the gradient computation at each iteration, proving impractical for large datasets. SGD proposes a solution to this by computing the gradient w.r.t individual data points, sampled stochastically from the dataset.

As with GD consider a convex, differentiable, and L-Lipschitz function $f : \mathbb{R}^d \to \mathbb{R}$, again taking its second-order Taylor expansion with step size t.

$$f(y) \;\leq\; f(x) + f(x)^T (y - x) + \frac{1}{2t} ||y - x||^2$$

As the gradient at each step is now computed w.r.t to individual data points, it equals the true gradient only in expectation and results in the introduction of unwanted variance. As before, taking $y = x^+ = x - tg$ to be the next set of parameters in the series, with step size t and g denoting the stochastic gradient estimation at each step. In considering the convergence of these steps, g has a bounded variance $Var[g] \leq \sigma^2$. The above expression can be rearranged to give,

$$f(x^+) \;\leq\; f(x) - t f(x)^T g + \frac{t}{2} ||g||^2$$

The stochastic estimation g equals the true gradient in expectation, $\nabla f(x) = \mathbb{E}[g]$. Taking the expectation w.r.t g on both side of the above expression yields the following,

where the terms on the far RHS follow from rearranging the expression for the variance of g (17).

$$Var[g] = \mathbb{E}\big[||g||^2\big] - ||\mathbb{E}[g]||^2 \tag{17}$$

$$\mathbb{E}\big[f(x^+)\big] \;\leq\; f(x) - t||f(x)||^2 + \frac{t}{2}\big(||\nabla f(x)||^2 + Var[g]\big)$$

Rearranging this and substituting the expression for the bounded variance,

$$\begin{aligned}\mathbb{E}\big[f(x^+)\big] \;&\leq\; f(x) - t(1-\frac{1}{2})||f(x)||^2 + \frac{t}{2}Var[g] \\ &\leq\; f(x) - \frac{1}{2}||f(x)||^2 + \frac{t}{2}\sigma^2 \end{aligned} \tag{18}$$

As was done with the convergence of GD, take the first-order convexity condition (11) and rearrange it for the optimum point x^* and another iterate x, this time including the estimated gradient g.

$$f(x) \;\leq\; f(x^*) + \mathbb{E}\big[g^T\big](x - x^*)$$

This is then substituted for $f(x)$ in (18), with (17) again rearranged.

$$\begin{aligned}\mathbb{E}\big[f(x^+)\big] \;&\leq\; f(x^*) + \mathbb{E}\big[g^T\big](x-x^*) - \frac{1}{2}||\nabla f(x)||^2 + \frac{t}{2}\sigma^2 \\ &\leq\; f(x^*) + \mathbb{E}\big[g^T\big](x-x^*) - \frac{1}{2}\mathbb{E}\big[||g||^2\big] + \frac{t}{2}\sigma^2 \\ &\leq\; f(x^*) + \mathbb{E}\Big[g^T(x-x^*) - \frac{1}{2}||g||^2\Big] + \frac{t}{2}\sigma^2 \end{aligned}$$

Rearranging and substituting the gradient update rule as in GD.

$$f(x^+) \;\leq\; f(x^*) + \frac{1}{2t}\big(||x-x^*||^2 - ||x^+ - x^*||^2\big) + \frac{t\sigma^2}{2}$$

The remainder of the proof follows much like the GD case above. Consider $x = x_0$ to be the initial iteration while $x^+ = x_k$ is taken to be any iteration after k steps. Moving $f(x^*)$ to the left and summing both sides over k gives.

$$\begin{aligned}\sum_{i=1}^{k}\big(\mathbb{E}\big[f(x_i)\big] - f(x^*)\big) \;&\leq\; \frac{1}{2t}\mathbb{E}\Big[||x_0 - x^*||^2 - ||x_i - x^*||^2\Big] + \frac{kt\sigma^2}{2} \\ &\leq\; \frac{1}{2t}\mathbb{E}\big[||x_0 - x^*||^2\big] + \frac{kt\sigma^2}{2} \\ &\leq\; \frac{1}{2t}||x_0 - x^*||^2 + \frac{kt\sigma^2}{2} \end{aligned}$$

Noting that $f(x_k)$ is non-decreasing after k iterations, the expression can be rewritten into its final form.

$$\mathbb{E}\big[f(x_k)\big] - f(x^*) \;\leq\; \frac{1}{k}\sum_{i=1}^{k}\big(\mathbb{E}\big[f(x_i)\big] - f(x^*)\big) \;\leq\; \frac{1}{2tk}||x_0 - x^*||^2 + \frac{t\sigma^2}{2} \tag{19}$$

Unlike GD, even after a large number of iterations k, SGD only converges towards a region around the minimum, bounded by the step size t. □

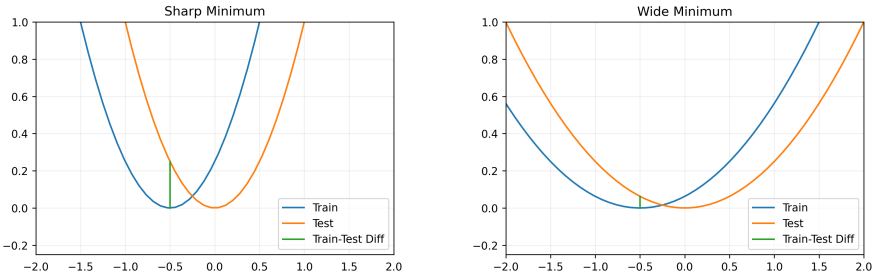


Figure 3: Illustration of how training loss generalises to test loss for different types of minima. In general, wider minima result in a smaller increase in loss moving to the test distribution.

3 Machine Learning Models

In evaluating optimisation algorithms used in ML, it is important to also consider what models these are applied to.

3.1 Neural Networks

Neural networks have established themselves as the firm leaders in the field of ML, with state-of-the-art models right across different sub-fields consisting entirely of various neural network architectures. As a result, any modern optimisation algorithm must perform well in the non-convex setting of a neural network loss function. The layered structure of neural networks sees the use of the backpropagation algorithm [47] to obtain the derivative of each network layer w.r.t the parameters, this is then used in the iterative updates of GD algorithms.

State-of-the-art deep NNs are non-convex, stemming from the repeated use of non-linear activation functions across different layers [45]. This is a necessity in NNs, as it allows for the effective modelling of more complex problems, unfortunately it results in a highly non-convex loss function which can prove difficult to optimise. In fact, finding the global minimum of non-convex functions is NP-Hard [46]. Furthermore, NNs with any significant depth are overparametrized, with the number of parameters exceeding dimensionality of the dataset, in most ML problems this would result in significant overfitting during training.

Remarkably, with all these red flags, neural networks can achieve state-of-the-art performance across all of machine learning where in most cases, SGD proves sufficient for effective training. The high dimensionality of the parameter spaces inside neural networks has meant that an inside view of their loss landscape remains shrouded in mystery, even with novel visualisation methods these typically offer limited insights. The surprising effectivity of NNs remains an open research topic.

3.1.1 Optimality of Flatness

One of the primary paradigms in optimisation literature considers the shape of the minima towards which algorithms converge and the effect that this has on the generalisation performance of the model. Both the training and test datasets can be thought of as sampled from the underlying data distribution, their distributions are similar but not identical and one may be shifted from the other. In the case of a sharp minimum,

Model Architecture	No. of Parameters
EfficientNet(B2) [17]	9.2M
ResNet(18) [18]	11.7M
InceptionV3 [19, 20]	27.2M
VGG(11) [21]	132.9M
RoBERTa [43]	80.4M

Figure 4: Neural Network architectures.

even slight displacement can result in a significant increase in loss, meanwhile a flatter minimum will generally have a smaller increase and a similar training-test loss. This principle is well illustrated in figure 3.

Intuition would suggest that sharp minima generalise poorly when compared to wider, flat minima. This is also what is suggested in several publication [4, 5], however one should be careful with formulating such intuitions as while this is the case in low-dimensional spaces such as the 2-D plot in figure 3 the effect of sharpness along individual dimensions within a parameter space consisting of millions of dimensions could prove to have less influence. Consequently, there have been several suggestions which question this concept [53, 54], and it remains a much debated research topic within the field of machine learning.

3.2 Computer Vision

Computer vision has long been a staple within the ML community, providing a task with real world application which is sufficiently difficult to allow for the effective evaluation of models and optimisation algorithms alike.

3.2.1 Image Classification

One of the oldest computer vision problems involves making a classification based on an input image, with the number of classes dependant on the dataset. The dataset used for evaluation is the CIFAR 100 [16] dataset, widely used to benchmark ML models it provides a sufficiently complex task for the optimisation algorithm, with 100 different classes and 60,000 images which are used to make the classification. The network architectures chosen for this study are shown in figure 4. These were selected to display the different design paradigms seen in ML today, from the slightly older and larger architectures such as VGG to the newer networks like EfficientNet which seek computational efficiency without sacrificing performance.

3.3 Natural Language Processing

Another field which has benefited greatly from the widespread application of neural networks is Natural Language Processing (NLP). Similarly to computer vision, NLP has gone through a revolution in the past decade with new network architectures [59] coming to dominate the field. Today, the paradigm commonly seen in NLP involves a two-step training procedure with the model first pre-trained on a language modelling task using a large body of text in the language of choice, followed by fine-tuning the model on specific downstream tasks such as Sentence Classification or Question Answering. In its evaluation, this paper focuses on the pre-training step, responsible for learning the structure of the language it influences the downstream performance of any fine-tuning

task.

The evaluation involves the WikiText-103 dataset [23], composed of 28,000 Wikipedia articles it sports over 100 million tokens inside its training set. The model of choice was selected to be RoBERTa [43], with 80 million parameters it is a smaller variant of the BERT [44] model which trains through masked language modelling, where the model attempts to correctly predict randomly masked words within a batch of text.

4 Optimisation Algorithms

There exist numerous variants of the GD algorithm, each aiming to optimise the training procedure of ML models. To illustrate the behaviour of these algorithms in different environments, this section looks at their convergence plots on various test functions. These are used to illustrate how the algorithms explore surface features and alone are not indicative of good performance as they equate to a simple 2-D optimisation problem with a single perfect data point. Nonetheless, some of the patterns seen in these translate into real ML problems. These tests are performed using the Beale (20) and Schubert (21) functions, whose initialisation points were carefully selected to investigate the variance handling capabilities of different algorithms.

$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 \tag{20}$$

$$f(x,y) = \Big(\sum_{j=1}^{5} \cos\big((j+1)x + j\big)\Big)\Big(\sum_{j=1}^{5} \cos\big((j+1)y + j\big)\Big) \tag{21}$$

4.1 Stochastic Gradient Descent

SGD proceeds to update the model parameters according to the derivative computed w.r.t to a stochastically sampled batch of data, $\{x_i, y_i\}$.

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(\theta, x_i)$$

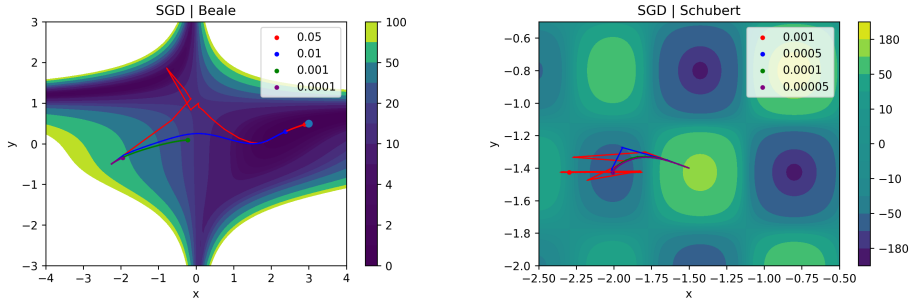
The reduced complexity of SGD when compared to its full-batch counterpart does not come free of burdens, as the stochastic sampling of data points means that the gradient computed at each iteration is an approximation of the true gradient and equates to it only in expectation. Furthermore, this gradient estimation now sees the introduction of variance (22) which can at times hinder the convergence of the algorithm.

$$\mathbb{V}\big[\nabla_\theta L(\theta, x_{n_b})\big] = \mathbb{E}\big[||\nabla_\theta L(\theta, x_{n_b})||^2\big] - ||\mathbb{E}\big[\nabla_\theta L(\theta, x_{n_b})\big]||^2 \tag{22}$$

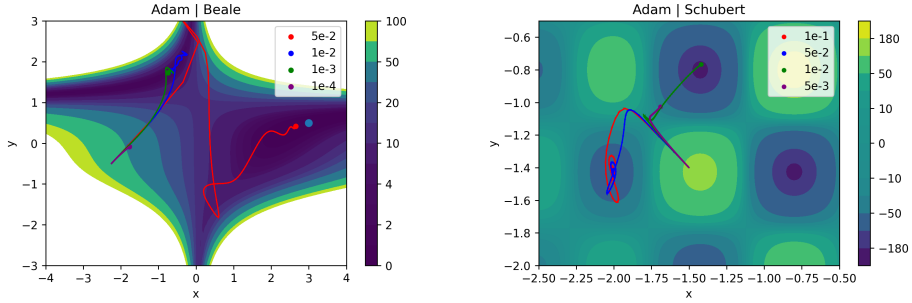
The variance sees the convergence rate of SGD reduce from the linear rate of full-batch GD to a sublinear rate. This however is not as detrimental as it may seem, as the reduction in complexity results in faster computation, allowing for more iterations to be computed to account for this reduced rate. The main drawback of SGD is that it does not actually converge towards the optimum θ^*, instead it converges towards a geometric ball centred on θ^* within the d-dimensional parameter space $\hat{\theta} \in \mathbb{R}^d$.

$$B_\theta(\theta^*) = \big\{x \in \hat{\theta} \ : \ d(\theta_*, x) < r\big\} \tag{23}$$

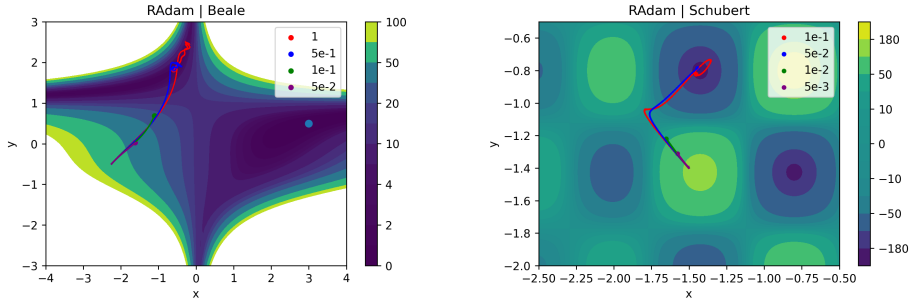
This principle was illustrated in section 2.7, with the size of the ball (23) proportional to the step size $r \propto t$ (19). This comes as a consequence of the stochastic noise within each SGD estimate, requiring a diminishing step size for convergence towards a fixed point,



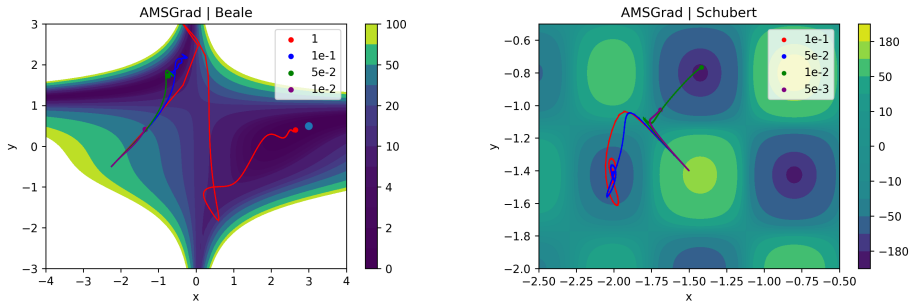
(a) **SGD**



(b) **Adam**



(c) **RAdam**



(d) **AMSGrad**

Figure 5: Algorithm convergence plots on the Beale and Schubert test functions, trained with 4 different learning rates for 100 iterations. The learning rates illustrate the range of values which allowed for convergence, with values beyond those shown proving detrimental to the algorithm.

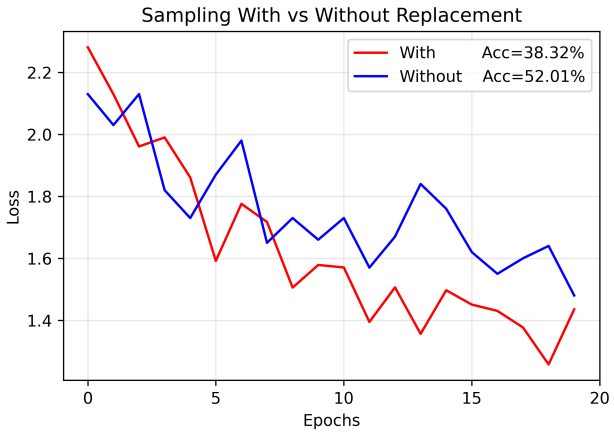


Figure 6: Training loss and test performance of a shallow convolutional neural network on the CIFAR10 dataset, comparing the performance when trained using data sampled with and without replacement.

as otherwise the algorithm continues to oscillate within (23). While this may suggest that SGD is a poor choice, empirical results have shown that it can obtain good generalisation performance on deep learning models even when using constant learning rates. This may indicate that convergence towards a region of low loss around the optimum is a sufficient goal for optimisation algorithms.

Figure 5 presents a series of plots displaying the descent path of various GD algorithms on the two test functions mentioned above. Each algorithm was evaluated at four different learning rates which displayed convergence, with rates above those shown leading to erratic divergence. From figure 5a it can be seen that SGD does well to navigate smoother landscapes such as that of the Beale function, with higher rates converging towards the optimum while smaller rates easily stall without a sufficient number of iterations. Furthermore, the utility of the variance within the gradient can be seen, as the large learning rates which may erroneously converge towards a local minimum can dislodge themselves ultimately converging towards the optimum. The steeper Schubert function illustrates that even with the inherent noise, SGD can still become trapped as all learning rates converge towards a local minimum beside the starting point.

One key question behind the implementation of SGD-like algorithms asks if the data is sampled with or without replacement. In almost all cases, stochastic algorithms sample data without replacement as this ensures that each epoch sees an entire pass over the dataset allowing the model to best fit the underlying data distribution. This effect can be seen in figure 6, with a clear improvement in the test accuracy of the model sampling without replacement. All algorithms utilising stochastic sampling in this paper do so without replacement.

4.2 Adaptive Learning Rate Methods

Adaptive learning rate algorithms seek to use additional gradient information to provide a per-parameter learning rate for each gradient update. They leverage information from previous steps in an attempt to reduce the variance inherent in stochastic systems.

4.2.1 Adam

Another baseline widely used in optimisation literate is Adam [24]. Proposed in 2014, it has been utilised repeatedly in training state-of-the-art models across different fields. Adam seeks to modify the gradient update of each parameter by scaling it, which results in a unique learning rate for each parameter in the model. The algorithm does this by utilising an exponentially decaying average of the gradient and squared gradient at each step. These estimate the first moment m_θ (mean) and the second moment v_θ (uncentered variance) respectively. The decay rates are controlled through hyperparameters $\beta_1, \beta_2 \in [0, 1)$.

$$m_\theta^{t+1} = \beta_1 m_\theta^t + (1 - \beta_1)\nabla_\theta L(\theta_t, x_i)$$
$$v_\theta^{t+1} = \beta_2 v_\theta^t + (1 - \beta_2)(\nabla_\theta L(\theta_t, x_i))^2$$

To avoid bias, which is most prevalent in the initial iterations, at each step the algorithm recomputes an unbiased estimate of the parameters.

$$\hat{m_\theta} = \frac{m_\theta^t}{1 - \beta_1^t}$$

$$\hat{v_\theta} = \frac{v_\theta^t}{1 - \beta_2^t}$$

The final gradient update is then given by (24), with the inclusion of a small constant $\epsilon << 1$ to avoid numerical explosions.

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m_\theta}}{\sqrt{\hat{v_\theta}} - \epsilon} \tag{24}$$

The algorithm looks to accommodate variance fluctuations within each parameter, reducing the update contributions from parameters which suffer from excessive variance caused by the stochastic sampling of the algorithm.

The behaviour of Adam differs quite significantly from SGD. From the plots in figure 5b it can be seen that Adam takes a more circuitous route to convergence. This proves detrimental for the Beale function where all but the largest rates appear to favour the local minimum. The larger rates appear to rebound from the local minimum, ultimately converging towards the global minimum of the function. In steeper landscapes, Adam favours smaller step sizes, where even the larger steps appear unable to escape from deep minima upon entrapment, while the smaller rates prove capable of better navigating the upper slopes of the function surface finding the correct descent path towards the optimal point.

Adam and SGD are widely considered the baseline algorithms in ML. To explore beyond these, this section now looks at some of the more recent proposals in the field. Most newer algorithms work from the base Adam optimiser, providing slight modifications in an attempt to improve its performance with modern architectures and models.

4.2.2 RAdam

As Adam utilises past gradient information, the initial iterations of the algorithm can be quite unstable while the algorithm gathers past information, resulting in high variance. RAdam [25] aims to fix this by modifying the update rules of Adam to rectify the high variance observed in the initial gradient updates.

First, during initialisation it computes the maximum allowable second moment length ρ_∞ which it uses to gauge if the model is experiencing a period of high variance.

$$\rho_\infty = \frac{2}{(1-\beta_2)} - 1$$

In the cases where the variance is small and does not exceed ρ_∞, the algorithm proceeds to update the parameters using the un-adapted momentum with m_θ from the definition of Adam.

$$\theta_{t+1} = \theta_t - \alpha \hat{m_\theta}^t$$

When the variance exceeds the limit set by ρ_{inf}, the algorithm computes a gradient rectification term which it includes in the RHS of the full Adam update step seen in (24), with the rectification term r_t given by,

$$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}$$

The behaviour of RAdam as seen in figure 5c displays the variance reduction mechanism at work. The high learning rates in both functions illustrate a much more uniform convergence without any significant stochastic behaviour. In the case of the Beale function this proves somewhat detrimental as it results in convergence towards the nearest minimum which in this case is a local minimum. The Schubert function however benefits from RAdam as it guides all descent paths along towards the global minimum. The reduced variance at the start means that the algorithm performs a much more controlled descent, strictly following the geometry of the function surface rather than overshooting it as was seen with Adam.

4.2.3 AMSGrad

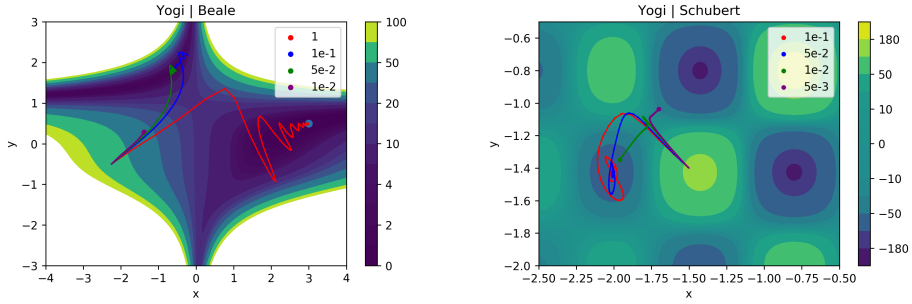
While RAdam looks to improve the performance of Adam in the initial stages of training, AMSGrad looks to improve the convergence at later stages. The key problem with Adam comes from its main feature, the decaying average mechanism which endows Adam with a short-term memory meaning that particularly informative data batches have a short-lived effect and are soon forgotten in favour of new and often uninformative data. The only change introduced in AMSGrad modifies the way in which the algorithm computes the second moment. Instead of computing the unbiased estimate like Adam, AMSGrad utilises the previously used second moment if it proves larger than the previous unbiased estimate.

$$\hat{v_\theta}^{t+1} = \max(\hat{v_\theta}^t, v_\theta^t)$$

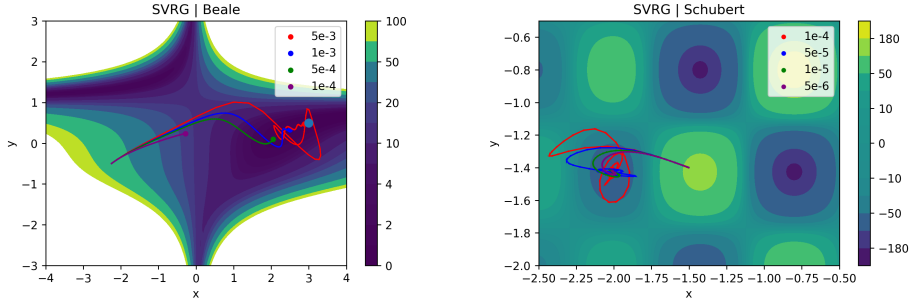
This allows the algorithm to maintain a memory of previous gradient information which provided significant updates, accelerating convergence. The effectivity of AMSGrad is not apparent for simple function such as those shown in figure 5d, where the convergence of AMSGrad appears identical to that of Adam. This mean that the long-term memory mechanism, which activates in high variance environments, is not utilised for such simple functions. The high variance utility of AMSGrad is more likely to become apparent in more complex environments such as in training neural networks, which will be explored in later sections.

4.2.4 Yogi

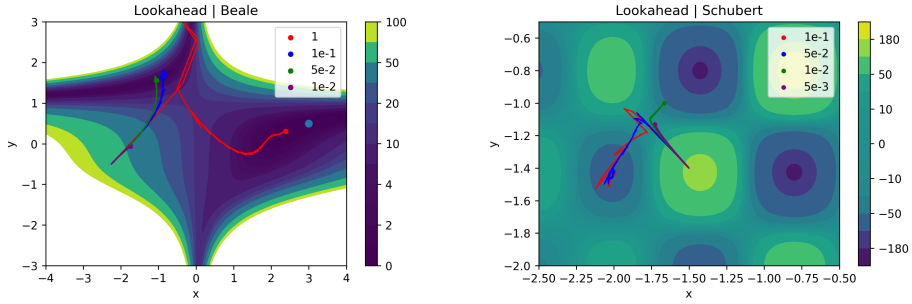
The final adaptive algorithm explored here is Yogi [27], an adaptation of Adam which in a similar manner to AdaGrad alters the update rule for the second moment term in



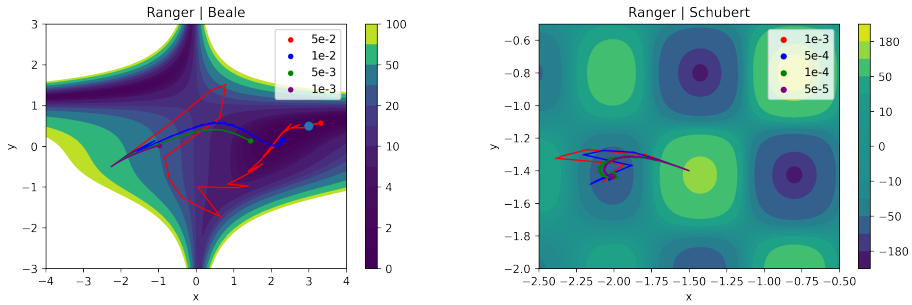
(a) **Yogi**



(b) **SVRG**



(c) **Lookahead**



(d) **Ranger**

Figure 7: Algorithm convergence plots on the Beale and Schubert test functions, trained with 4 different learning rates for 100 iterations. The learning rates illustrate the range which allowed for convergence with values outside of these proving detrimental to the algorithm.

Adam. Yogi sees the insertion of an additional term in computing the second moment for the next iteration (25).

$$v_\theta^{t+1} = v_\theta^t - (1-\beta_2)sign\big(v_\theta^t - \nabla_\theta L(\theta_t, x_i)\big)\nabla_\theta L(\theta_t, x_i) \tag{25}$$

The addition of this *sign* operator has a smoothing effect which prevents the rapid changes in gradient direction in regions of high variance. The effect of Yogi can be seen in 7a where for the Beale function the algorithm prevents the rapid changes of direction which saw Adam jump around at high learning rates, it does however result in a zig-zag motion which is generally inefficient for convergence though ultimately the algorithm is capable of finding the global minimum. The Schubert function sees a similar effect as the convergence path of high learning rates appears more curved, however this does not help in finding the global minimum as all but the smallest rates converge towards the local minimum.

4.3 Variance Reduction Methods

The next class of GD algorithms looks to actively reduce the variance in each gradient estimation without a significant increase in the complexity.

4.3.1 Stochastic Variance Reduced Gradient

The stochastic sampling of GD data points introduces a bias into the gradient estimation, with the bias of a data batch $\{x_i, y_i\}$ given by (26).

$$bias = \mathbb{E}\big[\nabla_\theta L(f(\theta, x_i), y_i)\big] - \nabla_\theta L(f(\theta, x), y) \tag{26}$$

SVRG [28] aims to reduce this bias through an outer and inner loop, utilising a snapshot of the model's parameters. At each outer step, SVRG computes a large batch gradient using $\{x_b, y_b\}$ to obtain an accurate gradient estimation. It then proceeds into the inner loop where it stochastically samples individual data points x_i and uses these to estimate the bias which is then rectified in the final update $\tilde{g_j}$ (27).

$$\tilde{g_j} = \nabla L_\theta(\tilde{\theta}_j, x_i) - \big(\nabla_\theta L(\theta_t, x_i) - \nabla_\theta L(\theta_t, x_b)\big) \tag{27}$$

The RHS of the outer loop attempts to subtract the estimated bias at each iteration. This is only an estimate of the bias, computed according to the batch sampled in the outer loop, hence the estimate can be improved by using a larger batch. However, a larger batch increases the complexity which must be carefully monitored so as to not exceed that of simply computing the full gradient. Following this estimate, the inner weights $\tilde{\theta}$ are updated using the standard GD update rule,

$$\tilde{\theta}_{j+1} = \tilde{\theta}_j - \alpha\tilde{g}_j$$

Finally, the outer parameters are set by the inner loop parameters in the final iteration. In this way, SVRG attempt to emulate a full-batch GD update with lower complexity.

While SVRG achieves excellent theoretical results, naive application to deep-learning problems display the shortcomings of SVRG in the modern scope of ML. Modern practices in ML such as data augmentation, batch normalisation [29] and dropout [30] prove problematic for SVRG as they result in the inner loop snapshot being different at each outer iteration. It has been shown [31] that these result in the variance reduction of SVRG being negligible, questioning its applicability to many modern architectures.

The behaviour of SVRG is well illustrated in figure 7b, where the algorithm requires extremely small learning rates for convergence as it displays erratic and divergent behaviour at rates above those shown. In the case of the Beale function, the algorithm appears to favour a more gradual decline which in this case results with all learning rates converging towards the global minimum. However, as seen from the Schubert function, the algorithm will still fall into steep valleys given sufficient curvature. This may suggest that SVRG is prone to convergence towards sharp minima which would prove to be an unfavourable trait.

4.3.2 Lookahead

Lookahead [32] utilises an inner loop and two sets of weights in a similar manner to SVRG. However, it uses its inner loop as an exploration tool rather than attempting to explicitly compute a variance reduction. The algorithm utilises a set of 'slow' ϕ and 'fast' θ weights, allowing for the attachment of any GD algorithm which it then uses to update the fast weights inside an inner loop according to the update rule of that algorithm. In the original implementation, Lookahead used Adam as the base algorithm which performed the inner loop exploration. After a sufficient number of inner iterations j, Lookahead performs an update to the slow weights by linearly interpolating between them and the fast weights (28).

$$\phi_{t+1} = \phi_t + \alpha(\theta_j - \phi_{t-1}) \tag{28}$$

This update rule allows the algorithm to explore local features using the fast weights, meanwhile the slow weights maintain an anchor to prevent the algorithm becoming stuck in potentially unfavourable minima.

From figure 7c the exploratory mechanism of Lookahead can be seen at work. The Beale plot shows that unlike the convergence of Adam which involved wildly curving loops, Lookahead proceeds in straight lines, leading directly to the next point along the optimisation path instead of following the curvature which is instead explored implicitly by the fast weights. This behaviour is also replicated in the Schubert plot. However, in both cases the descent path appears shorter, travelling a shorter distance in 100 iterations than the other algorithms with similar learning rates. This is caused by the inner loop of the algorithm, which sees the 'slow' weights updated every 5 iterations hence the algorithm only sees 20 parameter updates for 100 iterations. This effect is largely offset by the highly optimal descent direction however it remains noticeable in both plots.

4.3.3 Ranger

The final variance reduction algorithm of interest is Ranger [33]. Ranger builds upon Lookahead, combining it with RAdam and a gradient centralisation method to maximise performance. The primary modification in Ranger involves the inclusion of Gradient Centralisation (GC) [33] which centralises the gradients to have zero mean. Much like how batch normalisation acts on the activation outputs within a network, gradient centralisation acts on the computed gradient at each iteration. The method amounts to an operator ϕ_{GC} (4.3.3) which centralises the gradient.

$$\phi_{GC}\big(\nabla_\theta L(f(\theta, x), y)\big) = \nabla_\theta L(f(\theta, x), y) - \mu_\nabla$$

The GC operator has a regularisation effect in the parameter space, projecting the proposed gradient step onto a hyperplane resulting in a prediction which proves more robust to overfitting. Furthermore, the behaviour of gradient norms is similar to that observed

in batch normalisation which sees a reduction in the Lipschitz constant of the loss function [34] consequently smoothing the loss landscape and improving generalisation. The combination of the GC operator with Lookahead and RAdam proves particularly useful, and hence is proposed as a standalone optimiser.

Ranger displays robustness across different learning rates, as seen in figure 7d, where all rates converge towards the same minimum. In the case of the Beale function, this behaviour is desirable as all plots converge towards the global minimum, with the largest learning rate slightly overshooting it. The Schubert function displays convergence towards the nearest minimum, which in this case is local, with the algorithm taking a much more direct line of descent when compared to the descent of Lookahead. The algorithm also requires lower learning rates for convergence than those seen in Lookahead, behaving more like SVRG.

4.4 Second-Order Methods

Both second-order (SO) and natural-gradient (NG) methods are a form of preconditioned GD, utilising a matrix of local curvature information to condition the computed gradient in an attempt to yield a more optimal descent direction. SO methods receive a brief overlook as they are sparsely mentioned in modern optimisation literature, owing largely to their significant computational overhead which prohibits applicability. SO methods look to use the Hessian matrix of second order derivatives to precondition the gradient term used in the update.

Computing the optimal direction with access to second-order derivative involves formulating the objective function as a second-order Taylor expansion and then minimising it.

$$\begin{aligned} L(\theta) &= L(\theta + d) + \nabla_\theta L(\theta)^T d + \frac{1}{2}(\theta - \theta + d)^T H(\theta - \theta + d) \\ &= L(\theta + d) + \nabla_\theta L(\theta)^T d + \frac{1}{2} d^T H d \end{aligned}$$

Taking the derivative w.r.t d, setting this to zero and solving.

$$\nabla_d L(\theta) = 0 + \nabla_\theta L(\theta) + Hd$$

$$0 \ = \nabla_\theta L(\theta) + Hd$$

Solving for d, the optimal descent direction is obtained by rearranging the above.

$$d = -\frac{1}{H} \nabla_\theta L(\theta) \tag{29}$$

The descent direction (29) utilises the Hessian matrix $H \in \mathbb{R}^{d \times d}$, with d being the number of parameters in the model. The Hessian size scales quadratically with the size of the model and soon becomes intractable to compute and invert, invalidating this method for most models. As a result, many newer SO methods suggest computing the preconditioner without explicitly computing the Hessian itself, these are referred to as Hessian-Free (HF) methods. Such HF methods remain relatively complex compared to first order methods, however they make up for this with more optimal descent steps. The main limitation of these methods is their use of the Conjugate Gradient, which requires the curvature matrix to remain fixed [37] while the algorithm iterates. This means that HF algorithms process data at a much slower rate, causing them to remain prohibitively slow for large datasets. As will be shown below, Natural Gradient offers another preconditioning method which is not bound by such limitations.

4.5 Natural Gradient Methods

The GD methods described above exist entirely within the parameter space from which they obtain their gradient information and consequently navigate. Natural gradient (NG) methods shift this approach into the distribution space, where the optimisation problem aiming to minimise the loss is equivalent to one maximising the likelihood of the correct prediction by the model. While the Hessian matrix is used to describe the local curvature in parameter space, NG methods use the Fisher information matrix to describe the local curvature of the distribution space [35]. The objective function used in the distribution space is the Kullback–Leibler (KL) divergence (30), which measures how close two distributions are in a probability space X.

$$KL(f(\theta)||f(\tilde{\theta})) = \sum_{x \in X} f(\theta, x) \log\Big(\frac{f(\theta, x)}{f(\tilde{\theta}, x)}\Big) \tag{30}$$

In this case, the target distribution is the underlying distribution of the data which the model attempts to learn. The NG descent step is derived similarly to the SO method; however, it must now contend with the various features of the distribution space in which NG methods operate. The optimal direction of descent can be derived from the following minimisation problem, where the KL divergence is fixed to a constant c providing a more robust formulation which is not affected by reparameterization [36] as it only concerns itself with the distributions induced by the model parameters.

$$d^* = \min_d \;\; L(\theta + d)$$

The RHS of the above expression can be written in its Lagrangian form with the constrained KL divergence from which the proof proceeds by taking the first and second order Taylor expansions of $L(\theta + d)$ and the KL divergence respectively, with F representing the Fisher matrix (31). The full derivation of the Taylor expansion is provided in Appendix B.

$$\begin{aligned} d^* &= \min_d \;\; L(\theta + d) + \lambda\Big(KL\big(f(\theta||f(\theta + d)\big) - c\Big) \\ &= \min_d \;\; L(\theta) + \nabla L(\theta)^T d + \frac{1}{2}\lambda d^T F d - \lambda c \\ F &= \mathbb{E}\Big[\nabla \log\big(f(\theta, x)\big) \nabla \log\big(f(\theta, x)\big)^T\Big] \end{aligned} \tag{31}$$

Taking the derivative w.r.t d, setting to 0 and then solving for d gives the steepest descent vector.

$$0 = \nabla L(\theta) + \lambda F d$$

$$d = -\frac{1}{\lambda}\frac{\nabla L(\theta)}{F}$$

The λ term is dropped as it can be incorporated into the learning rate, leaving the definition of the natural gradient (32).

$$\tilde{\nabla} L(\theta) = F^{-1} \nabla L(\theta) \tag{32}$$

The natural gradient (32) is then used to precondition the computed gradient of the model's loss function w.r.t the model parameters, with an update step of the form (33).

$$\theta_{t+1} = \theta_t - \alpha \tilde{\nabla} L(\theta_t) \tag{33}$$

This naive implementation is intractable for all but the smallest of models, for the same reasons as was made apparent for SO method above. There do however exist some methods which aim to efficient approximate and invert the Fisher matrix F, to make the NG algorithm applicable to larger problems. These methods cannot be effectively visualised using the test functions mentioned previously as they work within the distribution space, hence this is omitted for the following algorithms.

4.5.1 Kronecker-factored Approximate Curvature

To allow the application of NG to large scale problems, the algorithm requires an efficient way of approximating and inverting the Fisher matrix. Kronecker-factored Approximate Curvature (KFAC) [37] does this by utilising the Kronecker product (34) to provide an accurate approximation.

$$A \in \mathbb{R}^{m\times n},\ B \in \mathbb{R}^{p\times q}\ :\ A \otimes B\ \in \mathbb{R}^{mp\times nq} \tag{34}$$

The main property of this function is that the inverse of a Kronecker matrix (34) can be computed by inverting the smaller constituent matrices (35), greatly simplifying the inversion process.

$$\big(A \otimes B\big)^{-1} = A^{-1} \otimes B^{-1} \tag{35}$$

Instead of computing the entire Fisher matrix, KFAC aims to directly compute an inverted diagonal approximation with a series of blocks corresponding to each network layer along the matrix diagonal.

Dealing with diagonal blocks, the algorithm computes a Fisher (31) approximation for each layer of the network. The Fisher matrix can be broken down into a composition of the gradient of the output of each layer g_i and the activation output of the previous layer a_{i-1}.

$$\nabla_i \log\big(f(\theta, x)\big) = g_i \otimes a_{i-1}$$

Using the properties of the Kronecker product (34), an approximation of the Fisher can be formulated using these gradients and outputs.

$$\begin{aligned} F &= \mathbb{E}\Big[(g_i \otimes a_{i-1})(g_i \otimes a_{i-1})^T\Big] \\ &= \mathbb{E}\Big[(g_i g_i^T) \otimes (a_{i-1} a_{i-1}^T)\Big] \\ &= \mathbb{E}\Big[g_i g_i^T\Big] \otimes \mathbb{E}\Big[a_{i-1} a_{i-1}^T\Big] = G_i \otimes A_{i-1} \end{aligned}$$

From here, the inversion of the Fisher matrix simply requires the inversion of the G_i and A_{i-1} matrices which are much smaller and hence the inversion can be performed efficiently, with complexity scaling as $\mathcal{O}(d)$ rather than $\mathcal{O}(d^2)$. The approximated Fisher matrix is then used in preconditioning the gradient information to compute the natural gradient (32) which finally updates the model parameters (33).

4.5.2 Shampoo

Another method similar to KFAC is Shampoo [38], aptly named as it preconditions the gradient. Shampoo aims to perform the preconditioning in a piece-wise fashion, where considering the gradient as a matrix of values, the algorithm computes two statistics which are then applied to the columns and rows of the gradient matrix. Taking G_t to be the gradient computed at iteration t, the algorithm then computes L_t and R_t as follows,

$$L_t = L_{t-1} + G_t G_t^T$$

$$R_t = R_{t-1} + G_t^T G_t$$

The parameter update (36) then uses these to precondition the gradient, with α being the learning rate.

$$\theta_{t+1} = \theta_t - \alpha L_t^{-1/4} G_t R_t^{-1/4} \tag{36}$$

Again, as in KFAC the inversion of the preconditioners can be done effectively as the algorithm deals with two smaller matrices rather than a single large matrix. The utilisation of second moment information for the computation of preconditioners L_t and R_t also sheds some light on the remarkable success of Adam, which similarly uses a decaying average of second moments of the gradient. This can be thought of as a rough approximation of the Fisher matrix which endows Adam-like methods with some natural gradient information allowing them to make a more intelligible decision on its descent direction.

5 Methodology

5.1 Project Management

This project followed a clear goal from the outset, looking to compare novel optimisation algorithms against each other, motivated by the lack of such comparisons in modern ML literature. Beginning with an extensive literature review of the field, the initial months were largely concerned with learning the theoretical fundamentals in the field of numerical optimisation which are used in formulating ML optimisation algorithms. Upon transitioning into the next stage which saw the beginning of test runs, it was realised that the currently available hardware was insufficient for all but the smallest of models, requiring a more powerful GPU for the training stage. As a result, the computational stage of the project was moved into the cloud using Google cloud services which provided a flexible virtual machine allowing for the modification of the RAM, GPU, and CPU according to the project requirements. Most of the code was executed inside a VM utilising a Nvidia Tesla T4 GPU along with 15GB of RAM which proved sufficient for most algorithms.

5.2 Generalisation Performance

Considering now the techniques used for evaluation, the most fundamental metric of any optimisation algorithm looks at the performance of the final ML model. In the case of image classification this is the test accuracy while for the language modelling task this is the perplexity. Beyond these it is also important to consider the run-time of the algorithm along with any memory requirements. As these could depend on machine hardware, they are evaluated relative to each of the other algorithms executed on the same machine. Most algorithms prove consistent in their run-time and memory consumption as will be illustrated in the coming sections.

5.3 Loss Surface Visualisation

Beyond the generalisation performance, there exist additional tools with which it is possible to gain further insights into the training process. The loss surface of deep neural networks is high dimensional, with the number of dimensions corresponding to the number of parameters within the model which can reach billions with modern state-of-the-art architectures. Therefore, it is not possible to visualise the actual loss landscape. There do however exist techniques for plotting local information about the loss surface, these have been widely used in ML literature for exploring the surface around the convergence

Algorithm	Gradient Update	Term Definitions
Baselines		
SGD	$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L$	
Adam	$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_\theta}{\sqrt{\hat{v}_\theta} - \epsilon}$	$\hat{m}_\theta = \frac{\beta_1 m_\theta^{t-1} + (1-\beta_1)\nabla_\theta L}{1-\beta_1}$ $\hat{v}_\theta = \frac{\beta_2 v_\theta^{t-1} + (1-\beta_2)\nabla_\theta L}{1-\beta_2}$ $\beta_1, \beta_2 \in [0, 1)$
Adaptive Methods		
RAdam	Low Var: $\theta_{t+1} = \theta_t - \alpha \hat{m}_\theta$ High Var: $\theta_{t+1} = \theta_t - \frac{\alpha r_t \hat{m}_\theta}{\sqrt{\hat{v}_\theta} - \epsilon}$	$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}$
AMSGrad	$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_\theta}{\sqrt{\hat{v}_\theta} - \epsilon}$	$\hat{v}_\theta = \max\left(\hat{v}_\theta, v_{t-1}\right)$
Yogi	$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_\theta}{\sqrt{\hat{v}_\theta} - \epsilon}$	$\hat{v}_\theta = \frac{v_t - (1-\beta_2)\text{sign}\left(v_t - \nabla_\theta L\right)\nabla_\theta L}{1-\beta_2}$
Variance Reduction		
SVRG	$\theta_{t+1} = \theta_t - \alpha \tilde{g}_t$	$\tilde{g}_t = \nabla_\theta L_i - (\nabla_\theta L_i - \nabla_\theta L_x)$
Lookahead	$\theta_{t+1} = \theta_t - \alpha \tilde{g}$	$\tilde{g} = \theta_t^{inner} - \alpha \theta_{t-1}$
Ranger	$\theta_{t+1} = \theta_t - \alpha \phi_{GC}\left(\tilde{g}\right)$	$\phi_{GC}\left(\nabla_\theta L\right) = \nabla_\theta L - \mu_\nabla$ Low Var: $\tilde{g} = \hat{m}_\theta$ High Var: $\tilde{g} = \frac{r_t \hat{m}_\theta}{\sqrt{\hat{v}_\theta} - \epsilon}$
Natural Gradient		
KFAC	$\theta_{t+1} = \theta_t - \alpha \tilde{\nabla}_\theta L$	$\tilde{\nabla}_\theta L = F^{-1} \nabla_\theta L$ $F = \mathbb{E}\left[g_i g_i^T\right] \otimes \mathbb{E}\left[a_{i-1} a_{i-1}^T\right]$
Shampoo	$\theta_{t+1} - \theta_t - \alpha L_t^{-1/4} \nabla_\theta L R_t^{-1/4}$	$L_t = L_{t-1} + \nabla_\theta L (\nabla_\theta L)^T$ $R_t = R_{t-1} + (\nabla_\theta L)^T \nabla_\theta L$

Figure 8: The compilation of gradient update rules for each of the algorithms considered above.

point [34, 40, 41, 42]. The primary goal of this method is to investigate the width of the minima towards which various algorithms converge. With support for the superior generalisation power of wide and flat minima [4, 5], these can be considered to be the desirable points of convergence.

5.3.1 1-D Linear Interpolation

Linear interpolation can be used to produce a 1-D plot of the loss function. This is done by selecting two sets of network parameters, in this case θ^* are selected to be the parameters of the converged networks while θ_1 is taken to be a random direction within the parameter space obtained by adding a perturbation to the weights of a newly initialised model.

$$L(\hat{\alpha}) = L\Big(\hat{\alpha}\theta^* + (1-\hat{\alpha})\theta_1\Big) \tag{37}$$

The function (37) then linearly interpolates, using $\hat{\alpha}$, between the two sets of parameters. Care should be taken in interpreting these results as they provide a dimensionality-reduced picture of the local surface and may overlook the significant non-convexities inherent in high-dimensional parameter spaces.

5.3.2 2-D Contour Plots

This idea can be extended to two dimensions, taking two separate random directions θ_1, θ_2 within the parameter space, produced using the perturbation described above, with θ^* being the parameters of the converged model. These can then be interpolated to produce a 2-D plot which provides more information.

$$L(\hat{\alpha}, \hat{\beta}) = L\Big(\theta^* + \hat{\alpha}\theta_1 + \hat{\beta}\theta_2\Big) \tag{38}$$

This method allows for a more detailed view of the local loss surrounding the convergence point, illustrating the type of minima towards which different optimisers converge. A wider minimum of low loss indicates that the model is robust to small changes in its parameters which could suggest better generalisation.

5.4 Second-Order Information

While the view afforded by visualisation methods involves significant dimensionality-reduction, second-order information such as the Hessian matrix allow a more direct look at the properties of the local curvature. Computing the entire Hessian is not feasible for most models, instead its eigenvalues can provide extensive information about the loss surface.

Firstly, looking directly at the eigenvalues themselves allows for the sharpness of the local loss surface to be deduced, as low ($|\lambda| \sim 0$) eigenvalues signify a flat surface lacking curvature meanwhile large ($|\lambda| > 100$) values are indicative of steep (positive or negative) curvature. Looking at individual eigenvalues provides limited information as these correspond to individual dimensions within the parameter space which may not be representative of the the remaining dimensions. Instead, the coming sections will look at the eigenvalue density obtained by computing 200 randomly sampled eigenvalues from the spectrum to obtain a good snapshot of the magnitudes looking keenly for any outliers. These are computed using the Stochastic Lanczos Quadrature algorithm, used in numerous studies concerning the loss surface of various neural network architectures [55, 56, 57].

5.5 Code

To facilitate the comparison across different algorithms, the project required a code base which could perform repeated training runs on different models, iterating over hyperparameter settings. In line with most ML research today, this was written in Python due to its wide range of data handling and ML libraries. The library of choice was PyTorch [58], with its widespread use in ML it allows for the implementation of custom optimiser classes which proved beneficial.

The training runs were executed on a Google Cloud virtual machine, utilising a Nvidia Tesla T4 GPU with 16 GB of GPU memory. The main memory requirement during training stems from the backward pass of the backpropagation algorithm which computes the gradient of each layer within the network, 16GB proved sufficient for all the models. The use of a virtual machine allowed the hardware to be scaled up where needed. This was only needed in the case of SVRG which required 30GB of RAM as it maintained several versions of the network parameters. The remaining algorithms all utilised less than 15GB RAM for all their runs.

6 Empirical Analysis

6.1 Image Classification

This section begins with the image classification task which looks at classifying 60,000 images from the CIFAR100 dataset to any of its 100 classes, providing a sufficiently difficult task to effectively test each GD optimiser. Below are presented the in-depth results for the ResNet18 architecture while the following section presents a more high-level overview of the other architectures including EfficientNet(B0), InceptionV3 and VGG11. Prior to the training run, each data batch underwent a standard data augmentation procedure involving a randomized crop followed by a random horizontal flip of the image. Methods such as these are widely used in computer visions problems to improve model performance.

6.1.1 ResNet18

The ResNet [18] architecture has proven to be very effective in computer vision problems, introducing skip connections which reuse activations from previous layers in an attempt to reduce model complexity and overfitting. There exist numerous ResNet architectures of varying depth, this study focuses on ResNet18, possessing 11.7 million parameters it ranks as one of the smaller architectures however, regardless of this it proves capable of obtaining a remarkably good test performance of over 70% after its 40 epoch run. This section now looks more closely at each algorithm in turn.

Looking at the performance of **SGD**, the algorithm performs best at higher learning rates with its optimal accuracy being 59.20%. As illustrated by the contour plots in figure 9, these converge towards wider and flatter minima which are expected to generalise better. The spectrum of eigenvalues provides a further probe into the geometry of the loss surface with flat regions producing low eigenvalues centred around zero, while sharp regions around the minimum result in large and positive eigenvalues. This can be seen in figure 9 where the learning rate of 0.01 which generalises poorly also produces large outlying eigenvalues within its spectrum, indicating sharp curvature.

Adam finds itself within a region surrounded by walls of extremely high loss, limiting the visualisation ability of contour plots as the resolution suffers. To better visualise

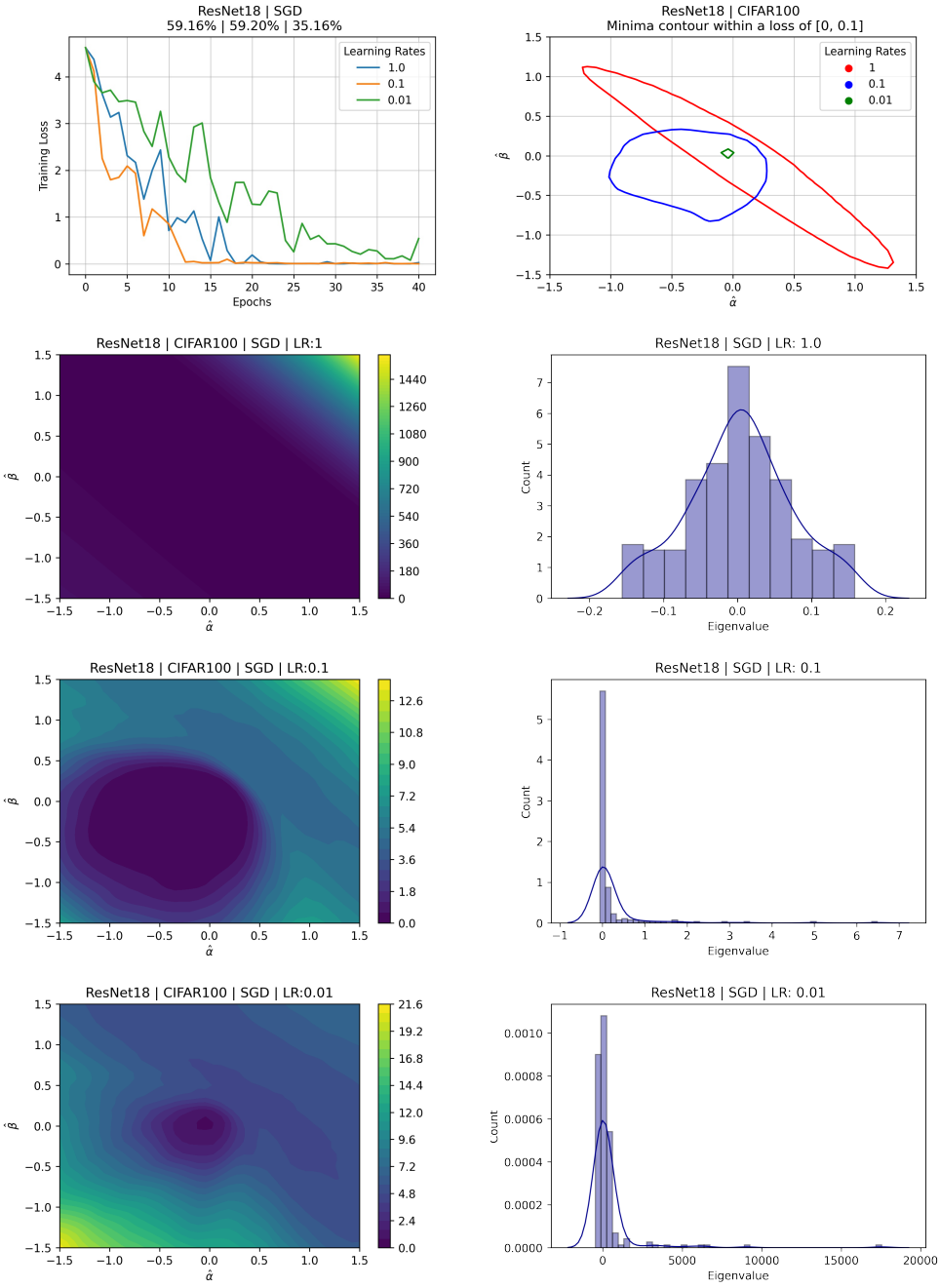


Figure 9: **SGD** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 0.1] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

the surface geometry at low loss, the plots in figure 10 only indicate the region within the loss range of $[0, 10]$, with any white region existing at a greater loss. The behaviour of Adam appears contrary to that of SGD, performing better at low learning rates and generalising poorly at higher rates. Ultimately, Adam achieves a superior test accuracy of 69.12% with a learning rate of 0.001, outperforming SGD by 10%, a significant margin which illustrates the utility of adaptive methods. Interestingly, while the minimum of the poorly performing learning rate appears markedly sharper according to the spectrum of eigenvalues, the optimal minimum appears sharper than that of the middle rate. This is illustrated in both the contour plots and the eigenvalue spectrum, which could indicate the tenuity of the connection between minimum sharpness and model generalisation, instead indicating that there exists a sufficient width beyond which minima do not benefit from further flatness.

RAdam puts on a superior performance, obtaining a test accuracy of 71.68% which ranks higher than both SGD and Adam. From the plots in figure 11 both sets of learning rates are seen to converge at a similar speed, although they appear to converge towards different features within the loss landscape. The lower learning rate converges towards a flatter region as indicated by the contour plot and the low eigenvalues within the density plot, meanwhile the higher learning rate of 0.0001 converges towards a point possessing extremely large eigenvalues of more than 2000. This indicates extremely sharp curvature along individual dimensions within the model's parameter space which prove detrimental to the model's performance.

Yogi performs similarly to RAdam, with the optimal learning rate around 0.001 and a test accuracy of 71.05%. The contour plots in figure 12 display the robustness of the middle learning rate to perturbations within the parameter space. Furthermore, the loss contour of the optimal learning rate appears wider than that of the other rates, corresponding to the best generalisation performance for the model. Compared to Adam, Yogi much like RAdam appears more robust to changes in its hyperparameters such as the learning rate. The spectral eigenvalue density displays increasingly large outlying eigenvalues for smaller learning rates, these however are few, corresponding to a small number of dimensions which see sharp curvature and the algorithm is still capable of good generalisation, outperforming the best of SGD which converged towards much flatter regions of the loss surface.

The last of the adaptive algorithms is **AMSGrad** which appears to perform worse than the other adaptive algorithms. Its optimal accuracy of 68.98% is the lowest, with performance close to that of unmodified Adam. It does however display greater robustness to the learning rate, with similar performance for all rates. Its Hessian plots in figure 13 appear contrary to the contour plots above them, indicating a disparity between the two which could stem from the limitation of the contour plots being a 2D projection from a much higher dimensional parameter space.

Each of the three adaptive algorithms does well to illustrate their ability to adapt the different learning rates to improve model performance, obtaining consistently high accuracies across the different rates relative to the fluctuations in accuracy seen for SGD and Adam. In the case of SGD this is due to the lower rates stalling the training process and preventing any further progress while in Adam it's the larger rates which amplify variance proving particularly detrimental in the early stages where it can hinder convergence.

Moving onto the variance reduction methods, **SVRG** performs worse than SGD obtaining an accuracy of 57.59%, with the eigenvalue plots in figure 14 possessing large eigenvalues of around 2000 indicating significant sharpness in the curvature surrounding

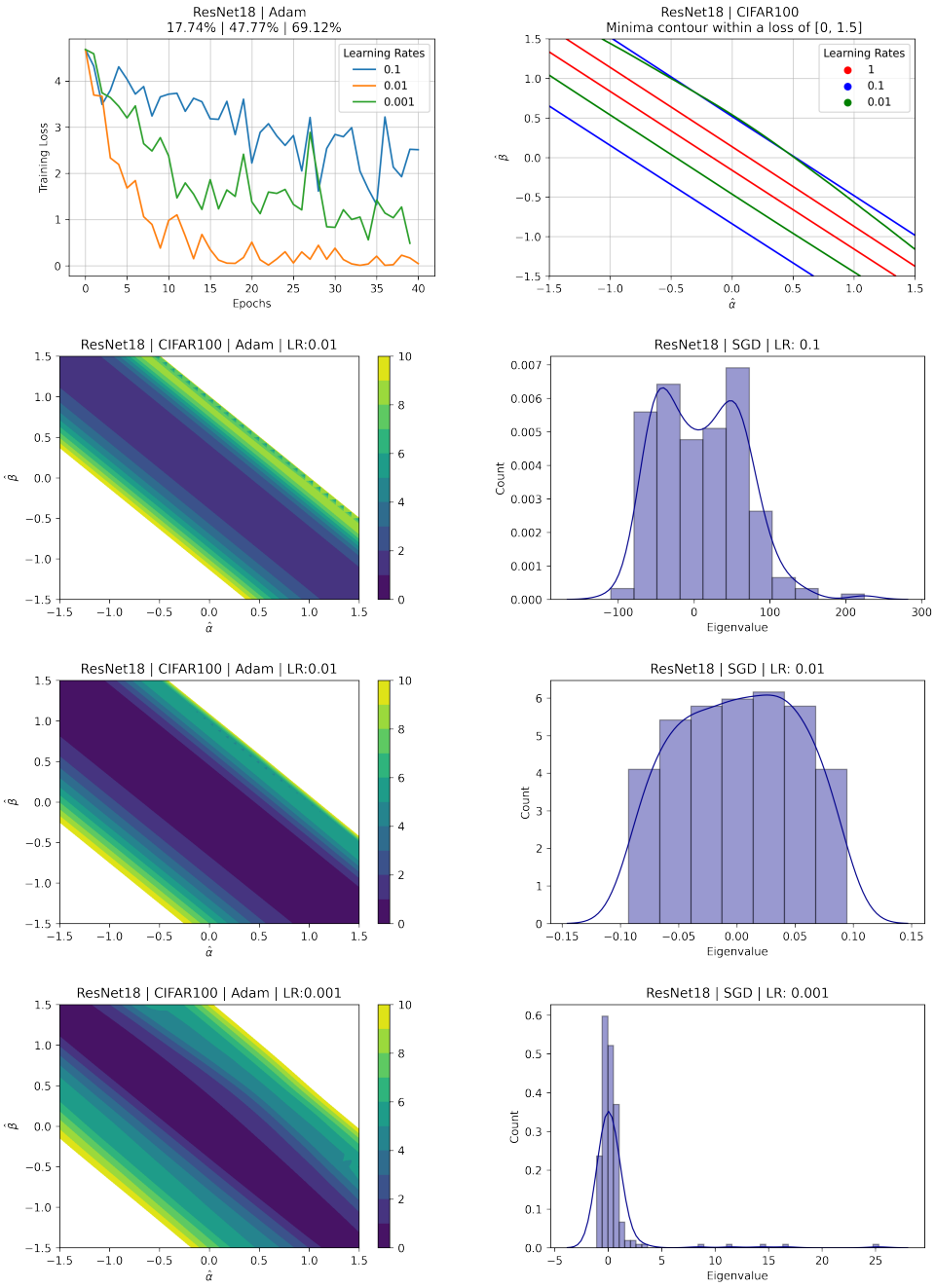


Figure 10: **Adam** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1.5] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

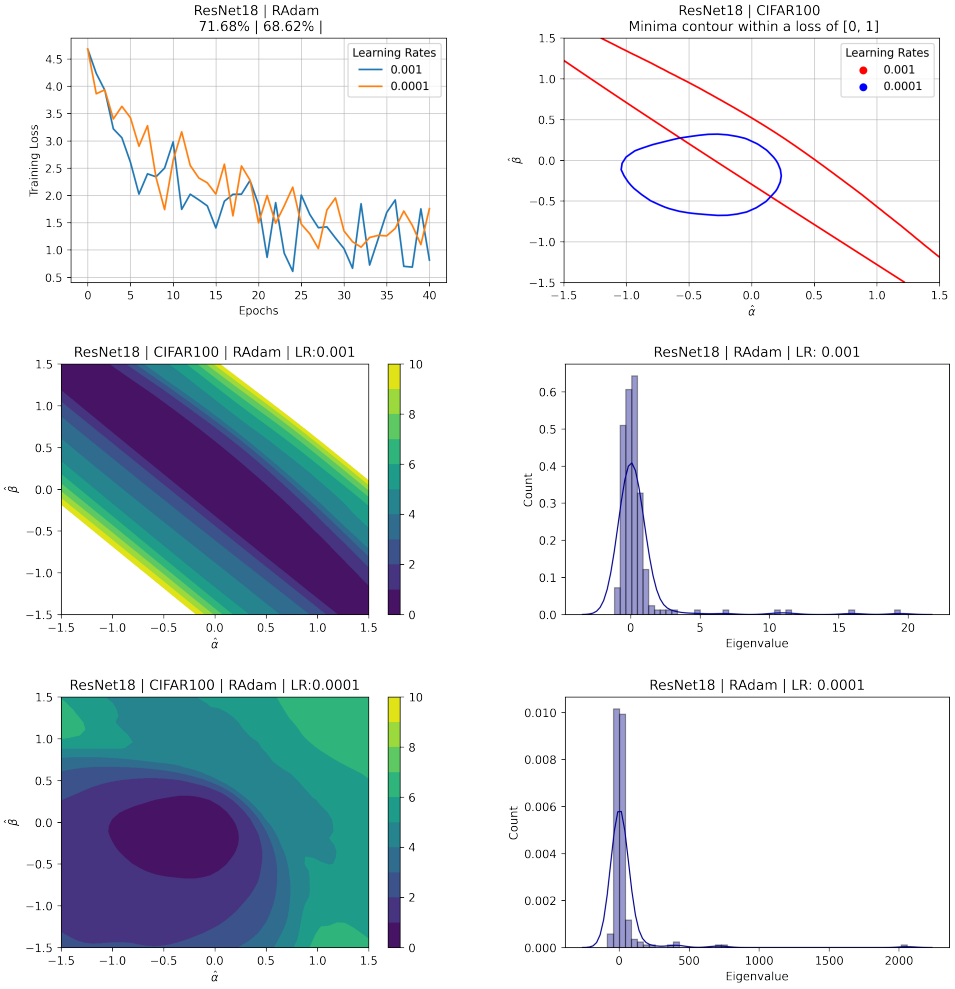


Figure 11: **RAdam** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

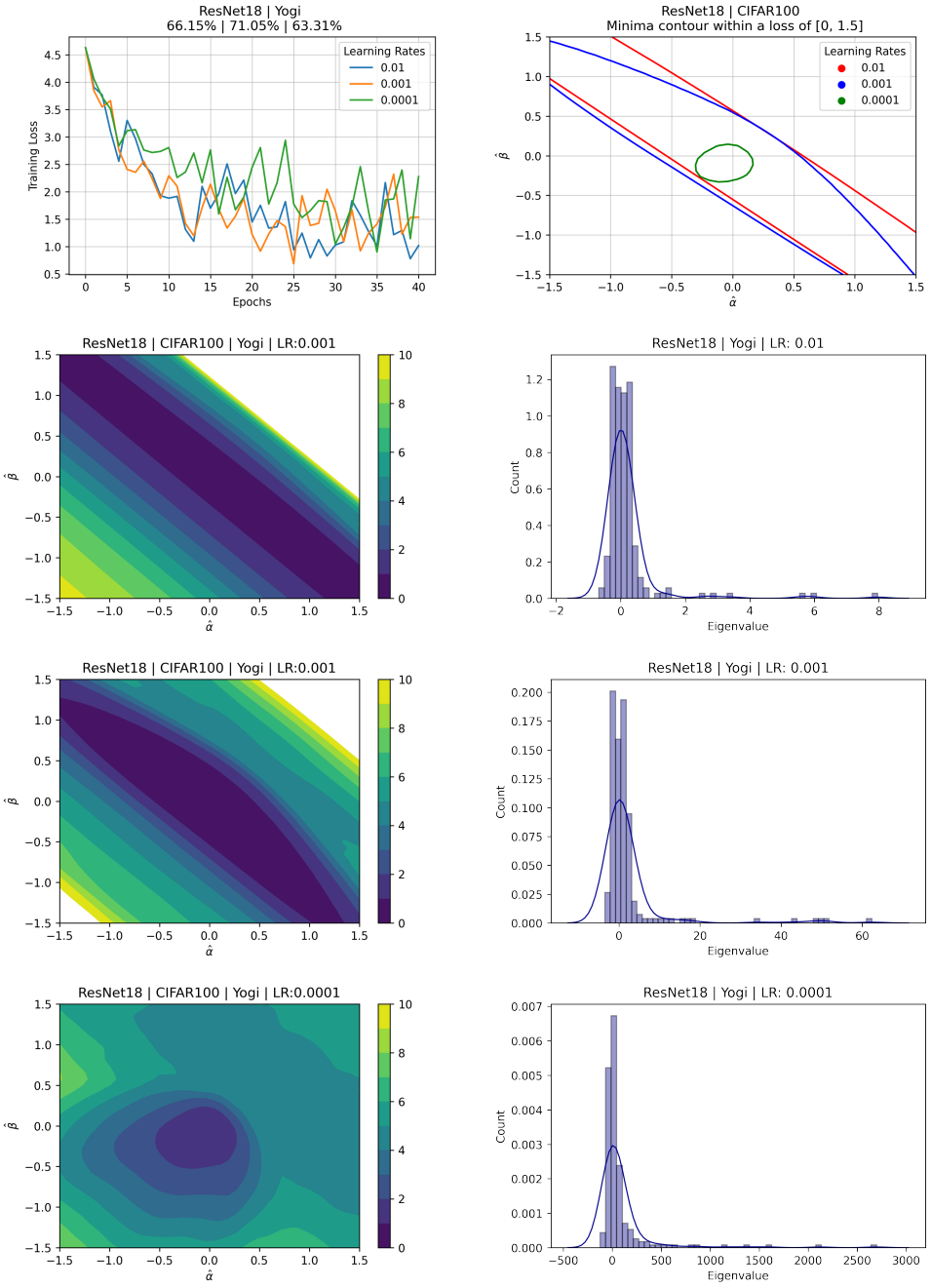


Figure 12: **Yogi** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1.5] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

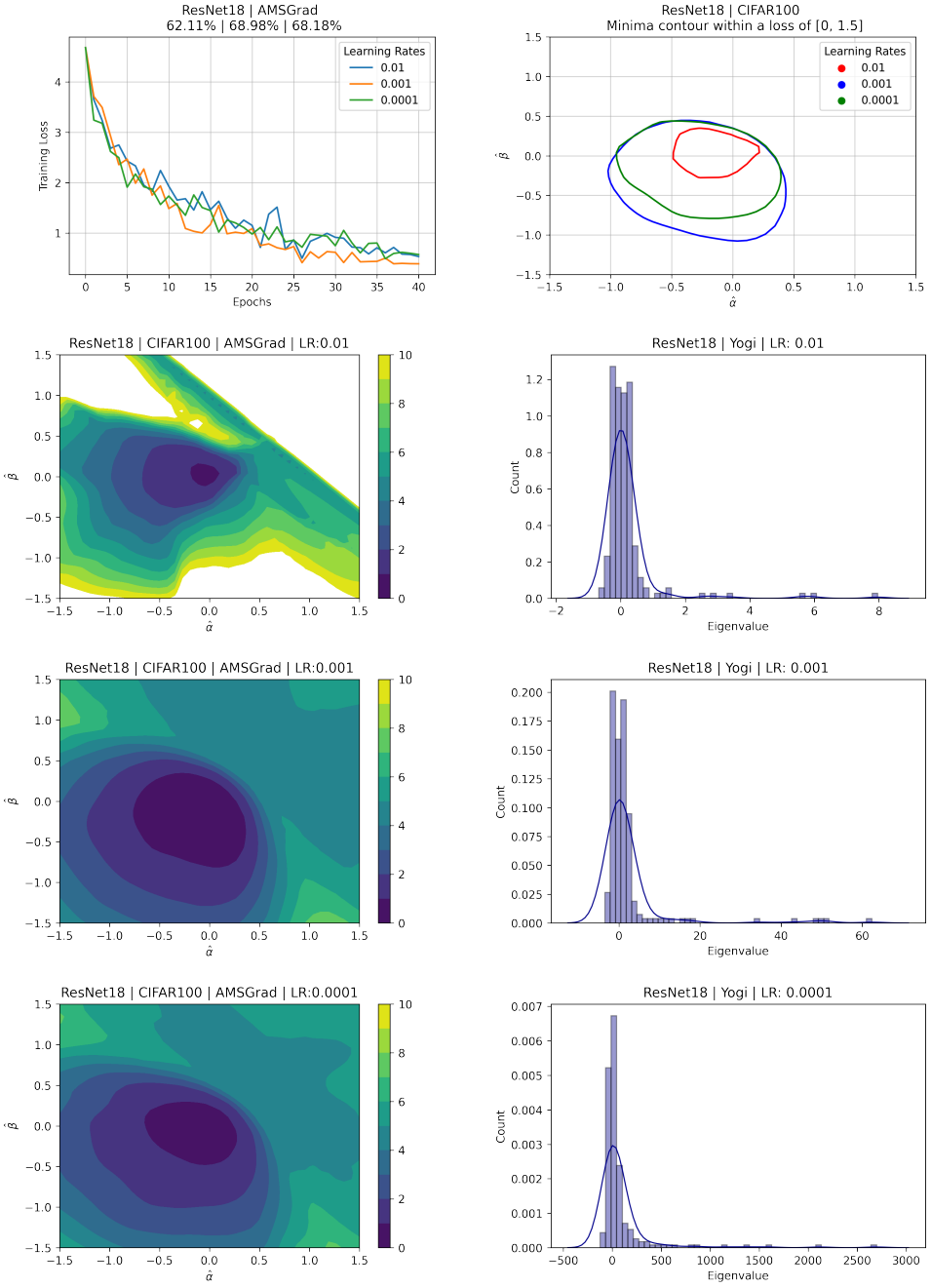


Figure 13: **AMSGrad** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1.5] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

each of the convergence points. Furthermore, the algorithm proves very costly to run, utilising a significant amount of memory due to the inner loop which requires the storage of a snapshot of the model. In the case of ResNet18 with 12 million parameters, SVRG required 30GB of RAM during its execution, this grew rapidly for larger models such as InceptionV3 and VGG11 quickly proving intractable.

Lookahead appears to be a more effective variance reduction method, obtaining a good test accuracy of 70.22%, accompanied by a wide minimum in figure 15, with other learning rates obtaining narrower minima. Furthermore, the simpler design of Lookahead does not result in any significant memory consumption compared to base Adam due to the linear interpolation method which updates the model parameters hence unlike SVRG, Lookahead scales well to larger models providing consistent improvements over Adam. The algorithm runs at the same speed as the others, with a 40 epoch run taking around 4 hours on a Tesla T4 GPU.

The final variance reduction algorithm is **Ranger**, which obtains a test accuracy of 73.02% with a learning rate of 0.001 converging towards a wide minimum. The smaller learning rate converges more slowly than the others resulting in a narrow minimum with some extreme eigenvalues within the Hessian. Remarkably even with these properties Ranger's worst result still outperforms SGD and it does so with no computational overhead, completing its 40 epoch run in around 4 hours like the others. Ranger appears to converge towards wider minima at the higher rates as seen in figure 16, these correspond to better generalisation.

As the first of the Natural Gradient methods, **KFAC** achieves an accuracy of 72.96% proving very similar to Ranger, furthermore it also displays improved robustness to the learning rate parameter with the accuracy across different rates being around 70.00%. This is best illustrated by the consistently low eigenvalues in the spectrum of each learning rate in figure 17, unlike other algorithms which tend to see spikes in eigenvalues at different rates. The disparity across the eigenvalue plots in figure 17 suggests that the learning rates converge towards different minima, where the optimal minimum is able to generalise well even with some sharpness.

Finally, **Shampoo** is the worst performing algorithms seen thus far, with an optimal accuracy of 57.03% the algorithm is outperformed by SGD. Surprisingly, as seen in figure 18, Shampoo is able to converge towards a region of low loss at later epochs, with the contour plots displaying a wider minimum for the lower learning rates which achieve better accuracy. However, even these minima result in poor generalisation. The main downside of Shampoo is its complexity, with the 40 epoch run taking more than double the time of any of the previously seen algorithms which completed their training in 4 hours. This is caused by the computation of the inverted pre-conditioner matrices which is done using Singular Value Decomposition, even with the acceleration afforded by a GPU this operation proves prohibitively slow.

The results of the ResNet architecture indicate that the relationship between the sharpness of minima and the generalisation performance of the model is not yet clear. In figure 18, Shampoo appears to converge towards generally flat minima according to its eigenvalues, however, it ultimately fails to generalise well as indicated by the poor test accuracy when compared to other algorithms. This is likely due to the region of the loss surface existing at a higher loss value which consequently results in poor test performance regardless of its relative flatness. Given that ResNet18 is a relatively small model, Shampoo is unlikely to find applications in larger models as it proves prohibitively slow to run and ultimately its performance proves worse than that of baseline algorithms. The case is similar for SVRG, with its excessive memory requirement which grows intractable for larger models. As a result, both SVRG and Shampoo are eliminated from

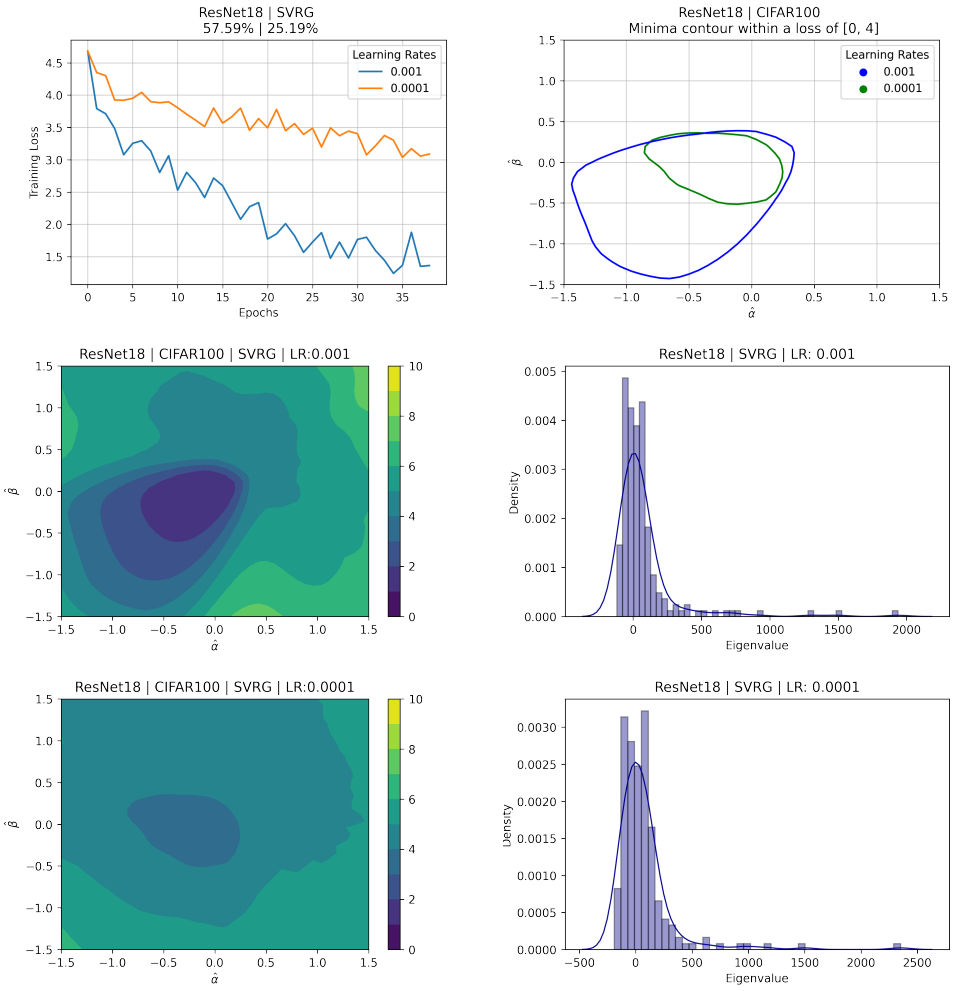


Figure 14: **SVRG** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 4] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

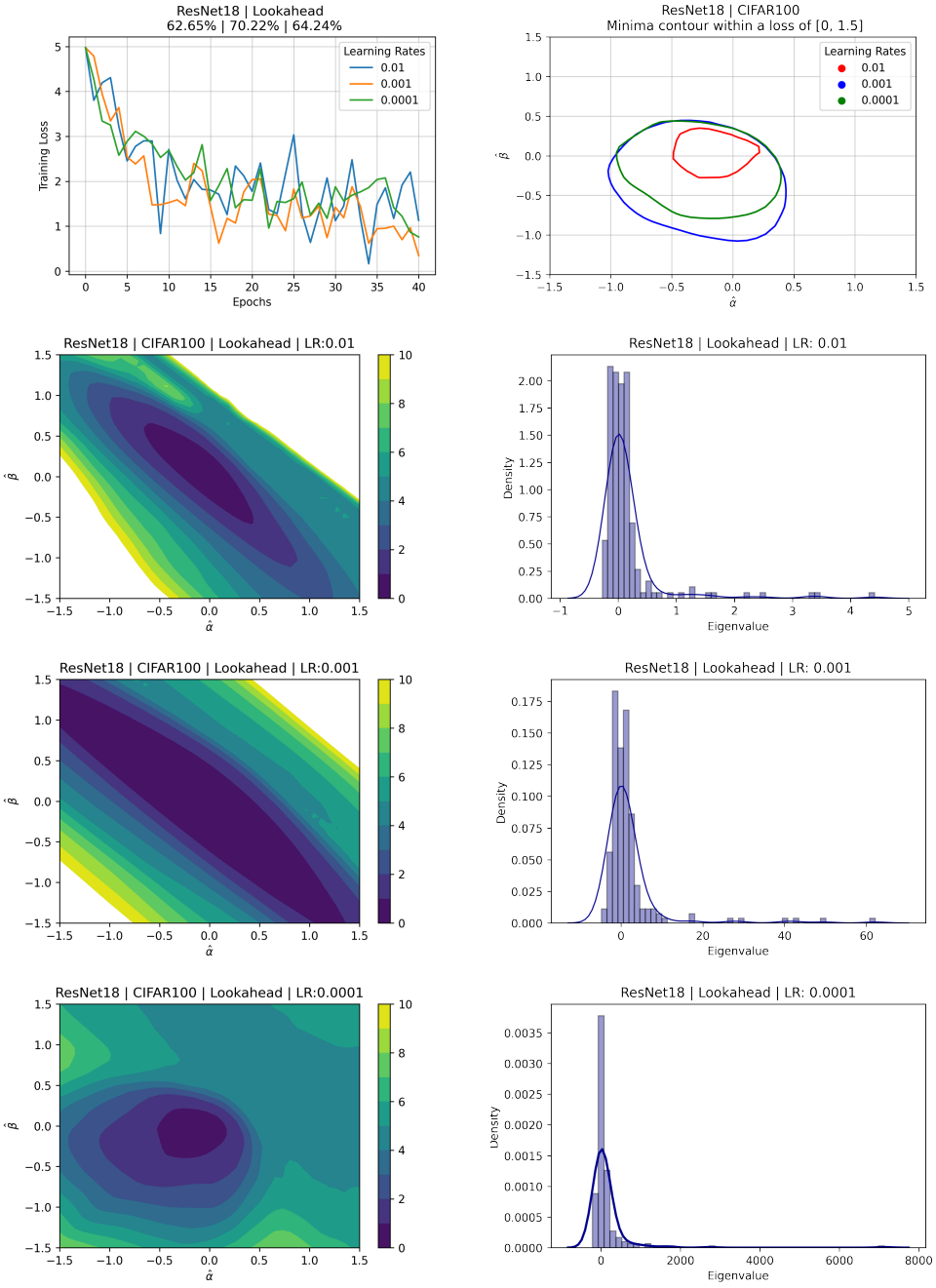


Figure 15: **Lookahead** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1.5] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

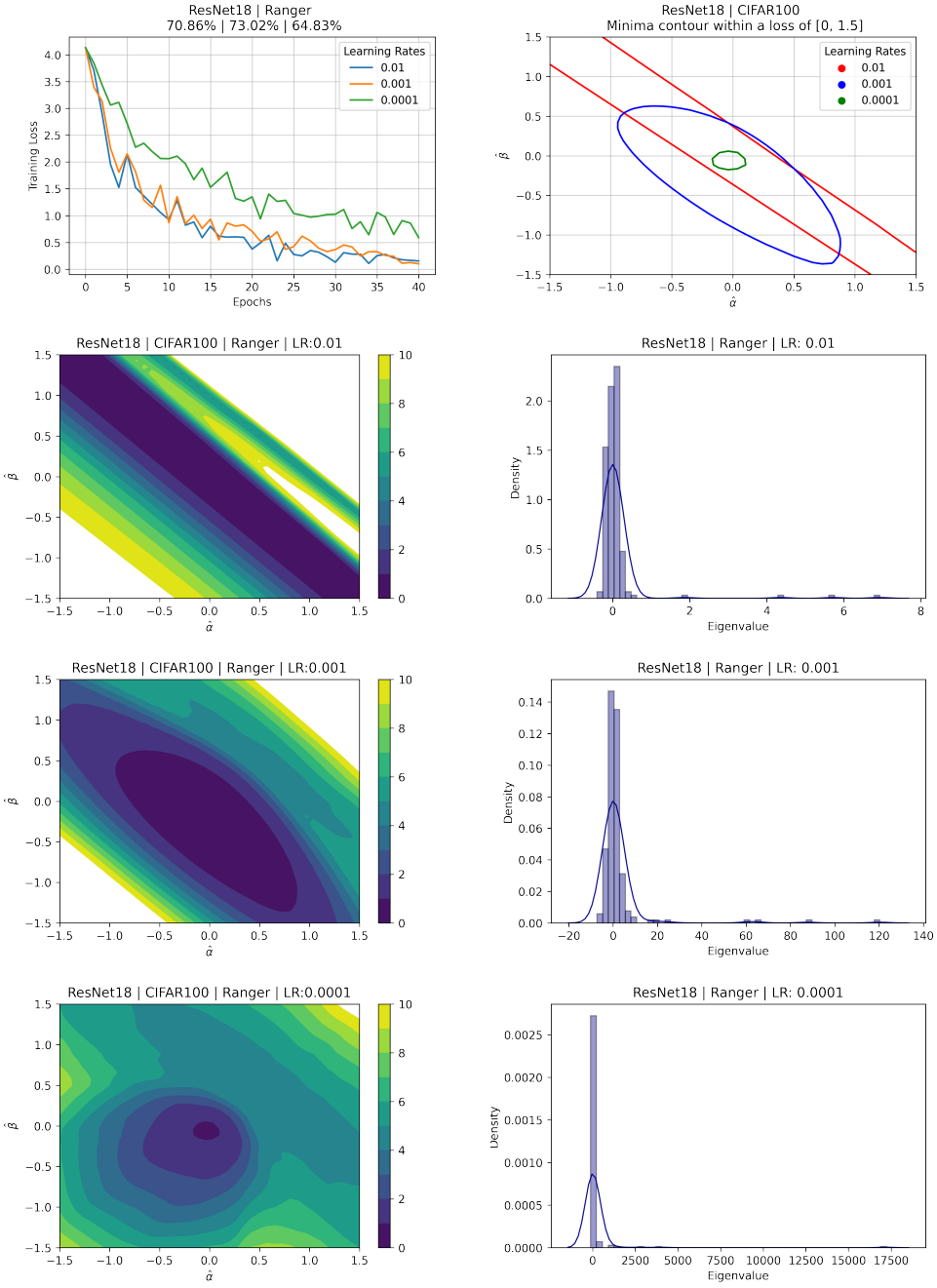


Figure 16: **Ranger** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1.5] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

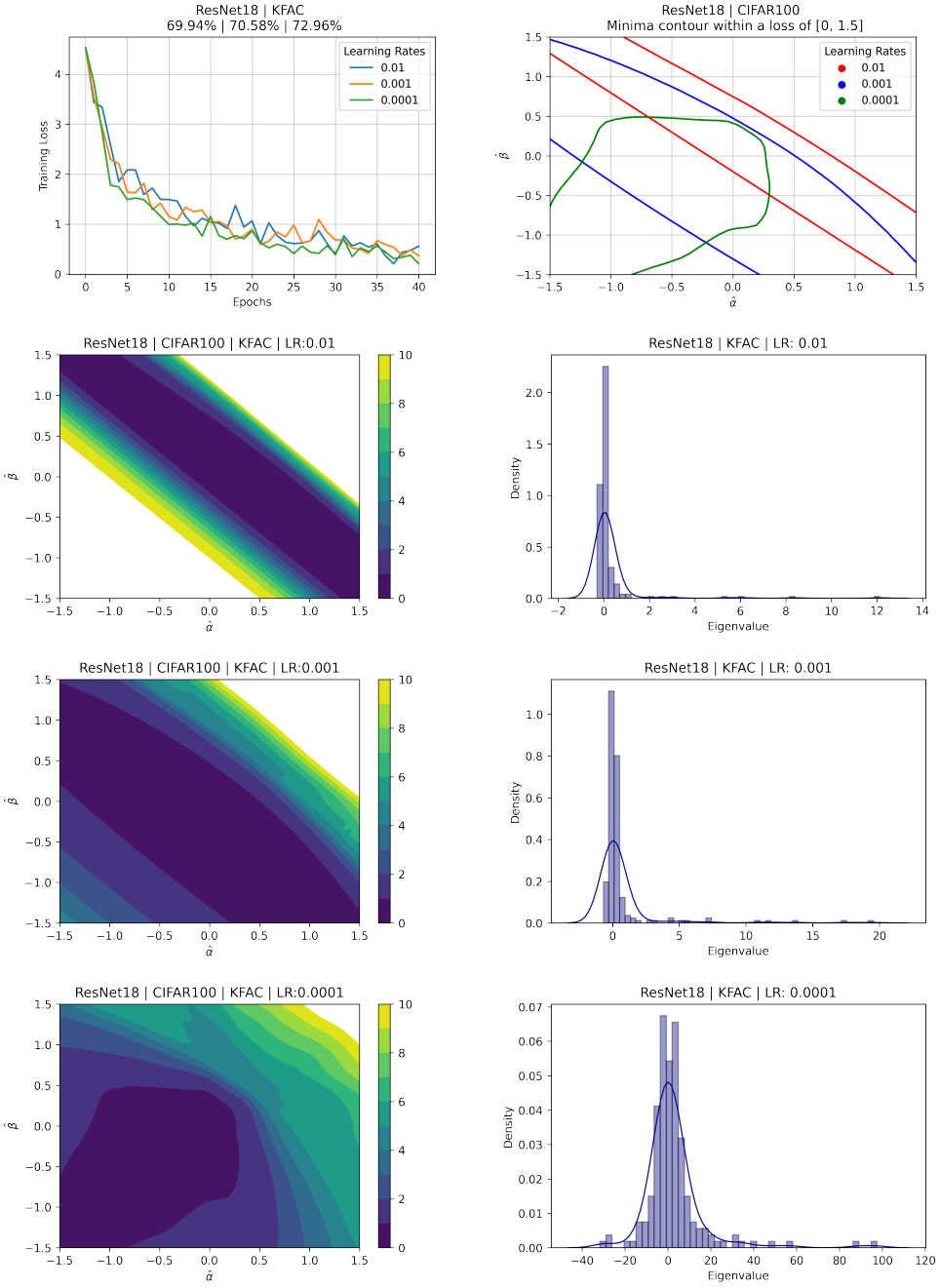


Figure 17: **KFAC** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1.5] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

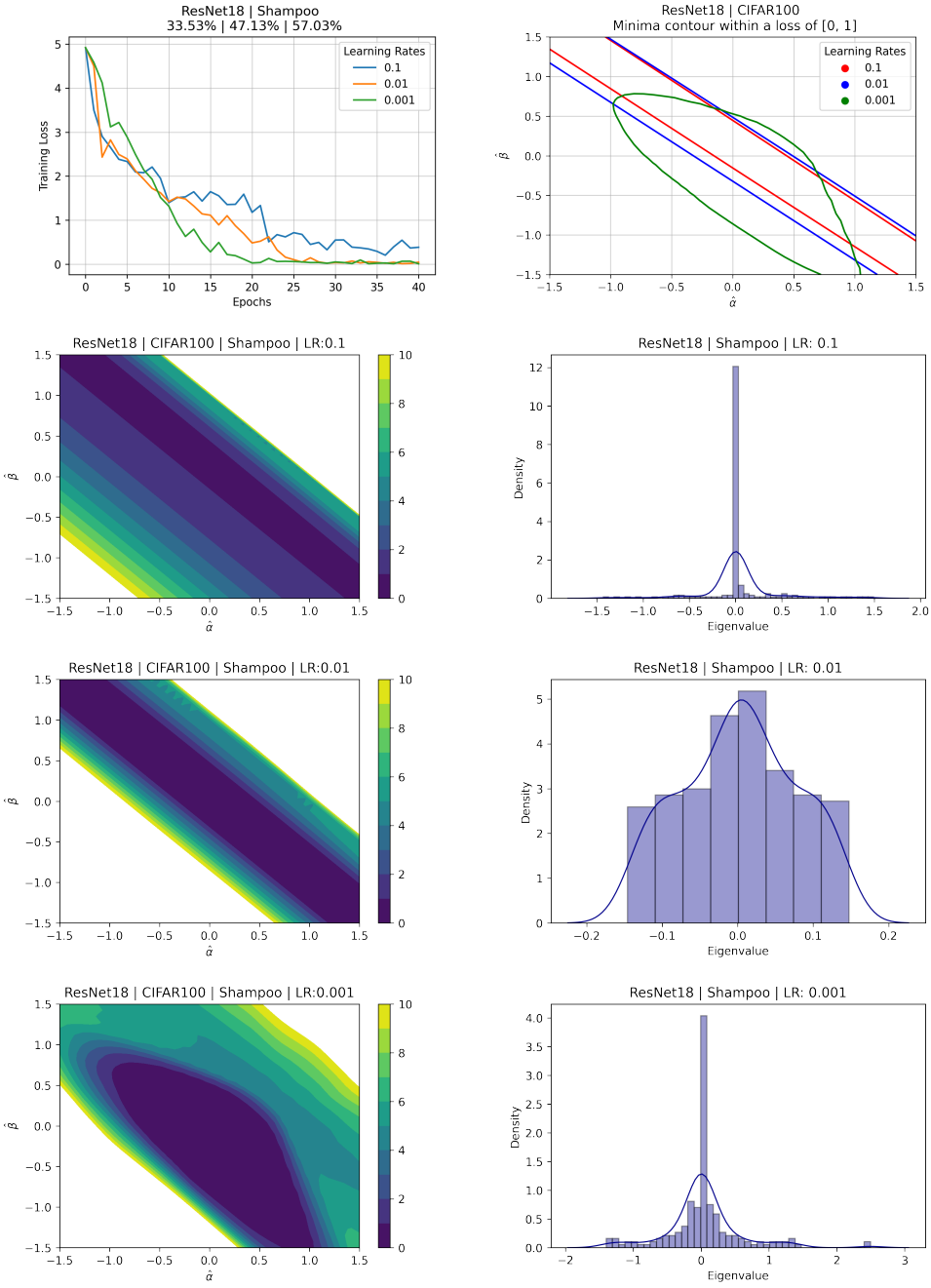


Figure 18: **Shampoo** (**Top-Left**) Training loss convergence plot over 40 training epochs. **(Top-Right)** Contour plot bounding the loss region within the range [0, 1] over different learning rates. **(Left-Column)** Contour loss-surface plots over region surrounding the point of convergence. **(Right-Column)** Spectral density of 200 sampled eigenvalues within the Hessian of the converged model at each learning rate.

	Test Accuracy (%)			
Optimiser	ResNet18	EfficientNet B0	InceptionV3	VGG11
SGD	59.20	57.91	58.54	59.71
Adam	69.12	56.56	64.39	58.52
RAdam	71.68	59.29	68.97	68.87
Yogi	71.05	62.64	69.12	68.19
AMSGrad	68.98	61.19	66.84	66.92
Lookahead	70.22	60.22	67.31	64.79
Ranger	**73.02**	61.68	69.86	68.44
KFAC	72.96	**64.23**	**70.44**	**69.81**

Figure 19: Test accuracies after 40 epochs of training on CIFAR100 for each of the image classification architectures explored in this project, with each architecture possessing the following number of parameters, **ResNet18**: 11.7M, **EfficientNet B0**: 9.2M, **InceptionV3**: 27.2M, **VGG11**: 132.9M

further consideration within this paper, as they are outperformed in all aspects of the optimisation process by the other algorithms and prove intractable for the training of larger models.

6.1.2 Other architectures

The other network architectures displayed in figure 19 suggest a similar pattern, with acceleration algorithms obtaining a superior performance against the baselines. Well performing algorithms appear to generalise well across different architectures with KFAC and Ranger performing the best. The disparity across the performance of each model arises due the architectures themselves, where in the case of EfficientNet the architecture is smaller than ResNet and is not capable of modelling the dataset as closely leading to a poorer accuracy. Consequently, on the other side, VGG11 is much larger than ResNet capable of modelling complex data, however it requires a greater number of epochs to effectively adjust its parameters meaning that at 40 epochs it achieves a lower test accuracy than ResNet. This also illustrate the remarkable ability of the skip-connections within ResNets which provide a complexity reduction mechanism making the network architecture easier and quicker to train than others.

6.2 Natural Language Processing

The second part of the evaluation sees each algorithm applied to the pre-training stage of the RoBERTa [43] model which involves it being trained on a language modelling task, in this case using English. The dataset used is the WikiText-103 dataset, proving sufficiently long to only require a single epoch to produce a trained model [52]. The evaluation metric of a language model is its perplexity which tells how effective the model is at predicting the next words. This section will omit the inclusion of the SVRG and Shampoo algorithms due to their limitations illustrated in the previous section.

The performance increases of acceleration algorithms prove even more notable in NLP applications, as seen in figure 21, each algorithm outperforms the base SGD and Adam by at least 10 perplexity, a considerable margin of improvement. The performance of **SGD** is illustrated in figure 20 where it appears to converge towards a relatively flat region of the loss surface. However, the minimum exists at a higher loss compared to the others, with the surrounding region possessing a loss magnitude of around 6 while

other algorithms obtain a loss of around 3. As such, the flatness of the minimum proves irrelevant in this case as the increased loss leads to the worst test perplexity, with SGD obtaining 87.81, proving more than double that of Adam.

Similarly to SGD, **Adam** performs worse than the acceleration algorithms, with the region surrounding its convergence point at a higher loss than others. This is further illustrated by the convergence plot in figure 20 where Adam displays poor convergence ultimately ending at a loss of around 6 resulting in a test perplexity of 40.93 ranking lower than other acceleration algorithms.

Each of the adaptive algorithms **RAdam**, **Yogi** and **AMSGrad** improve upon Adam with their test perplexity. Of these, Yogi performs the worst, with a perplexity of 33.11, its mechanism which aims at reducing the rapid changes in gradient directions proves of less use than the adaptive mechanisms of RAdam and AMSGrad which obtain a perplexity of 24.96 and 22.65 respectively. This disparity is illustrated in figure 20 where Yogi converges towards a higher training loss than the others which is also reflected in its contour plot, similarly to SGD this displays a minimum at a higher loss than others. Both RAdam and AMSGrad converge towards more optimal minima, existing at a lower loss with the wider minimum achieved by AMSGrad reflected in its improved test perplexity.

The variance reduction methods **Lookahead** and **Ranger** illustrate good loss convergence with improvements upon the test perplexity of baseline algorithms obtaining 25.79 and 22.50 respectively. Helped by the utilisation of RAdam, Ranger outperforms Lookahead as indicated in figure 20 by the region of the loss surface surrounding the minimum which proves flatter for Ranger. Regardless of this, Lookahead still does well to provide a significant improvement on Adam with its exploration mechanism.

Finally, **KFAC** appears to once again be the optimal algorithm, obtaining the best test perplexity of 19.74 and displaying a wider minimum than others in figure 20. These results prove similar to those for the image classification task above, suggesting that well performing algorithms can themselves generalise across different ML tasks. All algorithms in the NLP case found the optimal learning rates being around 100 times smaller in magnitude than those used for the image classification task. This is largely due to the much larger dataset which contains a greater number of data points, as this initially results in significant variance for individual data batches which can lead to an erratic descent path for larger learning rates.

Each full epoch training run of the RoBERTa model took around 6 hours, with all algorithms displaying a very consistent run-time, this proved sufficient for good model performance as illustrated by figure 21. Due to the increased size of the dataset, the execution of training required an increased amount of memory with each run utilising under 30GB of RAM all algorithms proved consistent in their memory usage. The repeated success of KFAC illustrates its ability to generalise not only across different network architecture but also different datasets and ML tasks. The paradigm of "sharp vs flat" minima is well illustrated by the plots in figure 20 where best performing algorithms obtain minima with reduced sharpness. The algorithms which perform the worst, such as SGD and Adam, find regions of the loss surface which would be considered flat however, these exist at higher loss values than the others and hence still lead to poor performance.

7 Conclusion

As the above sections have illustrated, acceleration algorithms can provide significant improvements to the model performance over baselines such as SGD, with a notable difference after only 40 epochs. With most models typically training for around 200

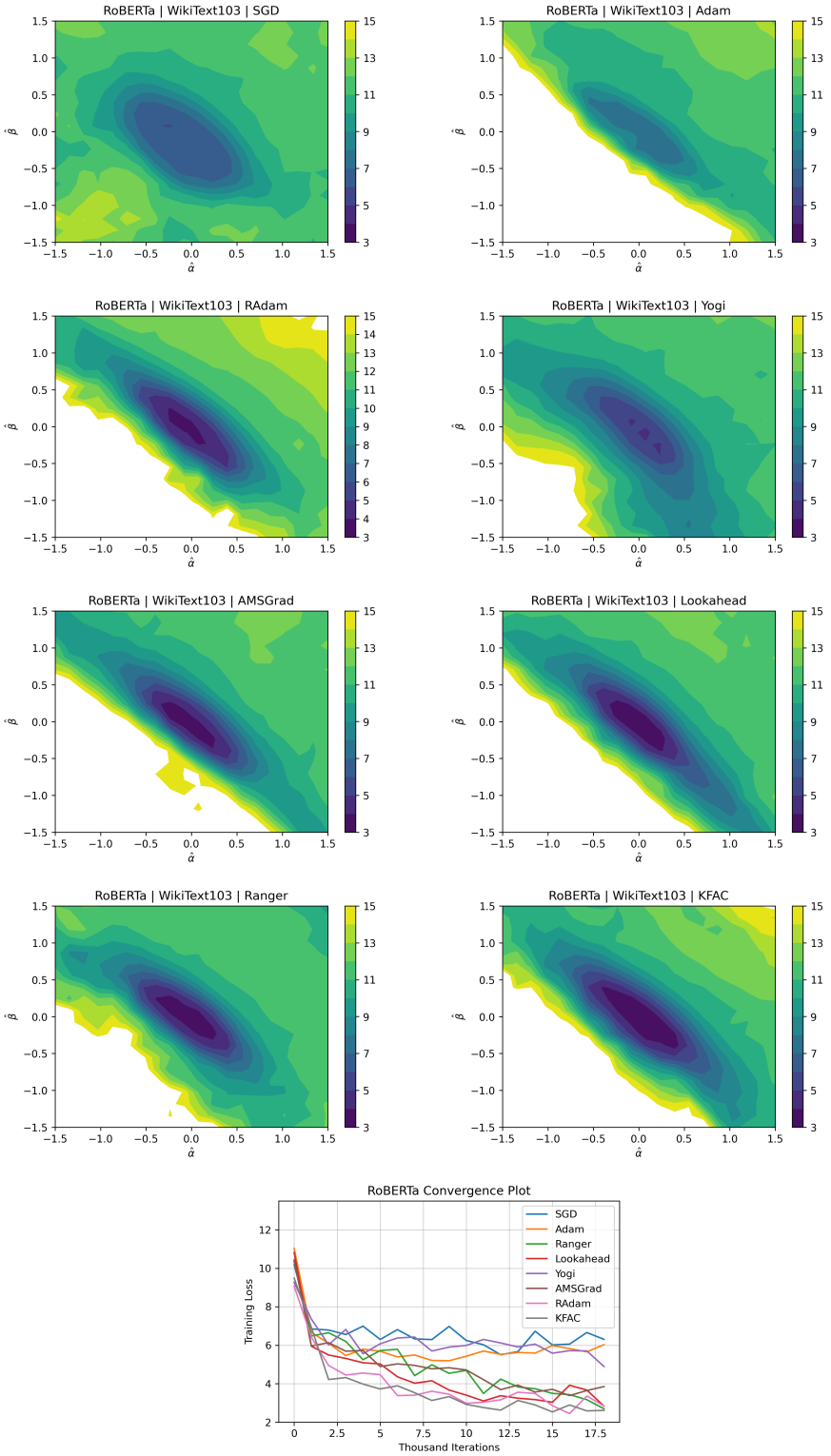


Figure 20: (**Top**) Loss-Contour plots for the convergent minima of each algorithm. (**Bottom**) The training loss convergence plot of each algorithm, evaluated on the test set of the WikiText103 dataset.

Optimiser	Test Perplexity
SGD	87.81
Adam	40.93
RAdam	25.96
Yogi	33.11
AMSGrad	22.65
Lookahead	25.79
Ranger	22.50
KFAC	19.74

Figure 21: Final test perplexity of each algorithm on the WikiText103 dataset.

epochs, this indicates that GD algorithms exert a significant influence on the performance of the model in these early stages of training. The performance gaps witnessed at 40 epochs can prove significant as a result of which any newly proposed algorithms should involve these accelerated methods in their evaluation of performance.

In designing new acceleration algorithms, care should be taken to consider computational and memory budgets as in cases such as SVRG and Shampoo, while feasible in theory they can prove inefficient when implemented into modern frameworks. In this case SVRG saw its memory requirements grow rapidly to facilitate the functionality of its inner loop while Shampoo saw performance issues owing to its implementation methods. On the contrary, such complex features can find efficient implementations as illustrated by Lookahead and KFAC which saw significant improvements utilising and inner loop and Natural Gradient information respectively and did so in an efficient manner with no major overhead. These well-performing algorithms proved to generalise well across different tasks and loss functions, performing well in both image classification and NLP.

Widely discussed in optimisation literature, the paradigm of "Flat vs Sharp" minima proves more complex than expected. While the general trend across different algorithms found that wider minima with a reduced number of large Hessian eigenvalues appear to generalise better, several algorithms reached regions containing eigenvalues of around 100 and still managed to generalise well. While on the contrary, Shampoo converged towards relatively flat regions of the loss surface when its generalisation performance proved far worse than that of other algorithms. In general, there appeared to be little correlation between the properties of the minima and model performance within regions where the eigenvalues were less than 100. Model performance did suffer as the eigenvalues grew to values larger than this, suggesting that there exists a threshold within which moderately sharp curvature along individual dimensions does not prove detrimental. Meanwhile outside of this region, extreme sharpness can lead to a loss of performance.

There exists a disparity between the performance of different network architectures at the 40 epoch mark. This stems from the smaller models with less parameters learning the data distribution quicker, however, they do so with a lower capacity for fitting complex data as illustrated by the lower test accuracies of the EfficientNet architecture in figure 19. Larger networks are capable of more closely fitting such distributions however they consequently require a greater number of iterations as at 40 epochs both Inception and VGG11 achieve a lower test accuracy than the smaller ResNet18. To expand the

analysis performed in this project, given additional time and resources it would be desirable to explore the performance of these algorithms all the way up to full convergence at around 200 epochs, monitoring their performance along the training process. Furthermore, their performance could also be investigated when implemented in conjunction with other ML tasks such as Reinforcement Learning which has seen increased use in recent years.

References

[1] LeCun, Yann, et al. *"Gradient-based learning applied to document recognition."* Proceedings of the IEEE 86.11 (1998): 2278-2324..

[2] Deng, Jia, et al. *"Imagenet: A large-scale hierarchical image database."* 2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009.

[3] Robbins, Herbert, and Sutton Monro. *"A stochastic approximation method."* The annals of mathematical statistics (1951): 400-407.

[4] Keskar, Nitish Shirish, et al. *"On large-batch training for deep learning: Generalization gap and sharp minima."* arXiv preprint arXiv:1609.04836 (2016).

[5] Im, Daniel Jiwoong, Michael Tao, and Kristin Branson. *"An empirical analysis of the optimization of deep network loss surfaces."* arXiv preprint arXiv:1612.04010 (2016).

[6] Hawkins, Douglas M. *"The problem of overfitting."* Journal of chemical information and computer sciences 44.1 (2004): 1-12.

[7] Kandel, Ibrahem, and Mauro Castelli. *"The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset."* ICT Express (2020).

[8] Süli, Endre, and David F. Mayers. *An introduction to numerical analysis.* Cambridge university press, 2003.

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

[10] Soudry, Daniel, and Yair Carmon. *"No bad local minima: Data independent training error guarantees for multilayer neural networks."* arXiv preprint arXiv:1605.08361 (2016).

[11] Lu, Haihao, and Kenji Kawaguchi. *"Depth creates no bad local minima."* arXiv preprint arXiv:1702.08580 (2017).

[12] Kawaguchi, Kenji, and Leslie Kaelbling. *"Elimination of all bad local minima in deep learning."* International Conference on Artificial Intelligence and Statistics. 2020.

[13] Swirszcz, Grzegorz, Wojciech Marian Czarnecki, and Razvan Pascanu. *"Local minima in training of neural networks."* arXiv preprint arXiv:1611.06310 (2016).

[14] Im, Daniel Jiwoong, Michael Tao, and Kristin Branson. *"An empirical analysis of the optimization of deep network loss surfaces."* arXiv preprint arXiv:1612.04010 (2016).

[15] Mehta, Dhagash, et al. *"The loss surface of deep linear networks viewed through the algebraic geometry lens."* arXiv preprint arXiv:1810.07716 (2018).

[16] Krizhevsky, Alex, and Geoffrey Hinton. *"Learning multiple layers of features from tiny images."* (2009): 7.

[17] Tan, Mingxing, and Quoc V. Le. *"Efficientnet: Rethinking model scaling for convolutional neural networks."* arXiv preprint arXiv:1905.11946 (2019).

[18] He, Kaiming, et al. *"Deep residual learning for image recognition."* Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[19] Szegedy, Christian, et al. *"Going deeper with convolutions."* Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[20] Szegedy, Christian, et al. *"Rethinking the inception architecture for computer vision."* Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[21] Simonyan, Karen, and Andrew Zisserman. *"Very deep convolutional networks for large-scale image recognition."* arXiv preprint arXiv:1409.1556 (2014).

[22] Liu, Wei, et al. *"Ssd: Single shot multibox detector."* European conference on computer vision. Springer, Cham, 2016.

[23] Merity, Stephen, et al. *"Pointer sentinel mixture models."* arXiv preprint arXiv:1609.07843 (2016).

[24] Kingma, Diederik P., and Jimmy Ba. *"Adam: A method for stochastic optimization."* arXiv preprint arXiv:1412.6980 (2014).

[25] Liu, Liyuan, et al. *"On the variance of the adaptive learning rate and beyond."* arXiv preprint arXiv:1908.03265 (2019).

[26] Reddi, Sashank J., Satyen Kale, and Sanjiv Kumar. *"On the convergence of adam and beyond."* arXiv preprint arXiv:1904.09237 (2019).

[27] Zaheer, Manzil, et al. *"Adaptive methods for nonconvex optimization."* Advances in neural information processing systems. 2018.

[28] Johnson, Rie, and Tong Zhang. *"Accelerating stochastic gradient descent using predictive variance reduction."* Advances in neural information processing systems. 2013.

[29] Ioffe, Sergey, and Christian Szegedy. *"Batch normalization: Accelerating deep network training by reducing internal covariate shift."* arXiv preprint arXiv:1502.03167 (2015).

[30] Srivastava, Nitish, et al. *"Dropout: a simple way to prevent neural networks from overfitting."* The journal of machine learning research 15.1 (2014): 1929-1958.

[31] Defazio, Aaron, and Léon Bottou. *"On the ineffectiveness of variance reduced optimization for deep learning."* Advances in Neural Information Processing Systems. 2019.

[32] Zhang, Michael, et al. *"Lookahead optimizer: k steps forward, 1 step back."* Advances in Neural Information Processing Systems. 2019.

[33] Yong, Hongwei, et al. "Gradient Centralization: A New Optimization Technique for Deep Neural Networks." arXiv preprint arXiv:2004.01461 (2020).

[34] Santurkar, Shibani, et al. *"How does batch normalization help optimization?."* Advances in Neural Information Processing Systems. 2018.

[35] Ly, Alexander, et al. *"A tutorial on Fisher information."* Journal of Mathematical Psychology 80 (2017): 40-55.

[36] Martens, James. *"New insights and perspectives on the natural gradient method."* arXiv preprint arXiv:1412.1193 (2014).

[37] Martens, James, and Roger Grosse. *"Optimizing neural networks with kronecker-factored approximate curvature."* International conference on machine learning. 2015.

[38] Gupta, Vineet, Tomer Koren, and Yoram Singer. *"Shampoo: Preconditioned stochastic tensor optimization."* arXiv preprint arXiv:1802.09568 (2018).

[39] Brown, Tom B., et al. *"Language models are few-shot learners."* arXiv preprint arXiv:2005.14165 (2020).

[40] Li, Hao, et al. *"Visualizing the loss landscape of neural nets."* Advances in Neural Information Processing Systems. 2018.

[41] Hao, Yaru, et al. *"Visualizing and understanding the effectiveness of BERT."* arXiv preprint arXiv:1908.05620 (2019).

[42] Jastrzebski, Stanislaw, et al. *"On the relation between the sharpest directions of DNN loss and the SGD step length."* arXiv preprint arXiv:1807.05031 (2018).

[43] Liu, Yinhan, et al. *"Roberta: A robustly optimized bert pretraining approach."*arXiv preprint arXiv:1907.11692 (2019).

[44] Devlin, Jacob, et al. *"Bert: Pre-training of deep bidirectional transformers for language understanding."* arXiv preprint arXiv:1810.04805 (2018).

[45] Goodfellow, Ian, et al. *"Deep learning. Vol. 1."* Cambridge: MIT press, 2016.

[46] Murty, Katta G., and Santosh N. Kabadi. *Some NP-complete problems in quadratic and nonlinear programming.* 1985.

[47] Hecht-Nielsen, Robert. *"Theory of the backpropagation neural network."* Neural networks for perception. Academic Press, 1992. 65-93.

[48] Shamir, Ohad. *"Without-replacement sampling for stochastic gradient methods."* Advances in neural information processing systems. 2016.

[49] Recht, Benjamin, and Christopher Ré. *"Beneath the valley of the noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences."* arXiv preprint arXiv:1202.4184 (2012).

[50] Bottou, Léon, and Olivier Bousquet. *"The tradeoffs of large scale learning."* Advances in neural information processing systems. 2008.

[51] Bottou, Léon, Frank E. Curtis, and Jorge Nocedal. *"Optimization methods for large-scale machine learning."* Siam Review 60.2 (2018): 223-311.

[52] Komatsuzaki, Aran. *"One Epoch Is All You Need."* arXiv preprint arXiv:1906.06669 (2019).

[53] Dinh, Laurent, et al. *"Sharp minima can generalize for deep nets."* arXiv preprint arXiv:1703.04933 (2017).

[54] Kawaguchi, Kenji, Leslie Pack Kaelbling, and Yoshua Bengio. *"Generalization in deep learning."* arXiv preprint arXiv:1710.05468 (2017).

[55] Yao, Zhewei, et al. *"PyHessian: Neural networks through the lens of the Hessian."* arXiv preprint arXiv:1912.07145 (2019).

[56] Ghorbani, Behrooz, Shankar Krishnan, and Ying Xiao. *"An investigation into neural net optimization via hessian eigenvalue density."* arXiv preprint arXiv:1901.10159 (2019).

[57] Golub, Gene H., and Gérard Meurant. *Matrices, moments and quadrature with applications*. Vol. 30. Princeton University Press, 2009.

[58] Paszke, Adam, et al. *"Pytorch: An imperative style, high-performance deep learning library."* Advances in neural information processing systems. 2019.

[59] Vaswani, Ashish, et al. *"Attention is all you need."* Advances in neural information processing systems. 2017.

A First-Order Convexity

The first-order condition can be derived from the general convexity condition.

$$f(\alpha y + (1-\alpha)x) \leq \alpha f(y) + (1-\alpha)f(x)$$

Proceeding, the goal is to rearrange this to isolate $f(y)$, $f(x)$ and any terms involving α.

$$f(x + \alpha(y-x)) \leq f(x) + \alpha(f(y) - f(x))$$
$$f(x + \alpha(y-x)) - f(x) \leq \alpha(f(y) - f(x))$$
$$\frac{f(x + \alpha(y-x)) - f(x)}{\alpha} \leq f(y) - f(x)$$
$$f(y) \geq f(x) + \frac{f(x + \alpha(y-x)) - f(x)}{\alpha}$$

The expression is now beginning to more closely resemble the first-order condition, all that remains is to deal with the term on the far right side.

$$g(\alpha) = f(x + \alpha(y-x))$$

Putting this back in.

$$f(y) \geq f(x) + \frac{g(\alpha) - g(0)}{\alpha}$$

Taking the limit $\lim_{\alpha \to 0}$ would grant the derivative $g'(0)$, however instead define a more general form from the first order Taylor expansion of g.

$$g'(t) = (\nabla f(x + \alpha(y-x)))^T (y-x)$$

For the case of $t = 0$,

$$f(y) \geq f(x) + g'(0) = f(x) + (\nabla f(x))^T (y-x)$$

Finally, the first-order condition is obtained.

$$f(y) \geq f(x) + (\nabla f(x))^T (y-x)$$

B Taylor Expansion of KL Divergence

Derivation of the second order Taylor expansion of the KL divergence for two distributions for functions $f(\theta)$ and $f(\theta + d)$ with data samples x.

$$KL\big[p_{\theta,x} || p_{\theta+d,x}\big] = KL\big[p_{\theta,x} || p_{\theta,x}\big] + (\nabla KL\big[p_{\theta,x} || p_{\theta,x}\big])^T d + \frac{1}{2} d^T F d$$

$$= KL\big[p_{\theta,x} || p_{\theta,x}\big] + \mathbb{E}\big[\nabla \log(p_{\theta,x})\big]^T d + \frac{1}{2} d^T F d$$

Firstly, note that the KL divergence of the same distribution is zero. Furthermore, the second term on the RHS can be broken down as follows.

$$\mathbb{E}\big[\nabla \log(p_{\theta,x})\big] = \int \nabla \log(p_{\theta,x}) p_{\theta,x} \, dx$$

$$= \int \frac{\nabla p_{\theta,x}}{p_{\theta,x}} p_{\theta,x} \, dx$$

$$= \int \nabla p_{\theta,x} \, dx = \nabla \int p_{\theta,x} \, dx$$
$$= \nabla 1 = 0$$

Hence, the second term on the RHS also becomes zero, resulting in the second order Taylor expansion of the KL divergence being,

$$KL\big[p_{\theta,x} || p_{\theta+d,x}\big] = \frac{1}{2} d^T F d$$