Second order optimization of neural networks with Kronecker factored approximate curvature

Karl Byberg Ulbæk - s183931

December 11, 2022

1 Abstract

Second order methods can optimize neural networks in a fraction of the iterations needed by traditional first order methods. However, the curvature matrix involved herein scales with the number of parameters squared and must be efficiently and accurately approximated to make second order optimization a viable option. The state of the art approach for doing so is Kronecker factored approximations. This study examines the underlying ideas which motivates the use of second order methods for neural networks optimization. Furthermore this study derives the original Kronecker factored approximate curvature (KFAC) theory for fully connected networks.
KFAC is subsequently implemented and compared to SGD with momentum across 3 different machine learning problems. Bayesian optimization for hyperparameters tuning is employed to ensure that the optimization approaches are performing to the best of their abilities and that the human element is eliminated. It is found that KFAC outperforms SGD.
Additionally, through the experimental work it is found that KFAC prefers similar parameters across problems which may potentially cut down on the time spent hyperparameter tuning. Furthermore it is discovered that KFAC has a strong preference for large batch sizes and is in fact unable to learn if the batch size is sufficiently small. This study proposes a solution which enables learning in these cases.

Contents

2 Background

Neural networks have become the state-of-the art modeling approach for both supervised and unsupervised machine learning in the recent decade. Optimization algorithms are the underlying engine which enables the neural networks to learn from data by adapting their parameters. The optimization algorithms attempt to solve the problem of minimizing an objective function which is a measure of the errors made by the network. They iteratively do so by approximating the objective function in a close vicinity of an evaluation point and then update the parameters according to the local approximation. Conventionally the optimization methods have been first order methods which primarily considers the direction of steepest descent as given by the first order derivatives of the objective function. These methods offer extremely efficient per iteration cost, but due to their confined understanding of the objective function are limited to taking small steps.
Second order methods are the natural extension of first order methods. They are an attempt at more sophisticated approximations of the objective functions by the use of curvature information. These methods achieve a better local understanding of the objective function which enables them to make greater updates to the network parameters, although this comes at the cost of significantly increasing computations per iteration. The full curvature information assumes the shape of a n *by* n matrix, where n is the number of parameters in the network. The most obvious and intuitive instance hereof is the matrix of 2. order derivatives, namely the hessian. However in the non convex optimization scene of neural network optimization, the hessian is impractical and rarely used, one reason being that it is indefinite. More widely used alternatives to the Hessian, as a curvature matrix, include the Generalized Gauss-Newton matrix, the Fisher information matrix and the empirical Fisher matrix. Nonetheless these are still of magnitude n *by* n which for anything but the smallest of networks make them computationally infeasible. For second order optimization to be viable the curvature matrix must be approximated to some extent.

2.1 Related work

The simple and most crude approximations are diagonal approximations. As the name suggests only the diagonal elements of the curvature matrix are considered. Examples hereof are RMS-prop and ADAM[1] as they incorporate the square root of the diagonal empirical Fisher. The approximation is highly inaccurate but does enable basic scaling differences between parameters. These methods scale linearly with the number of parameters and are only slightly more computationally demanding than traditional first order methods while still improving optimization performance significantly, thus making them popular choices.
Other crude approximation approaches include low rank approximations as seen in L-BFGS[2]. Here, rather than storing the full inverse of the hessian, only a subset of vectors from the most recent iterations are stored from which the hessian can be represented as a low rank approximation. L-BFGS was not intended for neural network optimization but is nonetheless an example of how to efficiently (not necessarily accurately) store the inverse hessian.
A more sophisticated approach to approximating the curvature matrix is the block-diagonal approximations. Each block corresponds to a subset of parameters which is believed to be closely related and thus deserves to be represented as a full matrix. However relationships between groups are not modeled by this approach and everything but the block diagonal entries of the curvature matrix are zeroed out. Each block may correspond to the weights on the connections going into a given unit or the weights on the connections going out of a unit. The computational cost associated with the block diagonal approach scales with the size of the block matrices. The best example hereof is TONGA[3]. This approach is very inefficient in practice and compares poorly to modern optimization techniques.
Hessian free[4] optimization is an approach which does not directly approximate the curvature matrix but rather approximately minimizes a quadratic model. This is achieved through the use of

preconditioned linear conjugate gradient, by utilizing the fact that the conjugate gradient method does not involve the entirety of the inverse curvature matrix but only selected matrix vector product with the inverse which can be performed rather efficiently.

State of the art with regards to efficiently and accurately approximating the curvature matrix is Kronecker factored approximations. The original work which popularized the idea originates from the 2015 paper[5] by Martens and Grosse. This work will serve as the foundation for this study and will from here on be referenced as the KFAC paper. The central ideas hereof (which will be presented in full in later sections) boils down to representing the curvature matrix as a block diagonal or block tridiagonal where each block itself is an approximate construction that emerges from the Kronecker product between 2 smaller matrices. These two smaller matrices may be estimated and inverted more efficiently than the full block matrix due to the Kronecker identity which implies that the inverse of the Kronecker product equals the Kronecker product of inverses. The original Kronecker factored approach was only derived and implemented for fully connected networks but has subsequently been extended to other architectures such as convolutional neural networks[6] and recurrent neural networks[7], as well as employed in the setting of reinforcement learning[8] and Bayesian deep learning[9].

Correspondingly the KFAC approach has been advanced to methods such as Eigenvalue corrected kronecker factorization, E-KFAC[10]. This method draws on ideas of both KFAC but also the diagonal approximations. Here a diagonal variance is kept track of, but not in terms of parameter coordinates but in a Kronecker factored eigenbasis. This yields a better approximation of the curvature matrix and allows partial updates of the curvature estimate at iteration level.
Other advances of the Kronecker factored approach includes the recent Swift KFAC[11]. This work argues that KFAC is still too computational inefficient for especially optimization of large networks. Therefore they propose further approximations, by utilizing the low rank properties of the Kronecker factored Fisher, to employ a faster low rank inversion technique. Additionally this work extends to ideas related to improving KFAC in the convolutional networks setting. Here they propose two dimensionality reduction schemes to reduce across the receptive fields of feature maps of the convolutional layers.
Additionally, a substantial body of work has gone into addressing concerns related to KFAC. Including concerns regarding the validity of the approximated curvature due to dependency between examples caused by, for instance, batch normalization layers in deep neural networks[12]. Another study accommodates KFACs inherent beneficial scaling with larger batch sizes. One concern that arises with this property are the memory limitations associated with the big batches which the study resolves by proposing a solution which distributes the workload across multiple machines[13]. Correspondingly KFACs preference for large batch sizes gives rise to concerns related to bigger batch sizes having a negative impact on the generation capabilities of the model[14]. However state of the art performance and generalization capabilities are achieved in this[15] study using a KFAC approach.

Second order optimization remains an active topic within the neural network optimization literature and Kronecker factored approximations still constitute the foundation in the curvature estimates used by current state of the art second order optimization methods.

2.2 Purpose of this study

This study will motivate the use of second order methods for neural network optimization through mathematical derivations of the central concepts Fisher information matrix and natural gradient. Furthermore the theory related to Kronecker factored approximate curvature for fully connected networks will be derived in full. Subsequently the theory will be put to use in a *from the ground up implementation* of KFAC. The implementation will be compared to SGD with momentum across 3 problems and Bayesian optimization will be employed for all hyperparamter tuning. Finally it will be answered if this study´s comparison may similarly conclude that KFAC outperforms SGD, as it was found in the original paper.

3 The natural gradient

The theory introduced in this section will serve to motivate the use of second order methods for optimisation, by deriving the Fisher information as the curvature matrix which in turn gives rise to the concept of natural gradient descent. The derivations and perspective will be based on two lectures[16] [17] by professor Fred Hamprecht, but the concepts and ideas as a whole originate from the work of especially Amari[18] and Desjardins[19]. The perspective will be kept on a general level and the theory will apply to any statistical model of some parameters which takes an input and outputs a probability distribution. This model can be expressed as $p(z \mid x, \theta)$ and is a conditional distribution over outputs z given some input x and model parameters θ. For compact notation the implicit form $p_\theta(x)$ will be used during the derivations.

3.1 Neural networks as statistical models

It might not be immediately apparent how theory related to statistical models of the form $p(z \mid x, \theta)$ are of relevance in the context of second order optimization of **neural networks**. To convince the reader hereof, this section will start out by arguing how neural networks may be perceived as such statistical models.
Depending on the problem there are various ways of translating a neural network into a statistical model. For binary classification, this can be achieved by letting the output neuron of the network be probabilistic. Thus, the prediction would be a conditional distribution of some random variable z given some input x. In this sense z would be Bernoulli distributed as $z \mid x \sim \text{Bern}(\sigma(x; \theta))$ where σ is the deterministic output of the network which in turn depends on the parameters and input. Correspondingly this can be extended to multi class classification by the multinomial distribution. A similar consideration can be made with regards to a regression problem. Instead of thinking of z as being Bernoulli distributed it now follows a normal distributing $z \mid x \sim N\left(\sigma(x; \theta), \delta^2\right)$ with a mean σ given by the deterministic prediction of the network as well as some variance δ^2.
In this sense, for an input, the neural network induces a distribution over possible outputs. Normally the network would be optimized according to minimizing the loss. However the notion of minimizing a loss can correspondingly be seen as a problem of finding the maximum likelihood solution.

$$L(\theta) = \prod_{i=n}^{n} p\left(t_i \mid x_i; \theta\right) \tag{1}$$

This is the likelihood of the parameters over some training set, with t being the true label of the sample. It is a product of conditional probabilities of observing the label given the input and parameters. Or for convenience the negative log likelihood.

$$\ell(\theta) = \sum_{i=n}^{n} -\log\left(p\left(t_i \mid x_i; \theta\right)\right) = \sum_{i=n}^{n} \text{loss}\left(t_i, z_i\right) \tag{2}$$

Which turns the parameter finding process into a minimization problem which even more so resembles that of minimizing a loss.

3.2 A proximal gradient descent perspective

The concept of natural gradient descent will be motivated from a proximal gradient descent view with the idea being to replace the proximal operator with something that better accounts for the underlying structure of the model. Traditional steepest descent can be formulated as follows.

$$\theta_{t+1} = \theta_t - \mu \left(\frac{\partial \ell(\theta, x)}{\partial \theta} \right) \tag{3}$$

Where $\frac{\partial \ell(\theta,x)}{\partial \theta}$ is the empirical loss as a function of parameters θ and training data x differentiated with respect to the parameters θ i.e. the gradient ∇_θ. The two notation styles refer to the same quantity and will be used interchangeably. The parameters and gradients are assumed to be a column vectors.
The concept of steepest descent can easily be extended to proximal gradient descent. Proximal gradient methods play a great role in non convex optimization theory. They are useful in particular when the objective function are made of a multitude of functions, some of which are non-differentiable. For optimization problems such as these it is often desired to not take too large steps since smaller steps will improve the convergence of the optimizer. On this note finding θ_{t+1} can be formulated as the following optimization problem.

$$\theta_{t+1} = \underset{\theta^*}{\operatorname{argmin}} \left(\frac{\partial \ell(\theta, x)}{\partial \theta} \right) \theta^* + \frac{1}{2\mu} \left(\theta^* - \theta_t\right)^\top \left(\theta^* - \theta_t\right) \tag{4}$$

The first term ensures that θ^* lies in the direction of steepest descent, whereas the last term $\left(\theta^* - \theta_t\right)^\top \left(\theta^* - \theta_t\right)$ is the proximal term and imposes a quadratic loss and thus enforces the notion of constraining the step sizes. The last term is itself convex and thereby does not further complicate the problem.
This new minimization problem can be solved by differentiating with respect to the new parameter θ^* and setting the expression equal to zero. Solving with respect to θ^* yields an update rule identical to that of steepest descent.

$$\theta_{t+1} = \theta_t - \mu \left(\frac{\partial \ell(\theta, x)}{\partial \theta} \right) \tag{5}$$

However the proximity term $\left(\theta^* - \theta_t\right)^\top \left(\theta^* - \theta_t\right)$ which seeks to constrain the magnitude of the updates does not know anything about the underlying structure of the model and is practically expressed in terms of the squared euclidean distance. This implies a series of problems which entrails that small changes to the parameters (from the perspective of euclidean distance) may result in large changes to the output. For instance the rate at which parameters change may be different, as some parameters may be given in centimeters, for example, while others are given in meters. Additional parameters may co-vary in the sense that walking in the direction of the gradient of one parameter may change the gradient of some other parameters.
The critical implications of the difference in scale is best emphasised through illustration.

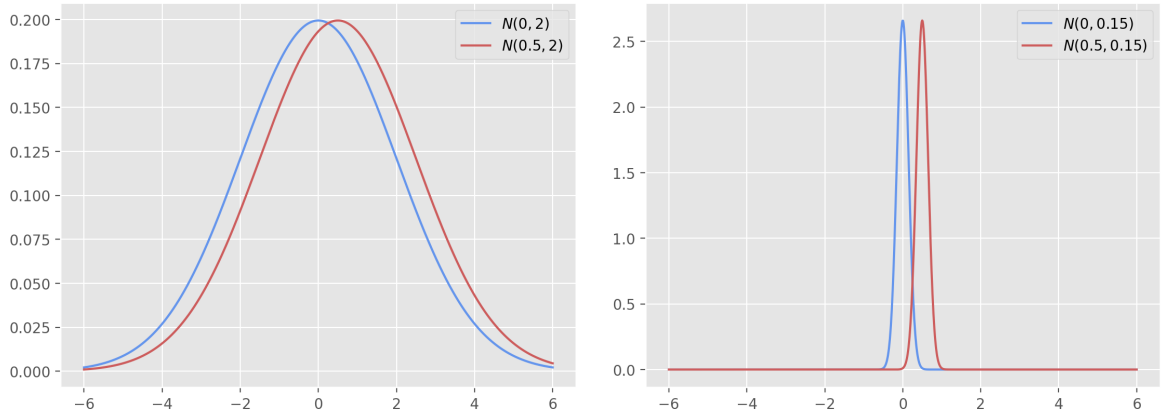

Figure 1: The left plot depicts 2 normal distributions with slightly different mean but the same standard deviation of 2. Similarly the right plot depicts 2 normal distributions with the same difference in means as the left plot but with a relatively smaller, yet identical, standard deviation of 0.15.

The change in parameter space as measured by the euclidean distance between the two pair of normal distributions are the same, namely a difference in mean of 0.5. This change in mean barely alters how we perceive the 2 distributions with large standard deviation (left). However the same change in mean has a crucial effect on the distribution when the standard deviation is smaller (right). The obvious drastic affect on the output distribution is not captured by the euclidean distance, thus demonstrating why it is a poor metric in this context.

3.3 The Fisher information matrix

Instead of penalizing based on the euclidean difference in parameter space, it makes more sense to penalize based on the change in output distribution of the model. The goal is to replace the proximal operator, which as of now is expressed as the squared euclidean distance between the parameters, with a metric which measures the difference between the output distributions associated with these parameters.
Inspired by information geometry varying the parameters θ can be seen as moving along a functional manifold. The functional manifold of a parametric statistical model is a Riemannian manifold[20]. One functional manifold corresponds to all models or networks with the same architecture but with different parameters. The manifold has an associated metric tensor[21]. In the field of differential geometry the metric tensor is a type of function or inner-product which takes as input a pair of tangent vectors at a point on a surface and produces a real scalar in a way similar to the dot product of vectors in euclidean space. Metric tensors are used to define the length of and angle between tangent vectors and allows one to compute the length of curves on the manifold. One definitional property of the Riemannian manifold is that that the inner product varies smoothly as you move from one point on the manifold to another. This metric tensor of the functional manifold associated with a model will come in the form of the Fisher information matrix, often referred to as simply "the Fisher". This results emerges by considering the difference in output distribution as measured by the Kullback–Leibler (KL) divergence. The KL divergence is a dissimilarity measure between two probability distribution and is given as follows.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \int p_\theta(x) \log \frac{p_\theta(x)}{p_{\theta+\phi}(x)} \mathrm{d}x \tag{6}$$

Where $p_\theta(x)$ is the resulting output density of the model with parameters θ when evaluating the input x. Similarly $p_{\theta+\phi}(x)$ is the output density of the same model but with a slight change in its parameters. In general the KL divergence is not symmetric but in the limit of ϕ approaching the zero vector it will be a *metric*.
First, the fraction in the logarithm is rewritten in terms of subtraction.

$$KL\left(p_\theta \| p_{\theta+\boldsymbol{\phi}}\right) = \int p_\theta(x) \log\left(p_\theta(x)\right) dx - \int p_\theta(x) \log\left(p_{\theta+\boldsymbol{\phi}}(x)\right) \mathrm{d}x \tag{7}$$

Subsequently 2 approximations will be applied, the first of which is a 1st order Taylor expansion of the density.

$$p_{\theta+\phi}(x) \approx p_\theta(x) + \left(\nabla_\theta p_\theta(x)\right)\ \phi \tag{8}$$

Additionally, from the definition of expected value the term $\int p_\theta(x) \log\left(p_\theta(x)\right) dx$ can be rewritten as such. Inserting the approximation and rewriting yields.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \mathrm{E}_x\left[\log\left(p_\theta(x)\right)\right] - \int p_\theta(x) \log\left(p_\theta(x) + \left(\nabla_\theta p_\theta(x)\right)\ \phi\right) \mathrm{d}x \tag{9}$$

The log term, $\log\left(p_\theta(x) + \left(\nabla_\theta p_\theta(x)\right)\ \phi\right)$ can be refactored to $\log\left(p_\theta(x)\left(1 + \frac{1}{p_\theta(x)}\left(\nabla_\theta p_\theta(x)\right)\phi\right)\right.$ which gives.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \mathrm{E}_x\left[\log\left(p_\theta(x)\right)\right] - \int p_\theta(x) \log\left(p_\theta(x)\left(1 + \frac{1}{p_\theta(x)}\ \left(\nabla_\theta p_\theta(x)\right)\ \phi\right)\right. \mathrm{d}x \tag{10}$$

The log term is expanded.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \mathrm{E}_x\left[\log\left(p_\theta(x)\right)\right] - \int p_\theta(x) \log\left(p_\theta(x)\right) dx - \int p_\theta(x) \log\left(1 + \frac{1}{p_\theta(x)}\ \left(\nabla_\theta p_\theta(x)\right)\ \phi\right)\right. \mathrm{d}x \tag{11}$$

Splitting up the log term yet again results in something which can be rewritten as an expectation.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \mathrm{E}_x\left[\log\left(p_\theta(x)\right)\right] - \mathrm{E}_x\left[\log\left(p_\theta(x)\right)\right] - \int p_\theta(x) \log\left(1 + \frac{\left(\nabla_\theta p_\theta(x)\right)\ \phi}{p_\theta(x)}\right) \mathrm{d}x \tag{12}$$

The expectations cancel out.

$$KL\left(p_\theta \| p_{\theta+\boldsymbol{\phi}}\right) = -\int p_\theta(x) \log\left(1 + \frac{\left(\nabla_\theta p_\theta(x)\right)\ \phi}{p_\theta(x)}\right) \mathrm{d}x \tag{13}$$

The contents of the log is now 1 plus a fraction. The faction is a very small number since ϕ was assumed to be in the limit of 0. Now the 2. approximation will be employed by utilizing the fact that the log of 1 plus something small can be approximated by.

$$\log(1+\epsilon) = \epsilon - \frac{\epsilon^2}{2} + \frac{\epsilon^3}{3} \dots \tag{14}$$

Rewriting in terms of this approximation up to and including the squared term gives.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = -\int p_\theta(x) \left(\frac{\left(\nabla_\theta p_\theta(x)\right)\ \phi}{p_\theta(x)} - \frac{\phi\left(\nabla_\theta p_\theta(x)\right)\ \left(\nabla_\theta p_\theta(x)\right)^\top\ \phi^\top}{2\, p_\theta^2(x)}\right) dx \tag{15}$$

The integration is with respect to x but the derivatives are with respect to θ. Therefore subject to some regularity conditions the gradient can be pulled out of the integral of the first term.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = -\nabla_\theta \int p_\theta(x) dx\, \phi + \int p_\theta(x) \frac{\phi\left(\nabla_\theta p_\theta(x)\right)\ \left(\nabla_\theta p_\theta(x)\right)^\top\ \phi^\top}{2\, p_\theta^2(x)} dx \tag{16}$$

Consider now the first term. $\int p_\theta(x)dx$ is the integral over a probability density and thus equals 1. Meaning that ∇_θ becomes the derivative of a constant i.e. zero, resulting in the whole first term zeroing out. Furthermore the fraction of the second term is split into two fractions and the ϕ vectors are pulled out of the integral.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \frac{1}{2}\phi \int p_\theta(x) \frac{\left(\nabla_\theta p_\theta(x)\right)\left(\nabla_\theta p_\theta(x)\right)^\top}{p_\theta(x)} dx\, \phi^\top \tag{17}$$

The following identity with respect to the gradients is used $\frac{1}{p_\theta(x)} \nabla_\theta p_\theta(x) = \nabla_\theta \log\left(p_\theta(x)\right)$ as it can be realized that the left-hand side is what is written as $\frac{\left(\nabla_\theta p_\theta(x)\right)}{p_\theta(x)}$ in the expression and can be substituted for the right-hand side.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \frac{1}{2}\phi \int p_\theta(x) \left(\nabla_\theta \log\left(p_\theta(x)\right)\right)\left(\nabla_\theta \log\left(p_\theta(x)\right)\right)^\top dx\, \phi^\top \tag{18}$$

Yet again this can be expressed in terms of an expectation yielding the final expression.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \frac{1}{2}\phi\, \mathrm{E}_x\left[\nabla_\theta \log\left(p_\theta(x)\right) \nabla_\theta \log\left(p_\theta(x)\right)^\top\right] \phi^\top \tag{19}$$

This expectation is exactly the Fisher information matrix denoted F_θ.

$$KL\left(p_\theta \| p_{\theta+\phi}\right) = \frac{1}{2}\phi F_\theta\, \phi^\top \tag{20}$$

The Fisher emerges as the approximated KL divergence between the two distributions $p_\theta(x)$ and $p_{\theta+\phi}(x)$ and is a measure of how much an infinitesimal change ϕ to the parameters θ will affect the distribution. Traditionally the Fisher is considered the expected value of the outer product of the score. Score being defined as the derivative of the log likelihood of the data under the given parameters.
The Fisher, being a metric which better accounts for the underlying structure of the model and how changes to the model affects the output, can be used as the proximal operator in the proximal gradient expression.

$$\theta_{t+1} = \theta_t + \underset{\phi}{\operatorname{argmin}} \left(\frac{\partial \ell(\theta, x)}{\partial \theta}\right) \phi + \frac{1}{2\,\mu}\, KL\left(p_\theta \| p_{\theta+\phi}\right) \tag{21}$$

$$\theta_{t+1} = \theta_t + \underset{\phi}{\operatorname{argmin}} \left(\frac{\partial \ell(\theta, x)}{\partial \theta}\right) \phi + \frac{1}{2\,\mu}\, \frac{1}{2}\phi F_\theta\, \phi^\top \tag{22}$$

Notice, since the minimization is only over ϕ (which is a difference/change), θ_t now has to be added in front of the minimization expression. The minimization is solved by differentiating with respect to ϕ and setting equal to 0. Solving for ϕ yields.

$$\phi = -2\,\mu\, F_\theta^{-1} \frac{\partial \ell(\theta, x)}{\partial \theta} \tag{23}$$

Disregarding the sign, the multiplication of 2 and the learning rate μ gives.

$$\tilde{\nabla}_\theta = F_\theta^{-1} \frac{\partial \ell(\theta, x)}{\partial \theta} \tag{24}$$

This expression is the natural gradient denoted $\tilde{\nabla}_\theta$ which simply consists of the dot product between the inverse Fisher and the gradient. It is called natural as F_θ^{-1} can be seen as a correction term which takes into account how the change in parameters affects the output of the model. For

instance if the units are changed from centimeters to meters this would heavily affect the gradients but the Fisher would change accordingly thus cancelling out the effect. In this sense the natural gradient is invariant to affine transformations as well as other transformations.
This is a highly desirable property and a great deal of studies have been inspired hereof and attempted to achieve similar effects by the means of various optimization tricks. Rather than continuously correcting for the negative impact of the potential difference scale of parameters, these methods try to combat the inherent difference in scale before they make rise to problems. Examples of these tricks include anything from the choice of activation function to whitening or normalizing the data, to assigning the parameters individual learning rates and batch normalization etc. The common denominator being to combat "inherent affine transformations" in the data.
The use of natural gradient implies additional desirable properties: Often multiple parameters alter the output distribution in a similar fashion, that is when the parameters are heavily correlated. Consider the example of a model with two heavily correlated parameters. The model gets updates and the two parameters in question are updates in an almost equivalent way. Thus, the particular part of the output distribution which they influence will be either over or under emphasised, since the effect of the update is practically applied twice. Correlation between parameters will be identified by the Fisher and emerge as relatively greater values in the off-diagonal. In this sense the Fisher can be considered a correlation matrix, and the use of the natural gradient will help to mitigate the negative effect of correlated parameters.

Although other methods have been somewhat successful in negating the effect of difference in scales of parameters, the natural gradient shines in comparison to any one of those approaches. Additionally the property of accounting for correlated parameters is quite unique to the natural gradient. The result being that the corrected descent direction achieved by natural gradient descent will result in much larger updates to the model and enable the model to be trained in far less iterations. That being said performing natural gradients comes at a great cost. The Fisher is of size n^2 where n is the number of parameters in the model. For a neural network with hundred thousands or millions of parameters the magnitude of n^2 will easily grow out of proportion making the calculations, storage and inversion of the Fisher infeasible in practice.

4 Kronecker factored approximate curvature

Performing natural gradient descent in its pure form for neural networks optimization will never be able to compete with steepest descent, due to the sheer amount of additional computation required to estimate and invert the Fisher. Not to mention the infeasibility in trying to store the Fisher in memory. For natural gradient descent to be competitive the Fisher must be efficiently and accurately approximated. The state of the art approach for doing so is called Kronecker factored approximations and the method originates from the 2015 paper "Optimizing Neural Networks with Kronecker-factored Approximate Curvature"[5] by James Martens and Roger Grosse. This section will derive the full KFAC theory in its original setting of fully connected neural networks. In addition to the paper the source material for this section includes the KFAC papers associated appendix[22], as well as a lecture[23] on the subject by one of the authors.

4.1 Neural network notation and the Kronecker product

Up until this point the introduced concepts has been independent of a specific machine learning model. This will change momentarily. Specifically optimization of neural networks are of interest, therefore this section will begin by defining basic notation and concepts related to these.
The function $f(x, \theta)$ will be defined as the mapping function of a neural network and the previously arbitrary parameters θ will the now be considered weights of the network. The network consists of l layers of neurons. It takes input x and transforms it into output z. The input is passed through the network one layer at a time by letting each neuron in the next layer receive a weighted sum of all neurons from the previous layer. Prior to being passed on to the next layer, the weighted sum is passed through a non-linear activation function $\phi(\cdot)$. Neither the first nor last layer has an activation function. For layer i the weighted sum prior to its activation is a vector denoted s_i, and after activation a_i. s_i can in this manner be expressed as the product between the weight matrix W_i and the activation's from the previous layer as follows.

$$s_i = W_i \bar{a}_{i-1} \quad a_i = \phi_i(s_i) \tag{25}$$

With this notation the input and output of the network can be written as respectively $x = a_0$ and $z = a_l$ The bar in $\bar{a}_{i-1}$ indicates that each layer includes one additional term namely the bias. This term is often represented separately but is here embedded to keep the math clean. This is accomplished by simply appending a with a 1. The weight matrices equivalently has one additional column often referred to as the bias vector.

Although not necessarily related to neural networks a few other concepts will also be introduced now, to not obstruct or misdirect attention by having to explain them upon use later.
Firstly the vec$(\cdot)$ operator will be of importance. It transforms a matrix into a vector by means of stacking all columns on top of each other. With vec$(\cdot)$ the network parameters θ i.e. the combined weight matrices can be defined as one vector as follows:

$$\theta = \left[\operatorname{vec}(W_1)^\top \operatorname{vec}(W_2)^\top \dots \operatorname{vec}(W_l)^\top\right]^\top \tag{26}$$

Going forward the network weights θ will be a vector represented in this manner.
Lastly there is the Kronecker product[24] which is indicated by the symbol $\otimes$ and can operate on any two combinations of vectors or matrices. Given for instance 2 matrices $\mathbf{A}$ and $\mathbf{B}$ the Kronecker product is given as follows:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix} \tag{27}$$

The resulting matrix has the same shape as the first matrix $\mathbf{A}$, but where each entry is no longer a single scalar but a matrix with the size and shape of the second matrix $\mathbf{B}$, resulting in a block matrix. The first block entry in the resulting block matrix corresponds to the first scalar entry in $\mathbf{A}$ multiplied onto all entries in $\mathbf{B}$, the next block entry corresponds to the next scalar entry of $\mathbf{A}$ multiplied onto all of $\mathbf{B}$ and so forth.
The Kronecker product possesses the following key identities which will play important roles in the following sections.

Transpose identity

$$(\mathbf{A} \otimes \mathbf{B})^\top = \mathbf{A}^\top \otimes \mathbf{B}^\top \tag{28}$$

Mixed product identity

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD}) \tag{29}$$

Inverse identity

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \tag{30}$$

Vector product identity

$$(\mathbf{A} \otimes \mathbf{B}) \operatorname{vec}(X) = \operatorname{vec}\left(\mathbf{B}X\mathbf{A}^\top\right) \tag{31}$$

4.2 Approximating the Fisher information matrix

From the previous section the natural gradient emerged as a means to perform second order optimization and appeared on the form.

$$\tilde{\nabla}_\theta = F_\theta^{-1} \frac{\partial \ell(\theta, x)}{\partial \theta} \tag{32}$$

Where F_θ is the Fisher, which from a classical statistics perspective is given by.

$$F_\theta = \mathrm{E}_x \left[\nabla_\theta \log \left(p_\theta(x) \right)^\top \nabla_\theta \log \left(p_\theta(x) \right) \right] \tag{33}$$

That is, the Fisher information matrix is the expected value of the outer product of the score. The Score being defined as the derivative of the log likelihood of the data under the given parameters. However recall that a neural network may be considered a statistical model that given some input induces a distribution over possible outputs. Correspondingly the optimization procedure of minimizing a loss could similarly be seen as that of minimizing the negative log likelihood. With this in mind the Fisher may be defined not in terms of the derivative of the log likelihood under some parameters but instead the derivative of a loss with respect to the weights of the network. For compact notation said derivative will be expressed as $\mathcal{D}\theta$.
On the note of notation: from here on it will be implicit that the Fisher is with respect to parameters θ and that the expectation is with respect to the inputs x and explicit indication hereof will be excluded from the subscript to reduce notation clutter.
Given the new notation the Fisher associated with the neural network emerges simply as.

$$F = \mathrm{E}\left[\mathcal{D}\theta \mathcal{D}\theta^\top \right] \tag{34}$$

Rather than considering all of the Fisher at once the Fisher can instead be decomposed into a l-by-l block matrix structure. This block structure will be on a per layer basis of the network and the decomposition of layer i and j into their block matrix is given by:

$$F_{i,j} = \mathrm{E}\left[\operatorname{vec}\left(\mathcal{D}W_i \right) \operatorname{vec}\left(\mathcal{D}W_j \right)^\top \right] \tag{35}$$

The derivative of the loss with respect to weights in layer i can be written as.

$$\mathcal{D}W_i = \mathcal{D}s_i \bar{a}_{i-1}^\top \tag{36}$$

Why this holds becomes apparent if you substitute in the derivatives expressed explicitly as well as the definition of $s_i = W_i a_{i-1}$

$$\mathcal{D}W_i = \frac{\partial l}{\partial W_i} = \frac{\partial l}{\partial W_i a_{i-1}} \bar{a}_{i-1}^\top \tag{37}$$

as the term a_{i-1} gets cancelled out, leaving only the trivial result $\frac{\partial l}{\partial W_i} = \frac{\partial l}{\partial W_i}$
Now let the quantity g_i be defined as.

$$g_i = \mathcal{D}s_i \tag{38}$$

Which is the derivative of the loss with respect to the inputs (before activation) at layer i i.e. the gradient of the unit inputs. Substituting this expression into equation(36) yields.

$$\mathcal{D}W_i = g_i \bar{a}_{i-1}^\top \tag{39}$$

Which can be expressed in terms of a Kronecker product.

$$g_i \bar{a}_{i-1}^\top = \bar{a}_{i-1} \otimes g_i \tag{40}$$

The Fisher blocks can now be rewritten using the above expression.

$$F_{i,j} = \mathrm{E}\left[\mathrm{vec}\left(\mathcal{D}W_i\right)\mathrm{vec}\left(\mathcal{D}W_j\right)^\top\right] = \mathrm{E}\left[\left(\bar{a}_{i-1} \otimes g_i\right)\left(\bar{a}_{j-1} \otimes g_j\right)^\top\right] \tag{41}$$

By the means of the Kronecker transpose identity from equation(28), the expression can be further rewritten.

$$\mathrm{E}\left[\left(\bar{a}_{i-1} \otimes g_i\right)\left(\bar{a}_{j-1} \otimes g_j\right)^\top\right] = \mathrm{E}\left[\left(\bar{a}_{i-1} \otimes g_i\right)\left(\bar{a}_{j-1}^\top \otimes g_j^\top\right)\right] \tag{42}$$

Finally with the use of the mixed product Kronecker identity from equation(29) the Fisher block emerges on the form.

$$F_{i,j} = \mathrm{E}\left[\bar{a}_{i-1}\bar{a}_{j-1}^\top \otimes g_i g_j^\top\right] \tag{43}$$

At this point the first approximation will be employed by assuming that the expectation of Kronecker products is equal to the Kronecker product of expectations.

$$F_{i,j} = \mathrm{E}\left[\bar{a}_{i-1}\bar{a}_{j-1}^\top \otimes g_i g_j^\top\right] \approx \mathrm{E}\left[\bar{a}_{i-1}\bar{a}_{j-1}^\top\right] \otimes \mathrm{E}\left[g_i g_j^\top\right] \tag{44}$$

By introducing $\bar{A}_{i,j} = \mathrm{E}\left[\bar{a}_i \bar{a}_j^\top\right]$ and $G_{i,j} = \mathrm{E}\left[g_i g_j^\top\right]$ the expression reduces to.

$$F_{i,j} \approx \bar{A}_{i-1,j-1} \otimes G_{i,j} = \tilde{F}_{i,j} \tag{45}$$

This is indeed a major approximation as the expectation of Kronecker products are in general not equal to the Kronecker product of expectations, and likely wont become exact under any realistic set of assumption. However the approximation proves to be fairly accurate when employed in practice. Note the change of notation to $\tilde{F}$, which indicates that the Fisher is now approximated.

4.3 Structure and interpretation of the inverse

Concider a general multivariate distribution over variables $v = \{v_1, \dots, v_n\}$ with associated covariance matrix denoted Σ. Let C be a matrix with zeroes in the diagonal and let C_i be the i-th row vector. Then entries of C_i are defined as the optimal set of weights for linear prediction of variable v_i from all other variables v.
Then let another matrix D be defined as a diagonal matrix, where each entry in the diagonal for instance $D_{i,i}$ corresponds to the variance of the error associated with the optimal linear prediction of the i-th variable v_i. In this setting it[25] was showed that the inverse covariance matrix can be expressed as:

$$\Sigma^{-1} = D^{-1}(I - C) \tag{46}$$

This result opens up for an intuitive interpretation of the inverse covariance. In the expression each row of the inverse covariance some what corresponds to the weights/coefficients of the optimal linear predictor of the i-th variable. It should be noted that as a result of the matrix multiplication with the diagonal matrix D these weights are adjusted by some scaling factor, but the result remains the same: If some variable j is relatively more useful than other variables for predicting variable i, the entry (i, j) in the inverse covariance matrix can be expected to be relatively bigger than the remaining entries of the i-th row in the inverse covariance matrix.

Instead of considering some arbitrary variable v let instead v be the gradient of the weights. The Fisher then emerges as the covariance of the gradients with respect to the models distribution. The above general result can now be directly transferred to the problem at hand.
Furthermore, if you think of the sequential structure of a fully connected feed forward network it intuitively makes sense that the gradient entries within the same layer $\mathcal{D}W_i$ contains the most information about each other i.e. will be the most useful for predicting one another. Following the same intuition the second best candidate will be entries in the adjacent layers, namely $\mathcal{D}W_{i-1}$ and $\mathcal{D}W_{i+1}$, Whereas entries in layers far from i should carry little to no information about $\mathcal{D}W_i$. Given this view on the relationship between the entries, the Fisher should be well approximated by block-diagonal, or for an even better approximation a block-tridiagonal.

4.4 Approximating the inverse Fisher as a block-diagonal

It was shown that the magnitude of the entries in the inverse Fisher could be interpreted as a measure for how well one gradient entry could predict some other gradient entry. It was further argued that gradients within the same layer and to some extent adjacent layers would be the most suitable for predicting each other, thus justifying either a block-diagonal or a block-tridiagonal Fisher.
The KFAC paper derives both how to approximate the block-diagonal as well as the block-tridiagonal. The tridiagonal method is not merely an extension of the diagonal case but rather it requires some entirely new considerations, concepts and approaches, while not necessarily yielding superior performance in practice. This study will only concern itself with the block-diagonal instance.
Let the block diagonal of the approximate Fisher $\tilde{F}$ be given by.

$$\begin{aligned} \breve{F} &= \operatorname{diag}\left(\tilde{F}_{1,1}, \tilde{F}_{2,2}, \dots, , \tilde{F}_{l,l}\right) \\ &= \operatorname{diag}\left(\bar{A}_{0,0} \otimes G_{1,1}, \bar{A}_{1,1} \otimes G_{2,2}, \dots, \bar{A}_{l-1,l-1} \otimes G_{l,l}\right) \end{aligned} \tag{47}$$

(Notice again the subtle change in notation between the approximate Fisher $\tilde{F}$ and the diagonal approximate Fisher $\breve{F}$)

Inverting the block diagonal is the same as inverting each block individually.

$$\tilde{F}^{-1} = \text{diag}\left((\bar{A}_{0,0} \otimes G_{1,1})^{-1}, (\bar{A}_{1,1} \otimes G_{2,2})^{-1}, \ldots, (\bar{A}_{l-1,l-1} \otimes G_{l,l})^{-1}\right) \tag{48}$$

By using the inverse Kronecker product identity from equation(30) the inverse can be expressed as.

$$\breve{F}^{-1} = \text{diag}\left(\bar{A}_{0,0}^{-1} \otimes G_{1,1}^{-1}, \bar{A}_{1,1}^{-1} \otimes G_{2,2}^{-1}, \ldots, \bar{A}_{l-1,l-1}^{-1} \otimes G_{l,l}^{-1}\right) \tag{49}$$

Thus computing the inverse of the block diagonal Fisher amounts to inverting $2l$ smaller matrices, which computationally is significantly cheaper.

The inverse by itself is not particularly useful, rather it is of interested to compute the product between the inverse and the gradient, as this results in the natural gradient. Recall that the gradient of layer i could be expressed as a column vector by $\text{vec}\left(\mathcal{D}W_i\right)$. Given that the Fisher is a block diagonal so will the natural gradient be and may therefore be index by only one subscript. The approximate natural gradient of layer i emerges as the matrix block.

$$\breve{\nabla}_i = \breve{F}_i^{-1} \, \text{vec}\left(\mathcal{D}W_i\right) \tag{50}$$

Notice the natural gradient has undergone a subtle change of notation to indicate that it is an approximation as well. (At this point it is uncertain if it may even be referred to as the natural gradient at this point) Finally, using the Kronecker identity related to products given by equation 31, the approximate natural gradient of layer i becomes as follows.

$$\breve{\nabla}_i = G_i^{-1} \, \mathcal{D}W_i \, \bar{A}_{i-1}^{-1} \tag{51}$$

The above equation constitutes the update rule for second order neural network optimization with Kronecker factored approximate curvature.

5 KFAC in practice

For the derived update rule to work, the quantities $\bar{A}_i$ and G_i need to be estimated. Additionally, for the update rule to work well, a series of optimization tricks must be employed. The following accounts for everything related to the practical employment of KFAC.

5.1 Computing the required statistics

Recall that the individual blocks of the Fisher approximation were given by:

$$\breve{F}_i = \bar{A}_{i-1} \otimes G_i \tag{52}$$

Where the $\bar{A}_{i-1}$ and G_i in turn were defined as the expectation of outer products of respectively the activations and the back propagated derivatives of the inputs.

$$\bar{A}_i = \mathrm{E}\left[\bar{a}_i \bar{a}_i^\top\right] \text{ and } G_i = \mathrm{E}\left[g_i g_i^\top\right] \tag{53}$$

These quantities need to be computed and continuously updated during the training of the network. The computation of $\mathrm{E}\left[\bar{a}_i \bar{a}_i^\top\right]$ is straight forward since the $\bar{a}_i$'s do not depend on training label t and the expectation can be taken with respect to the empirical training distribution over the inputs x. In practice, $\bar{A}_i$ is calculated as the outer product of $\bar{a}_i$ with itself, and the expectation emerges as taking the average over a batch.
The approach for computing G_i is not as straight forward since g_i's being backpropagated derivatives do depend on the labels t. Therefore the expectation must be taken with respect to both the empirical training distribution as well as the networks predictive distribution.
It is important that the expectation is not taken with respect to the empirical training distribution of labels, as this would result in the computation of the *empirical Fisher information matrix* which is not compatible with the theoretical analysis and perform worse in practice. Furthermore it would break the idea of natural gradient as this procedure is supposed to be independent of the training targets.
For computational efficiency the expectation will be approximated using Monte-Carlo estimates. These will be obtained by sampling labels from the network´s predictive distribution. This is done in the following manner. Perform the usual forward pass on a mini-batch, and save the predictions. In addition to the backward parse with respect to the proper training labels, that is computing the gradient, one additional backward parse will be performed. The additional backward parse will be with respect to a random selection of the just computed model predictions, as if these were the true training labels. These are the quantities g_i which may now be used to compute the outer product and will lastly be averaged over the whole mine batch resulting in G_i.

In regard to the additional computation associated with estimating $\bar{A}_i$ and G_i. It does involve computing a series of outer products as well as averages over mini-batches, however in practice this can be performed efficiently by only one additional matrix product per $\bar{A}_i$ or G_i. Furthermore the activations, $\bar{A}_i$'s and the model prediction distribution are freely available as byproducts (and may originate from the same input data) when computing the gradients, which are required to be computed under any circumstances. The g_i's do require performing the additional modified backwards pass, which is one of the more extensive operations associated with training neural networks. Across all operations associated with performing second order with optimization with Kronecker factored approximate for curvature, estimating $\bar{A}_i$ and G_i (for smaller batch sizes) only constitutes a small fraction of the total computation. The primary computational challenge lies with inverting the Fisher blocks.

5.2 Optimization tricks

The KFAC paper employs a combination of various regularization and optimization tricks. Including a Tikhonov regularization, which in turn is applied as an approximate adaption as to not break the integrity of the Kronecker product. Furthermore they introduce an adaptive rescaling/learning rate subrutine, where they use an exact quadratic model computed with the exact Fisher to rescale a proposed model update. This exact quadratic model is itself regularised by its own dampening term according to the Levenberg-Marquardt rule. The adaptive re-scaling subroutine is essentially a trust region approach which adjust the learning rate accordingly based on trust in the quadratic model, and is heavily inspired by previous work done the same authors [4].
This study's KFAC implementation heavily experimented with especially the the Tikhonov regularization approach. It was found that the regularization parameter would tend toward zero for the model to be able learn, indicating that rather than contributing with regularization the Tikhonov approach induced noise which in turn ruined the structure of the approximate Fisher, thus halting learning. Although the implementation of the Tikhonov approach proved unsuccessful during implementation, this study came up with some tricks of its own which vastly helped the learning process.

5.2.1 Normalizing the statistics

During early implementation it became apparent that especially the gradients of the unit inputs $g_i = \mathcal{D}s_i$ would tend to be small, in the magnitude of 10^{-8}. Recall that these values would subsequently be used in an outer product with themselves namely in $G_i = \mathrm{E}\left[g_i g_i^\top\right]$. In turn this would result in small values being multiplied with one another leading to something even smaller. The proceeding step is then to invert the outer product matrix consisting of very small numbers, which post inversion leads to a matrix constituted of now very large numbers. Conclusively the natural gradient would become disproportionately large, leading to obscure updates unless the learning rate is correspondingly small. Tweaking the learning rate in this scenario is difficult on multiple levels. Within the same training the magnitude of the g_i´s may change drastically (typically in an increasing fashion) leading to unstable training unless the learning rate as well is adjusted dynamically. Given that a good scheme for adjusting the learning rate accordingly throughout the training is found, it is highly unlikely that this scheme generalizes to other learning problems or networks structures.
The original paper addresses these problems with the adaptive rescaling subroutine which was superficially described in the introduction of this section. However this study's implementation goes about this issue in another way.
Recall that the update at layer i is given by $\breve{\nabla}_i = G_i^{-1}\mathcal{D}W_i\bar{A}_{i-1}^{-1}$. Rather than attempting to re-scale the learning rate, (which in practice is simply a scalar which at the next step would be multiplied onto $\breve{\nabla}_i$) it was found that re-scaling the matrices $\bar{A}_i$ and G_i and subsequently the natural gradient itself using dynamic information, yielded promising results. A wide variety of methods were attempted. For instance $\bar{A}_i$ and G_i were standardized by dividing them with their corresponding maximum value and then re-scaled by the square or fourth root of their the absolute value of their corresponding means. The reasoning being that they should be on a reasonable scale but simultaneously retain some of their size proportions relative to each other. Traditional standardisation by the means of subtracting the mean and dividing with the standard deviation was also attempted as well as linearly transform them into the range -1 to 1.
The only normalization approach which yielded promising results for $\breve{\nabla}_i$ were to simply divide it by its max value. It was found that normalizing $\breve{\nabla}_i$ had the greatest impact on the resulting update while normalization of G and A only had a minor impacts and was mostly kept due to numerical stability concerns. In the end, as to not over-complicate matters the final implementation uses the simple divide-by-maximum-value standardization technique for all of the 3 matrices $\bar{A}_i$, G_i and

$\breve{\nabla}_i$.

5.2.2 Momentum

The estimates of $\bar{A}_i$ and G_i have to be maintained and contentiously updated throughout the training. The authors of the KFAC paper proposes decaying average as the superior choice for continuously updating the estimates. They argue that both $\bar{A}_i$ and G_i depend on the models parameters which in turn also get updated throughout the optimization process. Thus a naive equally weighted average would assign to much credibility to previous estimates, estimates which gradually become less relevant as the iterations proceeds.
Concretely they settle on their running estimate being computed as the old estimate weighted by a factor of α plus the estimate from the current batch weighted by a factor of $1 - \alpha$. With α given by $\alpha = \min\{1 - \frac{1}{k}, 0.95\}$ and k being the iteration number. This can essentially be viewed as momentum that progressively increases. The greater the value of α, the more credibility is given to previous iterations. For simplicity α will be referred to as the the momentum factor and the method as a whole will be referred to as momentum.
In the case of this study this approach seemed to converge to too large of a momentum to fast, which negatively impacted learning stability. In the early iterations the new estimates of $\bar{A}_i$ and G_i (prior to being averaged) were so drastically different from the previous iteration that performing the average would leverage too much already outdated information. This is likely caused by the model undergoing huge changes in the first couple of iterations which implies drastic changes to the Fisher as well.
This study settled for a similar decaying average scheme but made the momentum factor α linearly (opposed to exponentially) proportional to the iteration number k by a scaling factor denoted m_{rate} for *momentum scaling rate*. In general the maximum α value of 0.95 seemed to large and it was substituted for a variable parameter m_{max} for *momentum maximum*. Applying these changes yields the following formula for the momentum factor.

$$\alpha = \min\{0.05 * k * m_{rate},\, m_{max}\} \tag{54}$$

Additionally the KFAC paper has a separate momentum scheme for the natural gradient itself, which involves a subroutine to determine its associated momentum factor. This additional scheme was excluded and instead replaced with the one just introduced and shares hyper parameters m_{max} and m_{max}. The reason being that the natural gradient for the most part depends on the Fisher which in turn is a directly constructed from $\bar{A}_i$ and G_i. Thus it can be expected that changes to the Fisher yields corresponding changes to the natural gradient and a similar momentum (in this case the same) seems reasonable, and provides and easy way to reduce unnecessary complexity.

6 Experimental setup and overall approach

Beyond the design choices directly related to KFAC as an optimizer, a great deal of choices has to be accounted for in regards to experimentally testing the KFAC implementation. Hereunder which machine learning problems KFAC should be faced with, their associated data and which neural network architecture may befit the problem. Additionally, the performance of KFAC on said problems is only interesting in comparison to a baseline. Lastly the hyperparameter tuning of KFAC as well as the baseline play an important role for the overall comparison and must be accounted for as well.

6.1 Testing KFAC on a variety of problems

The authors of the KFAC paper only concern themselves with testing their implementation on encoder-decoder problems. At least they have only included results from encoder-decoder problems in their paper, explaining that these problems are very difficult and has become the standard benchmark within neural network optimization literature. Concretely they solve 3 encoder decoder problems all of which originate from the 2006 paper[26].
The orignal KFAC theory and implementation only extends to fully connected feed forward networks, although the theory and implementation has later been extended to more sophisticated architectures. This study, following in the footsteps of the original paper, will stick to fully connected networks as well. These networks are for the most part outdated by modern standards and have been surpassed by more specialised, domain specific and advanced architectures. At this point fully connected network are a jack of all trades but still performs relatively well on encoder decoder problems, hereby helping to justify why the KFAC paper only concern themselves with these problems. Regression and classification problems on non structured data (opposed to image or sound data for instance) are another domain where fully connected networks have not been greatly surpassed either. This study´s implementation of KFAC serves primarily as a proof of concept and therefore does not have to limit itself to testing the KFAC on problems where fully connected networks are still relevant but can be extended to problems which by today's standards are perceived outdated to employ fully connected networks on. An obvious example of this is image classification. The purpose of the study is not to investigate how well a network can solve a specific problem, but rather how fast and efficiently can the network reach its full potential regarding solving said problem.
The above lays the ground for not only testing KFAC on a variety of problems but also to attempt problems where fully connected networks are considered inferior, in order to investigate the learning capabilities in as many settings as possible. In line with the KFAC paper this study will solve one of the encoder-decoder problems. Furthermore KFAC will be faced with a regression as well as an image classification problem.

6.2 Data and prepossessing

The 1. of the 3 problems in question is the encoder decoder problem on the mnist digits dataset[27], which also appears in the original paper. The dataset consists of 60,000 training images and 10,000 test images. The images are 28x28 pixels (784 total) gray scale images and depict one handwritten digit between 0 and 9 each. In practice they will be on the vector form achieved by stacking the columns of each image on top of each other. The only preprocessing conducted was to scale the pixel values of all images down to lie between 0 and 1, i.e dividing by the maximum possible pixel value 255.

The 2. problem is a regression problem. The associated dataset is called *Relative location of CT slices*[28] and has 53,500 samples. Each sample originates from one CT image from which

2 histograms were created. The first histogram describes the bone structure while the other describes the location of air inclusions inside the body. The 2 histograms are expressed as a total of 384 polar coordinates. These coordinates constitute the attributes of the training data for the regression problem. The value which is attempted to predict is a number between 0-180 that indicates the relative location of the image on the axial axis. 0 denoting the top of the head and 180 the sole of the feet. From the data set 5,000 samples were put aside and used as a test set. The only preprocessing performed is to scale the target value from being between 0-180 down to being between 0 and 1.
There are no particular strong reason for this choice of dataset/problem. The choice came to down it meeting the criteria of having previously been used solved as a regression problem and it having relatively many (for a regression dataset) attributes as well as samples.

The final task on which KFAC will be tested is that of a 10-class image classification problem. Specifically the cifar-10 image classification dataset[29], which consists of 50,000 training images. The images depict general classes such as: dogs, cats, ships, houses etc. Each image is 32x32 but rather than being gray scale they have 3 color channels. For the final build the images were kept in RGB to enforce the use of a bigger network. The information gained from maintaining the full information of all 3 colors barely seemed to improve accuracy potential of the models, but greatly increased training time. This is of no concern to this study as the focus is not how to efficiently use data or which architecture works the best but rather how to efficiently train a network to its full potential even in subpar circumstances. The images are reshaped by stacking all columns vectors of each channel on top of each other and then stacking all 3 channels resulting in one image having the shape of 3072 by 1. Similarly to the digits data, it is downscaled by dividing by 255 and has a separate test set consisting of 10,000 images.

It should be emphasises that whenever a model is evaluated, the evaluation in question will always be in terms of the full test sets. That is, the model performance over the full 10,000, 5,000 and 10,000 (depending on the problem) respective test samples.

6.3 A note on regularization and overfitting

The KFAC paper does not concern itself with test data or reporting the results in terms of test loss but sticks purely with training loss, arguing that they only care about optimization speed and not generalization capabilities. An argument for disregard test data set is that the simple encoder problems from the KFAC paper does not seem to overfit: During the experimental work of this study the encode decode problem on the mnist digits data set did not overfit nor did the cifar-10 data set overfit when framed as a encode decode problem. That being said, overfitting proved a big problem when using the cifar-10 data set for classification. For unambiguity, this study will only report the test loss as it is critical to the cifar-10 classification, while considering test loss instead of train loss is insignificant with regard to the results of the other 2 tasks.
To combat the issue of overfitting in the KFAC setting l2 regularization was manually implemented. L2 regularization is given by:

$$\ell_\theta = \sum_{i=1}^{n} \text{loss}\left(t_i, z_i\right) + \lambda \sum_{j=1}^{p} \theta_j^2 \tag{55}$$

Which is essentially the sum of the squared weights scaled by some factor λ and then added to the total loss. The expression introduces regularization by penalizing big, i.e. dominant weights.
It was found that l2 regularization addresses the issue of overfitting but at the cost of severely increasing training time while not enabling higher performance ceiling capabilities of the model. Additionally the inclusion of l2 regularization introduces yet another hyperparameter, λ which

needs tuning. Thus it was deemed a better solution to simply monitor test loss and terminate training right before overfitting occurs.

6.4 Activation function and weight initialization

This study and the KFAC paper implementation differ with regard to activation function and weight initialization, as a result of this study opting for more *modern* approaches. Rather than using the sigmoid (sometimes also denoted the logistic) activation function, this implementation uses the hyperbolic tangent activation function. Furthermore, rather than initializing the weights according "sparse initialization" technique from a previous paper[4] by the KFAC authors, all weights were initialized according "Xavier initialization"[30]. A popular weight initialization scheme which is designed for and thus commonly paired with the hyperbolic tangent activation function. It is a uniform distribution bounded by:

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \tag{56}$$

Where n_i is the number of incoming weights and n_{i+1} is number of outgoing weights for a given layer. The reasoning for these subtle and seemingly unnecessary design changes comes down them greatly increasing learning capabilities and stability during training and consistency between training runs.

6.5 Neural network architecture

The neural network architecture used for mnist digits encoder decoder problem is equivalent to the one used in the KFAC paper, which in turn is identical to one from the 2006 paper[26]. With neurons per layer structured in the following manner: 784-1000-500-250-30-250-500-1000-784, amounting to a total of 9 layers (7 hidden layers). The training of the encoder-decoder network used the mean squared reconstruction loss, which is simply the pixel wise squared difference between the input and the output averaged over the whole image.
The neuron and layer structure of the regression network has no particular origin and is merely an adaptation of the encoder-decoder network but scaled down slightly to better match in the input size and reshaped to match the output of the problem. The neuron per layer structure is as follows: 384-500-750-500-250-100-1, here amounting to 7 layers (5 hidden). The loss function for the training of the regression network was the mean squared error.
Similarly the classification network architecture has no particular origin but similarly bears resemblance with the encoder decoder network but reshaped and resized to match the input and output size thus resulting in a neuron per layer structure of: 3072-3000-2000-1500-1000-250-10, yielding again 7 layers total (5 hidden). The cross entropy loss was used for the optimization of the classification network.
Across all 3 networks the hyperbolic tangent activations were applied after each hidden layer. Furthermore the softmax function was applied to the output of the classification network, whereas the output of the encoder-decoder and the regression networks were left untouched.

A note on the bias term. Unintentionally all experiments were conducted with networks, which did not have bias terms in their layers. This is best attributed to an inadvertent consequence of the iterative nature of implementing an algorithm from the ground up. That is, everything is kept to a bare minimum initially and then gradually built upon and expanded whenever the current iteration works. This approach leads to the bias term being left out of the KFAC implementation initially and then unintentionally forgotten about. The absence of the bias was discovered very late in the process, well after all experiments had been conducted and the results had been produced and discussed. The effect of not including the bias is undesirable and has most definitely affected

performance negatively. However, this study's primary concern is not the peak performance but rather the relative performance between the 2 optimization methods. Therefore it is emphasised that **the nature of this comparison is still fair** as neither method had biases enabled in their networks. Lastly, the introduced KFAC theory clearly supports optimization over the bias term, and this unfortunate incident has nothing to do with technical nor theoretical limitations.

6.6 The baseline: SGD

The baseline with which KFAC will be compared is the most widely used optimization method for training neural networks, stochastic gradient descent (SGD). To stay in line with the KFAC paper and enhance the capabilities of the method, it will more specifically be stochastic gradient descent with Nesterov momentum[31]. Nesterov momentum closely resembles ordinary momentum and the momentum applied to the KFAC method, but has one key difference. Rather than the gradient term being calculated at the current position in parameter space θ_t the gradient is instead calculated at the position which also takes into account the momentum being carried over from the previous iteration. The intuition being that the gradients always point in the direction of steepest descent but the momentum from the previous iteration may not. Therefore to compensate, this needs to be taken into account, by assuming the position of the momentum when calculating the gradients.

$$\begin{aligned} v_{t+1} &= \eta v_t - \mu \nabla \ell \left(\theta_t + \eta v_t\right) \\ \theta_{t+1} &= \theta_t + v_{t+1} \end{aligned} \tag{57}$$

here η is the momentum factor which determines how much the velocity, v, may change at a given iteration i.e. how much the gradient at the current step can alter the velocity.
SGD with Nesterov momentum is not to be considered state of the art within the literature and numerous more sophisticated learning algorithms like ADAM and RMSprop to name a few have often been shown to outperform SGD [32]. However these methods have also been suspected[32] of generalizing worse (especially ADAM) and mostly outperforming SGD due to being less sensitive to their hyperparameters thus easier to tune as well as having great initial progress but then tending to slow down. As with everything related to neural networks there are no one superior optimization methods and it comes down to the specific instance. The recurrence and widespreadness within the literature as well as its simplicity yet effectiveness makes SGD the obvious choice as a baseline in general.

6.7 Bayesian optimization for hyper parameter tuning

This study strives to compare KFAC to SGD in 3 different settings. A encoder-decoder problem, a regression problem and a classification problem. The KFAC implementation has 4 central hyper-parameters which need tuning: Batch size, learning rate, maximum momentum and momentum scaling rate. SGD has 3: batch size, learning rate and momentum. Across all 3 problems and both optimization methods a total of 21 parameters needs tuning. In an attempt to make the process of tuning these parameters as unbiased and fair as possible Bayesian optimization (BO for short) was employed.

The theory related to how BO functions are beyond the interest and scope of this study. For the purpose of the study BO will be considered a black box tool which iteratively and efficiently finds the minimum of an unknown function. In this case the function is the training of a network and the adjustable parameters are the aforementioned hyperparameters, with the task being to find the parameters which results in the best possible training. Here best will be defined in terms of reaching the lowest test loss in the shortest amount of time. This will be enforced by letting the

networks train for a fixed amount of time rather than a fixed amount of data or iterations. For a reasonably tuned either SGD or KFAC optimized model the classification and encode-decoder problems can be "solved" in roughly 10 seconds. Solved in the sense that the learning curve has flattened a great deal and the model has achieved most of its learning potential given the circumstances. Stopping the training almost prematurely helps to better distinguish the good configurations, as given enough time most of the configurations would eventually reach good performance given the easy nature of the problems. The short training time also enables running each training at least twice to minimize the effect of a lucky run/weight initialization. Furthermore the short training time in general enables more iterations of BO thus increasing the odds of reaching (or at least close to) an optimal set of parameters.

The BO implementation used to conduct the hyperparameter search is a python library called GPyOpt[33]. All parameters were untouched and left at their default values. For the encoder-decoder and classification problem, the models were trained for 10 seconds two times and the average of the final two test losses were returned to the Bayesian Optimizer, which then picked a new set of parameters and repeated the process. The regression problem being "easier" to solve, was trained 4 times of 5 seconds each instead.
In all 3 instances the BO ran 20 initial iterations with random parameters before starting to pick parameters according to its internal optimization procedure. It subsequently ran for 45 minutes amounting to a total of approximately 150 iterations, depending on initialization time of the approaches which differed slightly. However train time was strictly measured to respectively 10 and 5 seconds and did not include any initialization or noise of the like.
The 4 hyperparameter configurations (the models themselves were not saved) which yielded the best average test loss for each optimization method (SGD or KFAC) for each problem will subsequently be used for the comparison of the 2 methods. The choice of going with the top 4 best models rather than just the best and retraining it 4 times was deemed a more natural way of displaying variability between training runs, and hyperparameter independent behavior.

6.8 Technical implementation details

All code was written in python using the PyTorch framework. Everything related to the KFAC optimizer was implemented completely from the ground up. The SGD with Nesterov momentum used for benchmarking is the native PyTorch implementation. All training was GPU accelerated and conducted on a GTX 980 paired with an i5-4690k and 16gb of ram. All code can be found at: `https://github.com/KarlUlbaek/KFAC-implementation`

7 Batch size properties of KFAC

The effectiveness of the BO process and the results of the comparison across all 3 problems will be presented in the next section. Now however, the attention will be directed towards exploring some of the properties of KFAC. For this section the 4 best/optimal hyperparameter configurations as found by BO for the encoder-decoder problem will be employed and modified depending on which property is investigated. This investigation is only presented for the encoder-decoder problem but similar results were found for the 2 other problems. If the reader is curious to the specifics of the optimal hyperparameter configurations in question, they will be presented in full later on but can be found in table{1}.

7.1 Scaling with batch size

The authors of the KFAC paper identified that KFAC benefited from greater batch sizes and for their experiments used a batch size of magnitude 500 or greater. The impact of batch size was similarly investigated in this study. The following plot displays the impact of batch size under training time constraints and update-iteration constraints.

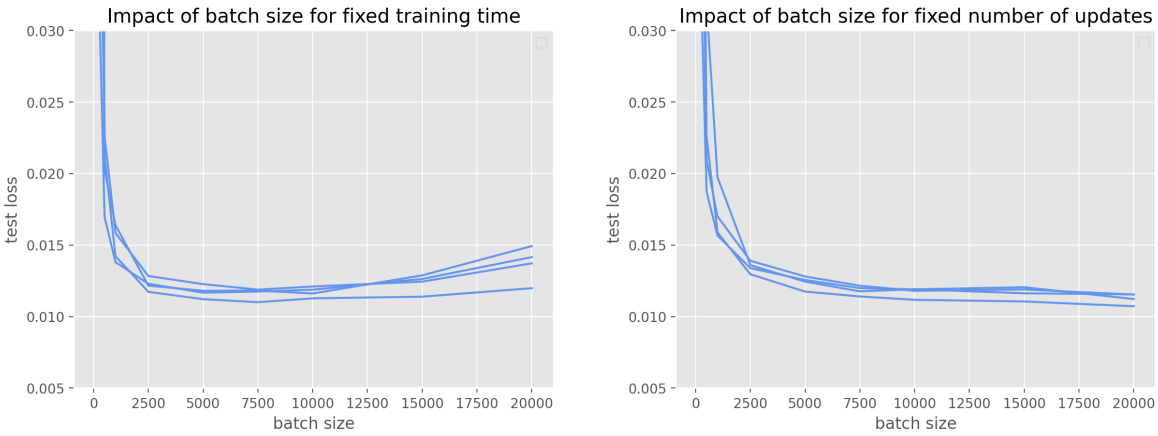

Figure 2: Both plots display the final test loss as a function batch size for the encoder-decoder problem. Each line corresponds to one of the 4 best hyper configurations as found by BO. The time constrained training ran for 10 seconds, while the update constrained ran for 50 updates.

From the right plot it appears that the greater the batch size the greater the learning capabilities per iteration, although severe diminishing returns sets in when exceeding a batch size of 5000. When the learning is time constrained rather than iteration constrained a sweet spot in the 5000-10000 area emerges. In other words, when KFAC is forced to learn as efficiently as possible with respect to time it settles for a quite big batch size. The tendency to prefer rather big batch sizes is best explained by the role of the matrices $\bar{A}_i$ or G_i, as the estimate of these needs to be sufficiently good for KFAC to make meaningful updates. The quality of the estimate is directly dependent on the amount of data used in the estimation, that is, the batch size. Although as indicated by the right plot, the estimates of $\bar{A}_i$ or G_i can become sufficiently good to the point where utilizing further data barely improves performance.
Another reason as to why KFAC prefers big batches even when optimizing under time constraints, could be explained by one of the big computational challenges of KFAC, matrix inversion. This study will by no means attempt to analytically quantify the computational complexity of KFAC,

but will merely state a few observations regarding it. Matrix inversion is a taxing computation and scales drastically with the size of the matrix, however the matrix inversion component of the implemented KFAC algorithm is independent of the batch size: Per iteration of KFAC the time spend inverting remains constant no matter the batch size. When optimizing under time constraints this property will most likely drag the batch sizes towards greater values to ensure that time costly matrix inversion operations are kept to a minimum and only performed on sufficiently good estimates of $\bar{A}_i$ or G_i that may lead to meaningful model updates. This notion of how much time KFAC spends inverting matrices is quantified in the right plot of figure{3}.

7.2 Batch size adjusted for learning rate

Additionally from figure{2} it can be observed that KFAC struggles to learn when the batch size is too small. Especially from 500 and down. This can easily be missed and pinpointing this number clearly is difficult on the presented plots as the x-axis spans from 0 to 20.000. Furthermore this behavior might simply be caused by the learning rate being too large as a result of the general connection between small batch size needing small learning rates due to lack of confidence in the estimated gradients.
The following plot seeks to further investigate KFACs learning capabilities when the batch size is small.

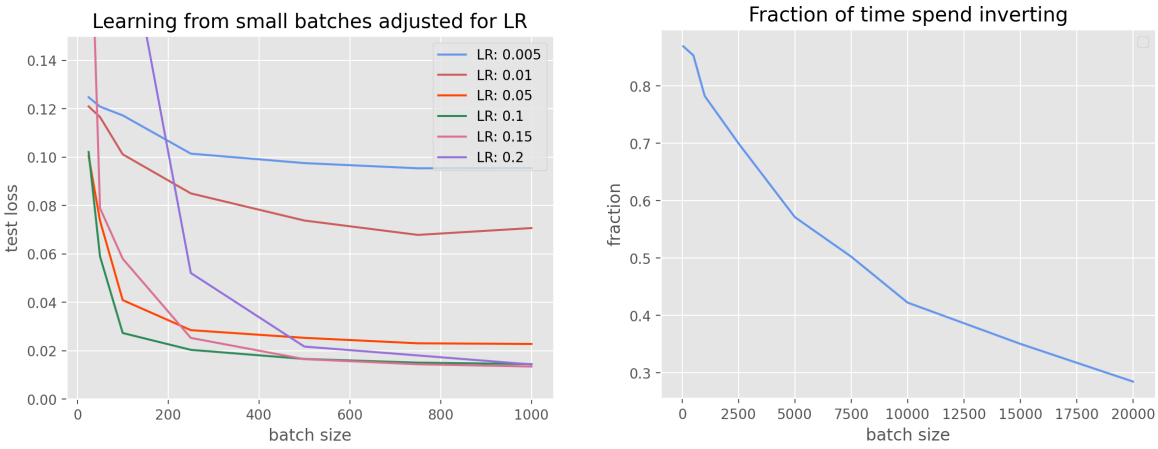

Figure 3: **The left** plot displays the final test loss as a function of batch size when training on the encoder-decoder problem for 10 seconds. For this plot only one (the best) of the hyperparameter configuration found by BO was used, but at different learning rates as indicated by the colors. **The right** plot shows the fraction of time per iteration KFAC spends on matrix inversion as a function of batch size. The plot should not be confused with the inversion time itself decreasing as a function of batch size. The inversion time is constant. It is rather the time spent on other computations which scales with batch size, thus reducing the relative time spent on matrix inversion.

For some perspective and context on the left plot of figure{3}: An untrained model will have a loss around 0.12. and the best configured models from the previous figure{2} had a loss around 0.012. With that in mind it is evident that with a learning rate in the 0.15-0.2 range (pink and purple) and a sufficiently small batch size (less than 100) KFAC is practically unable to learn. This span of learning rates (0.15-0.2) roughly corresponds to the 4 optimal learning rates found by BO, and it is clear these are tweaked for a much bigger batch size.

That being said, even when adjusted for learning rate KFAC is practically unable to learn from a batch size of 25 with the lowest observed loss being 0.10 (recall that an untrained model has a loss of 0.12), and still struggles severely at a batch size of 50 with the lowest observed loss being 0,06. First at a batch size of 100 KFAC starts to find footing with the need of batch sizes as big as 500 to learn efficiently.

Speculations as to why KFAC enjoys greater batch sizes have been brought up during the discussion of the previous figure{2}. The implications of KFAC needing big batch sizes to facilitate learning has not been addressed: The primary concern which comes to mind is GPU memory limitations. As of this study, GPU memory has not been a issue due to the small scale and simplicity of the problems, however if KFAC is to be applied in anything but toy examples, while still remaining competitive with other optimizers, this must be addressed. To further worsen the issue KFAC in general requires more GPU memory than SGD, due to the additional bookkeeping and computations per iteration. As it was the case with time complexity, quantifying memory complexity is beyond the scope and interest of this study.

7.3 Proposing artificial batch sizes

Another paper[13] has addressed the memory concern to some extend, but here the proposed solution involved distributing the computation to multiple machines, which in most circumstances is highly inconvenient.
This study proposes and implements one solution to the memory concern which does not require multiple machines: Instead of updating the model parameters every batch, only the estimates of the matrices $\bar{A}_i$, G_i as well as the gradient matrix are updated. These quantities were already calculated as averages over one batch, and thus averaging them over multiple batches is completely viable. Only when a certain "artificial batch size" is reached will the network get updated. This enables the use of arbitrarily big batch sizes, when GPU memory restricts the actual batch size.

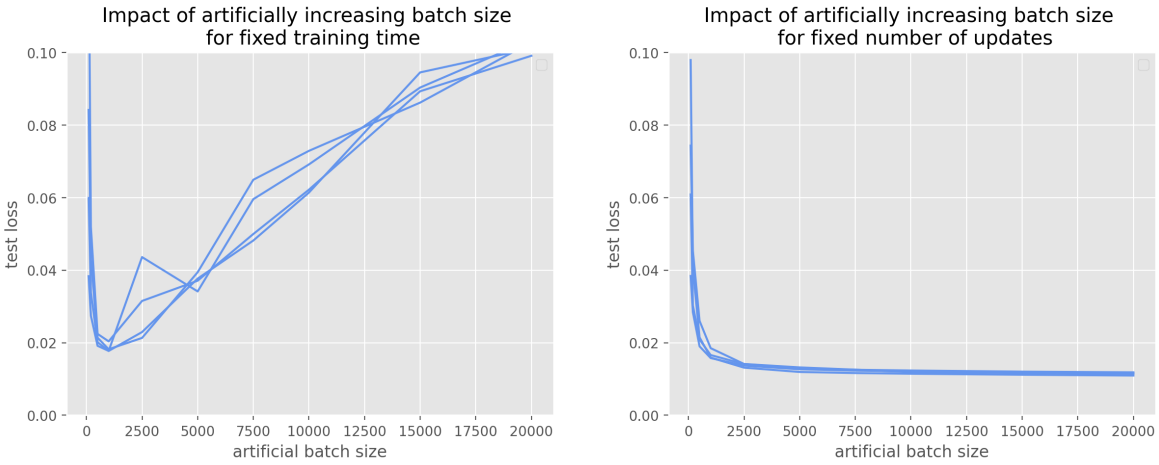

Figure 4: Both plots display the final test loss as a function of **artificial batch size** for the encoder-decoder problem. The actual batch size was fixed to 25. Each line corresponds to one of the 4 optimal hyper configurations as found by BO. The time constrained training ran for 10 seconds, while the update constrained ran for 50 updates.

From the left plot of figure{3} it became apparent that even when adjusted for learning rate KFAC

was completely unable to learn from batches of size 25. From the left plot of figure{4} it is evident that with the proposed *artificial batch size approach* KFAC is able to reach a respectable test loss of 0.02, this is achieved at an artificial batch size of 1000. Previously KFAC (as seen in figure{2}) would tend towards a batch size of 5000-10000 for optimal efficiently, this is no longer the case as the internal efficiency dynamics are shifted when using artificial batch sizes. This is likely a result of having to perform several more forward parses and backward parses, due to the small batches used.
However, if the training is not time constrained but rather iteration constrained as seen in the right plot of figure{4} KFAC is practically unaffected by using artificial batch size versus actual batch sizes.
This study will not attempt to compare the concept of artificial batch size to other methods, approaches or optimizers. It primarily serves as a prove of concept which enables KFAC to learn even in circumstances where batch size is limited due to (for instance) GPU memory constraints. Neither was in depth investigations of exactly when to employ the artificial batch size approach conducted. But as a general rule of thumb it was found that:

- Always use the biggest possible batch size.
- If this batch size is less than 250 artificially increasing it will lead to improved performance.
- If this batch size is around 500 artificially increasing it most likely will improve performance.
- If this batch size is around 1000 artificially increasing it may improve performance.

8 Results and discussion

8.1 Results of Bayesian optimization

This study has put a great deal of emphasis on hyperparameter tuning and settled for BO as a means to conduct the tuning in an effective and unbiased way. But did it work as intended, and what are the results of the tuning process itself? As a sanity check the results of the BO process will be examined.

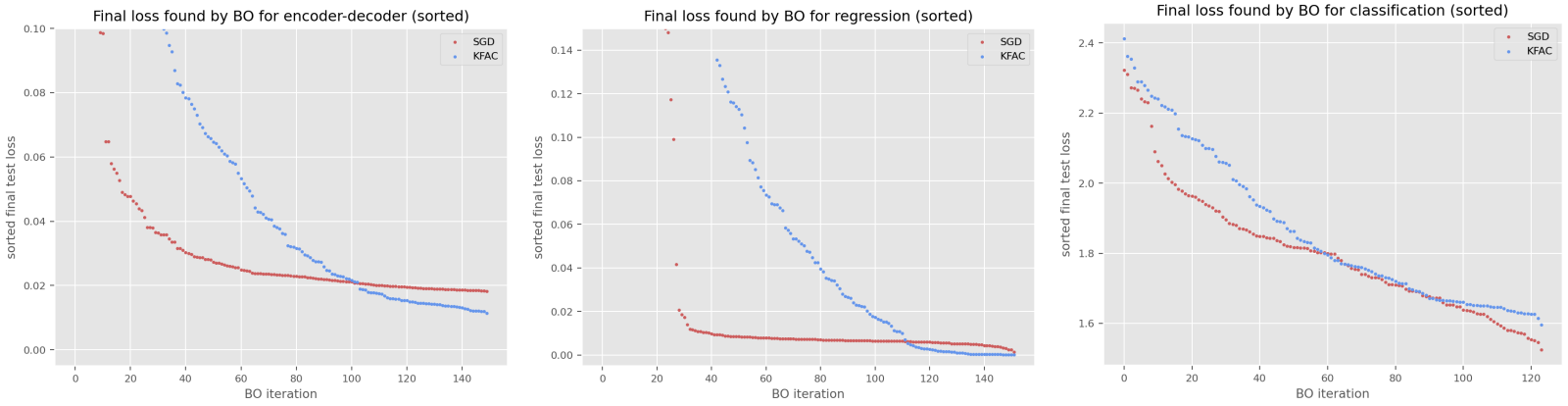

Figure 5: All losses found by Bayesian optimization for the 2 optimization methods on all 3 problems. The losses have been sorted from lowest to highest. The x-axis does **not** express the chronological order of the Bayesian optimization iterations. The x-axis is meaningless and should be disregarded. Larger versions of the plots can be found in the appendix.

The above plots serve to prove that Bayesian optimization does indeed find configurations of parameters which yield vastly different losses. Furthermore it displays to which extend this is the case. The graph of sorted losses for SGD for the encoder-decoder problem and the regression problem are rather flat indication that for these problems a wide variety of parameters yield good results (This observation is heavily based on the assumption that the Bayesian optimizer keeps exploring and does not just bounce around parameters close to previous parameters thus yielding similar losses). In this sense KFAC seems to be more parameter dependent, as it has its losses not nearly as concentrated around tight intervals across the three problems.
As to the quality of the parameters found: Both with regard to KFAC and SGD the parameters were better than what was found by manual tuning. Whether this is due to sheer amount of different parameter configurations tested or because BO truly is able to efficiently search the parameter space is of less relevance. What matters is that the human element was removed from the equation, making the comparison less biased towards either method.

8.2 The best hyperparameters found by Bayesian optimization

Furthermore there are insights to be found in the concrete optimal/best parameter configurations found by BO. The following tables constitute all relevant parameters found for each problem and each method.

SGD			
Loss	LR	Batch size	M
0.0182	1.2967	470	0.6931
0.0183	0.9777	842	0.7809
0.0183	1.3582	122	0.2219
0.0185	1.988	248	0.7669

K-FAC				
Loss	LR	Batch size	m_{rate}	m_{max}
0.0113	0.1809	9543	0.5279	0.6066
0.0119	0.198	10109	0.779	0.5447
0.012	0.1438	4419	1.1906	0.5561
0.0121	0.1595	4765	1.7628	0.3936

Table 1: Top 4 best final losses and their associated parameters found by BO for the encoder decoder problem.

SGD			
Loss	LR	Batch size	M
0.0014	0.0064	3523	0.99
0.0024	0.006	588	0.9019
0.0024	0.0104	595	0.8382
0.0032	0.0138	1789	0.812

K-FAC				
Loss	LR	Batch size	m_{rate}	m_{max}
0.00013	0.0392	13869	1.1173	0.7309
0.00016	0.0414	12236	0.5753	0.8381
0.00017	0.1394	13433	0.5718	0.938
0.00020	0.1515	12727	0.9036	0.7516

Table 2: 4 best final losses and corresponding parameters found by BO for the regression problem.

SGD			
Loss	LR	Batch size	M
1.525	0.011	721	0.971
1.5457	0.0092	714	0.9067
1.5506	0.0062	503	0.8712
1.5532	0.0094	721	0.929

K-FAC				
Loss	LR	Batch size	m_{rate}	m_{max}
1.5965	0.2731	8111	1.9215	0.349
1.6146	0.2731	8110	1.6143	0.4392
1.626	0.2844	8027	0.9522	0.3131
1.627	0.2856	8043	1.5372	0.3755

Table 3: 4 best final losses and corresponding parameters found by BO for the classification problem.

It does not make much sense to try and interpret the best parameter configurations for each problem and each method individually. Findings for one problem hardly generalizes to the underlying properties of the optimization method. Rather it will be attempted to point out patterns across the problems.
It seems that SGD in general enjoys a high value for the momentum setting in the range of 0.7 and above. This does not seem to be the case for KFAC and even less so when considering that KFAC starts at a very low momentum and then gradually increases it in accordance to the momentum rate parameter before eventually reaching the max momentum. This behavior is best explained by KFAC taking relatively large jumps in the parameter space i.e. making big changes to the model and thus being unable to relay on previous information, while the opposite is true for SGD.
During the interpretations of figure{5} it was argued that SGD had a large amount of parameters which yielded good results. It was speculated that this could be due to Bayesian Optimization not exploring but rather bouncing around in the neighborhood of already good configurations. From the looks of the values of the concrete parameters found by Bayesian Optimization there seems to be a lot of diversity within the same problems. This observation supports the notion of SGD having a wide range of parameters that will yield a good model.
Here the opposite is true for KFAC, as the parameters within each problem (especially batch size and LR, arguably also the most important parameters) seems to be more closely grouped. Furthermore, this trend with regard to KFAC seems to apply across the problems, with the majority

of learning rates being within a tight interval of 0.14-0.28 with a few exceptions. This is similarly a trend with regard to the batch size preferred by KFAC, as this parameter mostly lies within an interval of 8000-14000. Similar conclusions cannot be drawn for SGD.

From figure{5} it could be observed that only few parameter configurations yield great results for KFAC. As just discussed, these configurations are similar across problems. In other words, KFAC has a smaller quantity of good parameter configurations, however they seem to be consistent across problems. This parameter similarity across problems enables model tuning based on notions and experience from other/past problems, rather than having to start from scratch whenever a new problem is faced, which is arguably the case for SGD. Obviously, this property is redundant and indifferent in the setting of this study where the models can be trained in a matter of 10 seconds and BO easily can be run for hundreds of iterations. However, for non-toy problems being able to draw on experience is highly desirable, and could potentially significantly speed up hyperparameter tuning.

8.3 Main comparison of KFAC versus SGD

The following will serve as the main comparison between the two optimization approaches. For each problem KFAC and SGD will be represented by their 4 best hyperparameter configurations as found by BO and presented in table{1}{2}{3}. The comparison will be in terms of the **test learning-curve over time**. The time horizon of interest and thus the one being plotted is the same 10 and 5 seconds which was used during the BO tuning process. In addition, but of less relevance, the **test learning-curve as a function of model updates** will be presented as well. One last thing to note: each point on the learning curve lines corresponds to the evaluation of the full test sets of respectively 10,000, 5,000 and 10,000 test samples.

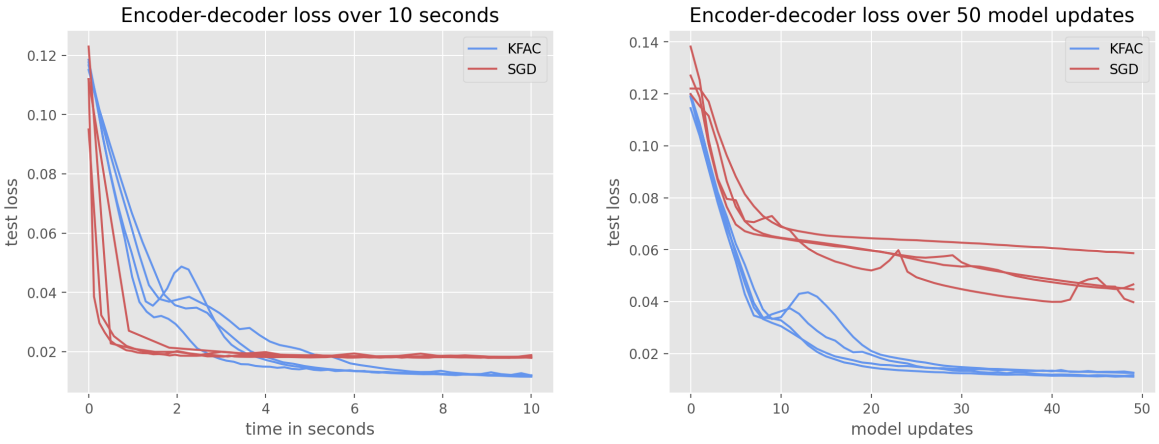

Figure 6: Test loss as a function of time and number of updates for the 4 best KFAC and SGD models on the encoder-decoder problem.

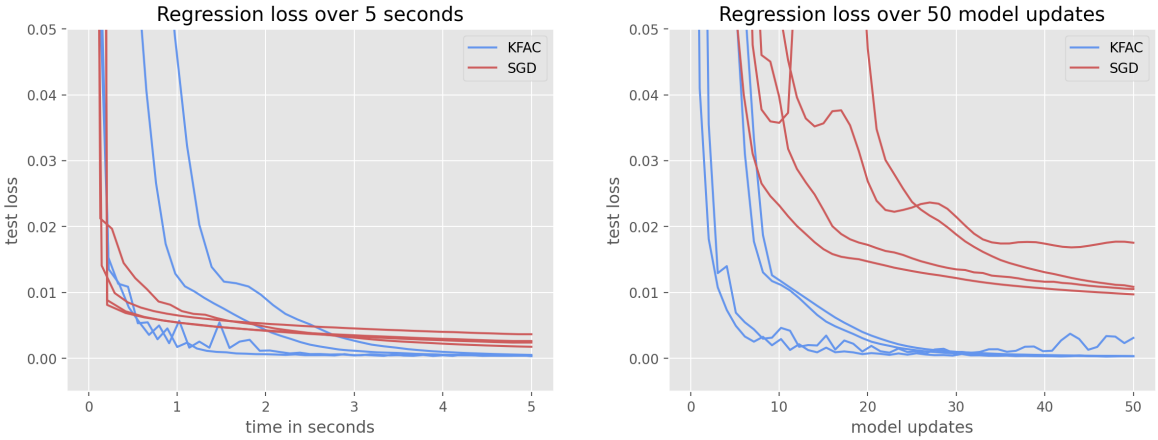

Figure 7: Test loss as a function of time and number of updates for the 4 best KFAC and SGD models on the regression problem.

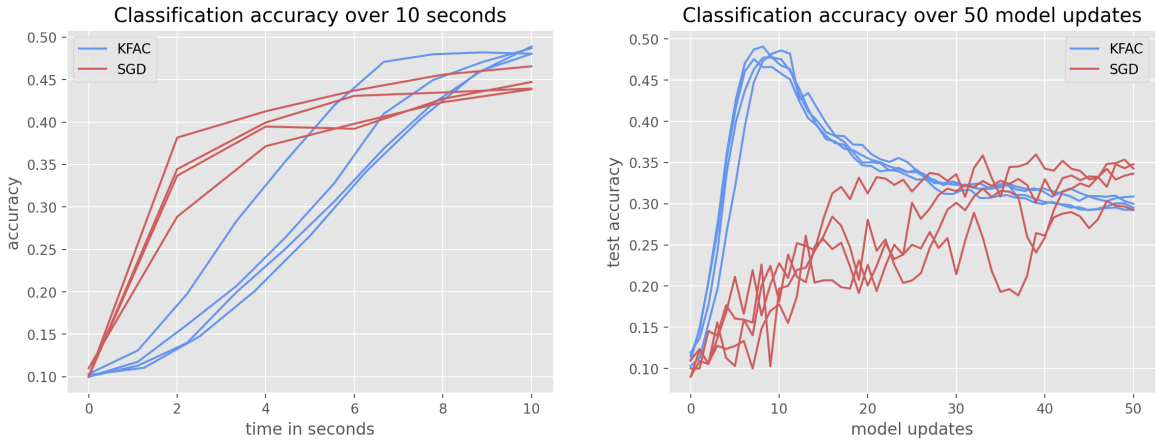

Figure 8: **Test accuracy** as a function of time and number of updates for the 4 best KFAC and SGD models on the classification problem.

8.3.1 Update constrained optimization

KFAC displays clear superiority when comparing the 2 methods on the basis of number of model updates. This behavior was to be expected and is not particular interesting, but can give some notion about the relative magnitude of the updates being made by either method. It should be kept in mind that the BO was performed under time constraints and not update constraints. In either setting KFAC prioritizes a large batch size, but had the BO been update constrained SGD would prefer larger batch size to enable bigger learning rate and through that larger updates thus narrowing the gab slightly.
A tendency across all problems is that SGD improves rapidly in the first couple of iterations, but quickly reaches some threshold and greatly slows down to the point where further iterations barely yield any improvement. For KFAC the transition from the initial learning is considerably less

extreme, and it seems as if KFAC is able to force itself through the threshold and continue to make meaningful updates.

8.3.2 Time constrained optimization

Although not addressed directly the outcome of the time constrained comparison between the two methods, should be of no surprise if the reader has been paying any attention to the contents of the previously discussed tables. With regard to the encoder-decoder and the regression problem all KFAC models manages to achieve a better loss than all SGD models after having trained for the same amount of time. One might start to suspect the reason why the KFAC paper only choose to compare the methods on encoder-decoder problems, as KFAC seems to perform especially well on that particular kind of problem.

Determining which method performs better on the time constrained classification problem is less straight forward. The nature of classification problems enables the use of another and more relatable performance metric, that is the classification accuracy. From the plot it is evident that all KFAC trained models manages to beat all SGD trained models. However from table{3} it can be seen that SGD manages a better loss, which is also the case if one was to inspect the loss plot for the classification problem which can be found in the appendix figure{9}.
It is indeed the case that SGD consistently achieves a better loss, while KFAC consistently obtains a better accuracy. This can be attributed to one of two things (or both): It might be the case that for an incorrectly predicted label the prediction distribution of KFAC for that input is "more wrong": in the sense that KFAC attributes barely any mass to the correct target, thus yielding a worse loss. Equally likely is that when KFAC predicts correctly it was rather uncertain. That is, in the prediction distribution the correct target only has slightly more mass than the incorrect targets, and hereby yielding a correct label but a poor loss. Without going into the discussion of the importance of not producing false positives or false negatives in certain real world applications this study will deem accuracy more important than loss for classifications problems in general. On that foundation KFAC barely beats SGD on the classification problem as well.
From inspection of the update constrained plot, it is apparent that KFAC suffers heavily from overfitting. Already after the first 10 iterations, which is roughly equivalent to 10 seconds, KFAC starts to overfit. The training loss/accuracy will continue to improve rapidly beyond the first 10 updates but these improvements do not carry over to test loss/accuracy performance. Overfitting to this extend did not take place for SGD even after being trained for a significant amount of time. In the literature[14] it has been shown that training on bigger batches are associated with a greater tendency to overfit and generalize poorly. KFAC does indeed have a preference for large batch sizes and as indicated by the classification problem, this property might very well apply to KFAC also. Overfitting was not an issue in either of the other problems and KFACs hypothesises tendency to overfit was not further investigated.

8.4 Summing up the comparison

The overall takeaways from the comparison of the two optimization approaches are as follows:

- Within each problem (and given the respective performance measures) all 4 KFAC trained models beat all 4 corresponding SGD models. Amounting to all 12 KFAC models surpassing their SGD counterparts.
- KFAC significantly outperforms SGD on the encoder decoder problem.
- KFAC convincingly outperforms SGD on the regression problem.

- KFAC outperforms SGD with respect to test accuracy but not in terms of test loss on the classification problem.
- In the case of KFAC, good hyperparameter configurations are similar across problems. This is not the case for SGD.
- SGD seems to have a greater amount and greater variety of parameters configurations which works well i.e. it is less dependant on a narrow set of parameters.

Given the exact circumstances of the experiments conducted in this study, KFAC is the overall better optimizer in comparison to SGD with Nestorov momentum.

8.5 Further discussion and future work

The results found in this study gives a more generalized picture of what can be expected of KFAC compared to the results found in the original paper for two important reasons. Firstly, KFAC is tested on a multitude of problems rather than only the encoder decoder type problems. A type of problem which KFAC conveniently appears to perform particularly well on. In addition the hyperparameter tuning process is completely free of human elements, since all parameters were selected by an impartial BO.
On a note on this study's results in direct comparison to the results found by the original paper. Although the underlying optimizers KFAC and SGD are the same, the employed approaches are fundamentally different in regard to a series of design choices. Including: training time, hyper parameter tuning, test loss/train loss, optimization tricks, regularization, momentum. A direct comparison would by no means be fair. Additionally both comparisons are in terms of mostly inspecting learning curves and do not come down to one number which could be held against each other. Nonetheless, in spite of all the differences, the conclusions are the same. KFAC outperforms SGD.

8.5.1 Limitations of the comparison

The results found in this study are deemed more adequate with regard to how KFAC compares to SGD in general, but they are by no means definite.

Although this study conducts the comparison on a multitude of problems and network architectures, these are still mostly toy problems and have next to non direct real world applications: I.e. there are no guarantee that the results interpolates to more complex use cases. Especially it should be kept in mind that the network architectures are quite simple and all fully connected. (As mentioned this comes down to the introduced theory only covering fully connected architectures). By today's standards fully connected networks are rarely used in real world applications due to more domain specific options being available. However fully connected layers are still widely present in modern architectures and these sections of the networks may very well be optimized using KFAC, though this might unnecessarily over complicate matters.

All findings are based on models which are trained for a maximum of 10 (5 for the regression) seconds. In the context of training neural networks this is considered a very short amount of time. The justification being that nothing beyond that time horizon can be concluded with certainty since the hyperparameters were specifically selected based on training of exactly that amount of time. Had the models trained for a long amount of time during BO another set of parameters might have been deemed optimal. Therefore this study does only concern itself with what happens within that exact time frame as it is the only time horizon on which anything can be stated with certainty. Ideally, and most likely, the results found under the given circumstances generalizes to

longer training sessions as well.
Clear and well-founded reasons for the design choice of only performing BO for 10 seconds per iteration are stated under the section textitBayesian optimization for hyper parameter tuning. Nonetheless comparison over a greater time horizon is an obvious candidate for future work.

This study only compares KFAC to one other optimization approach, namely SGD. The SGD baseline is given the best possible requisites to perform well. It has the addition of Nesterov momentum as well as getting a very comprehensive BO treatment to enable finding a good set of hyperparameters which may let it perform to its full capabilities. While SGD is a solid choice as a baseline, having KFAC beat it in 3 different scenarios does not even make KFAC state of the art in regard to those 3 exact problems. There are many choices when it comes to optimizers, and further investigation of KFAC as an optimization approach should involve comparing it to other popular choices.

The biggest limitation associated with the experimental work conducted and potentially the greatest candidate for future work, is the unintended exclusion of the bias term from the neural network architectures used in the study. As already explained this unfortunate incident has nothing to do with technical or theoretical limitations. In addition, the primary concern of this study is the relative performance between the optimization approaches, and the nature of the comparison between the approaches are still completely fair. Nonetheless the bias term plays an important role in the network architecture and there is no valid reason not to include it. Furthermore it could be theorised that the unfortunate exclusion favors KFAC. The argument being that bias would normally help to accommodate for issues associated with difference scale of the data. However second order methods are inherently invariant to issues related to scaling and simple transformations, thus making the exclusion of the bias a relatively smaller problem for KFAC.

8.5.2 Is KFAC recommendable?

SGD as well as other popular optimizers have a clear convenience factor in having already been implemented in popular neural network frameworks such as Pytorch and Tensorflow. For the sake of argument this may be disregarded, as there are no immediate limitations restricting these frameworks to cover KFAC.
For “simple” machine learning problems similar to those utilized in the study, KFAC seems like a very viable optimization choice. The less positive implications being that you limit yourself to fully connected networks (only as of this implementation). Big batch sizes become important for optimal efficiency. Which in turn may entrail GPU memory issues. However the proposed artificial batch size method is one approach to combat this issue, while only slightly impacting learning efficiency. In general memory should not be an issue in regard to the “simple” machine learning problems in question. The use of big batches may also lead to generalisation issues, but the extent to which this affects KFAC has not been investigated and the general advice is to closely monitor test performance.
On a more positive note KFAC should outperform finely tuned SGD with Nesterov momentum. KFAC seems to perform exceedingly well on encoder decoder problems. Lastly, if a multitude of networks needs to be trained either due to variation in the problem or the architecture, KFAC may be a particularly good choice. The reason being that KFAC seems to be rather consistent in regard to hyperparameter preference across problems thus potentially cutting down hyper parameter tuning time.

9 Conclusion

Through derivation the Fisher emerges as the approximate Kullback Leibler divergence between the output distribution of a statistical model and the output distribution of the same model with some infinitesimal change to its parameters. The Fisher appears as a metric which better accounts for the underlying structure of the model, and its inverse appears in the natural gradient as a correction term to the descent direction.

The use of natural gradient descent is made feasible through a series of approximations. It is argued that the sequential structure of the network enables a block diagonal approximation of the fisher. Furthermore it is shown how each Fisher block may be approximated by two smaller matrices through clever use of the Kronecker product and its associated identities. In particular it is exploiting that the inverse of a Kronecker product equals the Kronecker product of inverses.

KFAC is subsequently implemented from the ground up and it is found that KFAC has a strong preference for large batch sizes, and in fact is completely unable to learn if the batch size is too small. This study proposes a solution to the problem which involves artificially increasing the batch size, and enables learning in cases where KFAC was previously unable to.

Finally KFAC is compared to SGD with Nesterov momentum on an encoder-decoder regression and classification problem. It is found that KFAC outperforms SGD across all problems with a few reservations in regard to the classification comparison. Additionally it is indicated that KFAC inherently prefers similar hyperparameters across problems potentially reducing the time spend on tuning the model.

10 Bibliography

[1] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 12 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[2] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," pp. 503–528, 1989.

[3] N. L. Roux, P.-A. Manzagol, and Y. Bengio, "Topmoumoute online natural gradient algorithm."

[4] J. Martens, "Deep learning via hessian-free optimization," 2010.

[5] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximate curvature."

[6] R. Grosse and J. Martens, "A kronecker-factored approximate fisher matrix for convolution layers," 2 2016. [Online]. Available: http://arxiv.org/abs/1602.01407

[7] J. M. Deepmind, J. Ba, and M. J. G. Brain, "Kronecker-factored curvature approxima-tions for recurrent neural networks."

[8] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation." [Online]. Available: https://github.com/openai/baselines.

[9] G. Zhang, S. Sun, D. Duvenaud, and R. Grosse, "Noisy natural gradient as variational inference," 2018.

[10] T. George, C. Laurent, X. Bouthillier, N. Ballas, and P. Vincent, "Fast approximate natural gradient descent in a kronecker-factored eigenbasis."

[11] Z. Tang, F. Jiang, M. Gong, H. Li, Y. Wu, F. Yu, Z. Wang, and M. Wang, "Skfac: Training neural networks with faster kronecker-factored approximate curvature."

[12] J. Lee, H. G. Hong, D. Joo, and J. Kim, "Continual learning with extended kronecker-factored approximate curvature."

[13] J. Ba, R. Grosse, and J. Martens, "Distributed second-order optimization using kronecker-factored approximations."

[14] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," 9 2016. [Online]. Available: http://arxiv.org/abs/1609.04836

[15] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, R. Yokota, and S. Matsuoka, "Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks."

[16] "Lecture 3.5 natural gradient optimization (i) — neural networks — mlcv 2017 - youtube." [Online]. Available: https://www.youtube.com/watch?v=eio6l-Po83o

[17] "Natural gradients - basics of modern image analysis - youtube." [Online]. Available: https://www.youtube.com/watch?v=oNWvg2CaMfI

[18] S.-I. Amari, "Natural gradient works efficiently in learning."

[19] G. Desjardins, K. Simonyan, R. Pascanu, and K. Kavukcuoglu, "Natural neural networks," 7 2015. [Online]. Available: http://arxiv.org/abs/1507.00210

[20] "Riemannian manifold - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Riemannian_manifold

[21] "Metric tensor - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Metric_tensor

[22] "Appendices for the icml paper "optimizing neural networks with kronecker-factored approximate curvature"."

[23] "2nd-order optimization for neural network training - youtube." [Online]. Available: https://www.youtube.com/watch?v=qAVZd6dHxPA&t=300s

[24] "Kronecker product - wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Kronecker_product

[25] M. Pourahmadi, "Covariance estimation: The glm and regularization perspectives," *Statistical Science*, vol. 26, pp. 369–387, 2011.

[26] M. W. Klein, C. Enkrich, M. Wegener, and S. Linden, "Reducing the dimensionality ofdata with neural networks," *Science*, vol. 313, pp. 502–504, 7 2006.

[27] "Mnist handwritten digit database, yann lecun, corinna cortes and chris burges." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[28] "Uci machine learning repository: Relative location of ct slices on axial axis data set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Relative+location+of+CT+slices+on+axial+axis

[29] "Cifar-10 and cifar-100 datasets." [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[30] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks." [Online]. Available: http://www.iro.umontreal.

[31] "Understanding nesterov momentum (nag) - dominik schmidt." [Online]. Available: https://dominikschmidt.xyz/nesterov-momentum/

[32] "A 2021 guide to improving cnns-optimizers: Adam vs sgd — by sieun park — geek culture — medium." [Online]. Available: https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008

[33] "Github - sheffieldml/gpyopt: Gaussian process optimization using gpy." [Online]. Available: https://github.com/SheffieldML/GPyOpt

11 Appendix

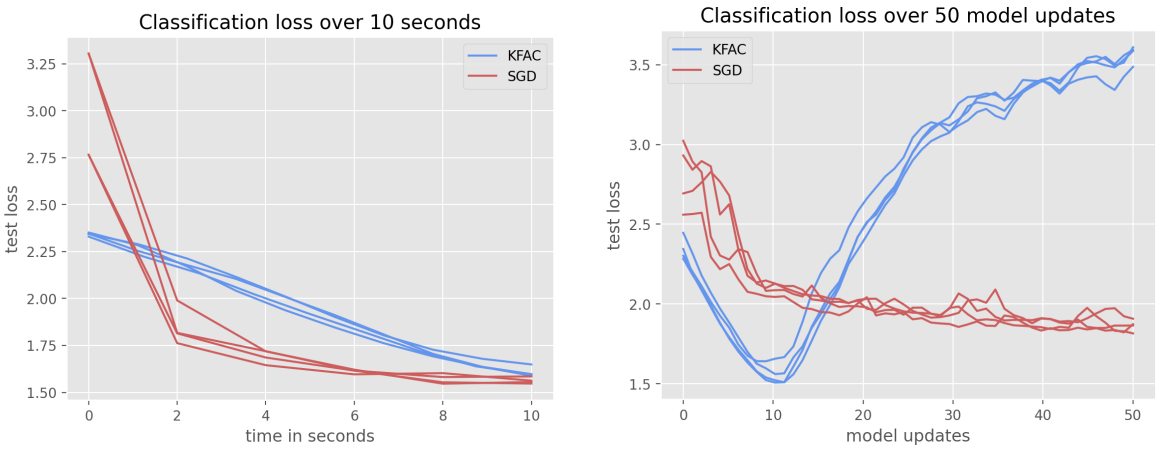

Figure 9: **Test loss** as a function of time and number of updates for the 4 best KFAC and SGD models on the classification problem.

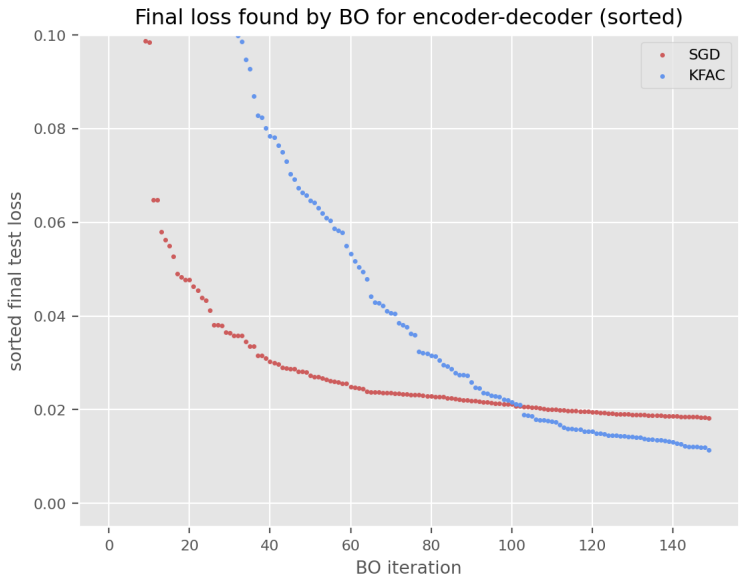

Figure 10: BO iterations in sorted order for the encoder decoder problem

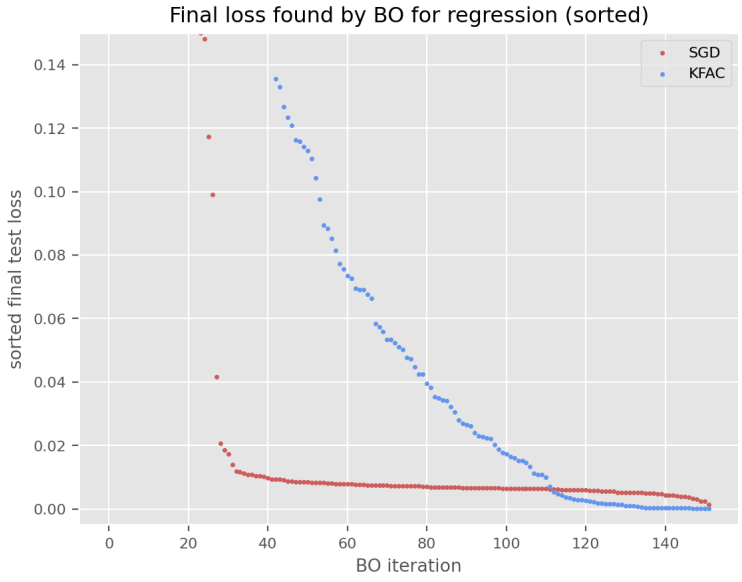

Figure 11: BO iterations in sorted order for the regression problem

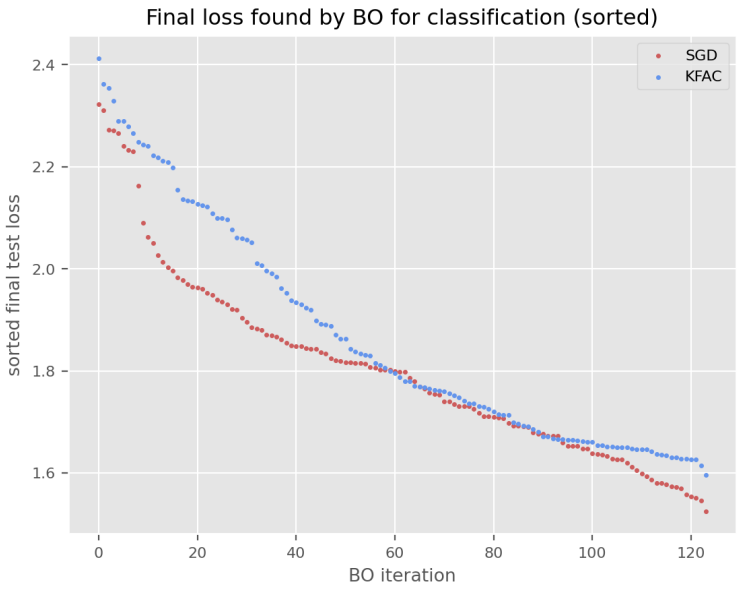

Figure 12: BO iterations in sorted order for the classification problem