DEEP CURVATURE SUITE

Anonymous authors
Paper under double-blind review

ABSTRACT

Despite providing rich information into neural networks geometry aiding in their understanding and applications in second order optimisation, accessing curvature information is still a daunting engineering challenge and hence inaccessible to most practitioners. In some cases, proxy sampled diagonal approximations, which we show on both real and synthetic examples can be arbitrarily bad, are employed instead. We hence provide an open-source software package, the Deep Curvature Suite to the community. Our goal is to enable easy access to curvature information for real networks and datasets, not just toy examples. Beyond the calculation of a highly accurate moment matched approximation of the Hessian spectrum using the Lanczos algorithm, our package provides extensive loss surface visualisation, the calculation of the gradient/Hessian variance and includes second order deep learning optimisers. As a further contribution, we address and disprove many common misconceptions in the Machine Learning literature, namely that the Lanczos algorithm learns eigenvalues from the top down. We also prove using high dimensional concentration inequalities that for specific classes of matrices a single random vector is sufficient for accurate spectral estimation, which informs our Algorithm design choice and spectral visualisation method. We showcase our package practical utility on a series of examples based on realistic modern neural networks tested on CIFAR-10/100 datasets.

1 INTRODUCTION

The success of Deep Neural Networks (DNNs) trained with stochastic gradient descent (SGD) based optimizers on a range of tasks, has led to an explosion in the availability of easy to use out of the box high performance software implementations. Automatic differentiation packages such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017) have become widely adopted, with higher level packages allowing users to state their model, dataset and optimiser in a few lines of code (Chollet, 2015), effortlessly achieving state of the art performance. However, the pace of development of software extracting second order information, representing the curvature at a point in weight space, has not kept abreast. Researchers aspiring to evaluate or use curvature information need to implement their own libraries, which are rarely shared or kept up to date. Naive implementations are computationally intractable for all but the smallest of models. Hence, researchers typically either completely ignore curvature information or use highly optimistic approximations, such as the diagonal elements of the matrix or of a surrogate matrix, with limited justification or empirical analysis of the harshness of the aforementioned approximations Chaudhari et al. (2016); Dangel et al. (2019) The curvature at a point in weight-space informs us about the local conditioning of the problem (i.e the ratio of the largest to smallest eigenvalues $\frac{\lambda_1}{\lambda_P}$). This determines the rate of convergence for first order methods and informs us about the optimal learning and momentum rates (Nesterov, 2013). The most common areas where curvature information is employed are analyses of the **Loss Surface** and **Newton** type methods in optimization.

Loss surface visualization of deep neural networks have often focused on two dimensional slices of random vectors (Li et al., 2017) or the changes in the loss traversing a set of random vectors drawn from the d-dimensional Gaussian distribution (Izmailov et al., 2018). Whilst both of these approaches have been informative, it is not clear that these high dimensional loss surfaces, often containing millions or billions of dimensions, can be well captured in this manner. Recent empirical analyses of the neural network loss surfaces invoking full eigen-decomposition (which captures the full dimensionality of the surface at a particular point in weight space) (Sagun et al., 2016; 2017)

have been limited to toy examples with < 5000 parameters. Other works have used the diagonal of the Fisher information matrix (Chaudhari et al., 2016), an assumption we will challenge in this paper. From a practical perspective, specific properties of the loss surface not captured by the aforementioned approaches, such as the *flatness* such as the trace, Frobenius and spectral norm have been used to characterise the generalisation of a solution found by SGD (Wu et al., 2018; Izmailov et al., 2018; He et al., 2019; Jastrzkebski et al., 2017; 2018; Keskar et al., 2016). Under a Bayesian and minimum description length argument (Hochreiter and Schmidhuber, 1997) flatter minima should generalise better than sharper minima. These properties are extremely easy in principle to estimate, at a computational cost of a small multiple of gradient evaluations. However the calculation of these properties are not typically included in standard Deep Learning frameworks, which limits the ability of researchers to undertake such analysis.

From a theoretical stand point the full spectrum of Deep Neural Networks could in principle be used to validate/invalidate novel theoretical contributions. Analysis relating the loss surface to spin-glass models from condensed matter physics and random matrix theory (Choromanska et al., 2015b;a) rely on a number of unrealistic assumptions, such as input independence. Since such an assumption is clearly not true, one potential verification of the practical applicability of these results would be to visualise the spectra of large real networks and commonly used datasets to evaluate whether they match those of random matrices. Such matrices have closed analytical formulas describing their spectra (Tao, 2012; Akemann et al., 2011) and statistical tests comparing the two could be undertaken .

Other important areas of loss surface investigation include understanding the effectiveness of batch normalization (Ioffe and Szegedy, 2015). Recent convergence proofs (Santurkar et al., 2018) bound the maximal eigenvalue of the Hessian with respect to the activations and bounds with respect to the weights on a per layer basis. Bounds on a per layer basis do not imply anything about the bounds of the entire Hessian and furthermore it has been argued that the full spectrum must be calculated to give insights on the alteration of the landscape (Kohler et al., 2018).

Second Order Optimisation Methods solve the minimisation problem for the loss, L associated with parameters $\boldsymbol{p}$ and perturbation $\boldsymbol{d}$ to the second order in Taylor expansion,

$$\boldsymbol{d}^* = \text{argmin}_{\boldsymbol{d}} L(\boldsymbol{p}+\boldsymbol{d}) = \text{argmin}_{\boldsymbol{d}} L(\boldsymbol{p}+\boldsymbol{d}) = \text{argmin}_{\boldsymbol{d}} L(\boldsymbol{p}) + \nabla L^T \boldsymbol{d} + \tfrac{1}{2}\boldsymbol{d}^T \bar{\boldsymbol{H}} \boldsymbol{d} \tag{1}$$

Where instead of the Hessian $\boldsymbol{H} = \nabla\nabla L(\boldsymbol{p}) \in \mathbb{R}^{n\times n}$, a surrogate positive definite approximation to the Hessian $\bar{\boldsymbol{H}}$, such as the (Generalised) Gauss-Newton (Martens, 2010; Martens and Sutskever, 2012), is employed so to make sure the minimum is lower bounded; its solution is

$$\boldsymbol{d} = -\bar{\boldsymbol{H}}^{-1}\nabla L(\boldsymbol{p}) = -\sum_i^N \frac{1}{\lambda_i}\boldsymbol{u}_i \boldsymbol{u}_i^T \nabla L(\boldsymbol{p}) \tag{2}$$

where $\boldsymbol{u}_i$ correspond to the generalised Hessian eigenvectors. The parameters are updated with $\boldsymbol{p} = \boldsymbol{p} - \alpha \boldsymbol{d}$, in which α is the global learning rate. For some common activations and loss functions typical in deep learning, such as the cross-entropy loss and sigmoid activation the generalised Gauss-Newton is equivalent to the Fisher information matrix (Pascanu and Bengio, 2013). The Hessian may be expressed in terms of the activation σ at the output of the final layer $f(\boldsymbol{x})$ using the chain rule as

$$\boldsymbol{H}_{batch}(\boldsymbol{w})_{ij} = \sum_{b=1}^{N}\left(\sum_{k=0}^{d_y}\sum_{l=0}^{d_y}\frac{\partial^2\sigma}{\partial f_l(\boldsymbol{x})\partial f_k(\boldsymbol{x})}(f(\boldsymbol{x}))\frac{\partial f_l(\boldsymbol{x})}{\partial x_j}\frac{\partial f_k(\boldsymbol{x})}{\partial x_i} + \sum_{k=0}^{d_y}\frac{\partial\sigma}{\partial x_j}(f(\boldsymbol{x}))\frac{\partial^2 f_k(\boldsymbol{x})}{\partial x_j \partial x_i}\right)/N \tag{3}$$

The first term on the LHS of Equation 3 is known as the *Generalised Gauss-Newton* matrix. Despite the success of second order optimisation methods using the Generalised Gauss Newton for difficult problems on which SGD is known to stall, such as recurrent neural networks (Martens and Sutskever, 2012), or auto-encoders (Martens, 2016). Researchers wanting to implement second order methods such as (Vinyals and Povey, 2012; Martens and Sutskever, 2012; Dauphin et al., 2014) face the aforementioned problems of difficult implementation. As a small side note, Bayesian neural networks using the Laplace approximation feature the Hessian inverse multiplied by a vector (Bishop, 2006).

1.1 CONTRIBUTIONS

In this paper, we make calculating and visualizing curvature information as simple as calculating the gradient at a saved checkpoint in weight-space. Specifically we combine fast Hessian vector products (Pearlmutter, 1994), advanced linear algebraic techniques (Golub and Meurant, 1994) and high dimensional geometry (Hutchinson, 1990), to derive highly accurate moment matched discrete approximations to the neural network spectrum in a fraction of the network training time. The computational complexity of our approach is $\mathcal{O}(mP)$, where m is the number of moments which we wish to match, P is the number of model parameters, as opposed to full exact eigendecomposition which has a numerical cost of $\mathcal{O}(P^3)$, which is infeasible for large neural networks. We use the GPytorch implementation (Gardner et al., 2018) of the *Lanczos algorithm* (Meurant and Strakoš, 2006), which we introduce in Section 4 and discuss the most common misconceptions in the literature in Section 4.

Our package supports the Hessian as well as the commonly used Generalsed Gauss Newton approximation (Martens, 2010; Martens and Sutskever, 2012; Henriques et al., 2019). Our code allows for the use of batch normalization (Ioffe and Szegedy, 2015) in neural networks. Whilst we keep batch norm training mode on as default, this default setting can be altered easily. We provide additional novel theoretically justified visualisations for both the moment matched spectrum and the loss surface traversed in those directions. Our package further allows for the calculation of the variance of the gradient and the variance of the Hessian, the latter has been related to the stability of SGD in Wu et al. (2018). As a minor contribution, we also include two stochastic Lanczos based optimisers in our code.

Related Work: Recent work making curvature information more available, again through diagonal approximations, explicitly disallows the use of batch normalization (Dangel et al., 2019). Our software package extends seamlessly to batch normalization, allowing for evaluation in both train and eval mode. Whilst the scope of Dangel et al. (2019) is slightly different to our work, as their focus is primarily on the ability to compute these quantities at speed online for simple models, in Section 5 we explicitly compare their diagonal MC approximations against our method for spectral visualisation on a range of synthetic and real neural networks and find these approximations to be inadequate for spectral analysis. Independent work has also considered using the Lanczos algorithm for Hessian computation Yao et al. (2019); Ghorbani et al. (2019); Papyan (2018); Izmailov et al. (2019). However neither of these packages includes a similar level of focus on the visualisation aspect which we integrate. Furthermore to the best of our knowledge, the computation of the Hessian variance Wu et al. (2018) and in-built reference stochastic second order optimisers is not available in any other package.

2 DEEP CURVATURE

We introduce to our package, the **Deep Curvature suite**, a software package that allows analysis and visualisation of deep neural network curvature. The main features and functionalities of our package are:

- **Network training and evaluation** we provide a range of pre-built modern popular neural network structures, such as VGG and variants of ResNets, and various optimisation schemes in addition to the ones already present in the PyTorch frameworks, such as K-FAC and SWATS. These facilitates faster training and evaluation of the networks (although it is worth noting that any PyTorch-compatible optimisers or architectures can be easily integrated into our analysis framework).

- **Eigenspectrum analysis of the curvature matrices** Powered by the Lanczos techniques implemented in GPyTorch (Gardner et al., 2018) and outlined in Section 3, *with a single random vector* we use the Pearlmutter matrix-vector product trick for fast inference of the eigenvalues and eigenvectors of the common curvature matrices of the deep neural networks. In addition to the standard Hessian matrix, we also include the feature for infer-

ence of the eigen-information of the Generalised Gauss-Newton matrix, a commonly used positive-definite surrogate to Hessian[1].

- **Advanced Statistics of Networks** In addition to the commonly used statistics to evaluate network training and performance such as the training and testing losses and accuracy, we support computations of more advanced statistics: For example, we support squared mean and variance of gradients and Hessians (and GGN), squared norms of Hessian and GGN, L2 and L-inf norms of the network weights and etc. These statistics are useful and relevant for a wide range of purposes such as the designs of second-order optimisers and network architecture.
- **Visualisations** For all main features above, we include accompanying visualisation tools. In addition, with the eigen-information obtained, we also feature visualisations of the **loss landscape** by studying the sensitivity of the neural network to perturbations of weights. While similar tools have been available, we would like to emphasise that one key difference is that, instead of the *random* directions as featured in some other packages, we explicitly perturb the weights in the *eigenvector* directions, which should yield more informative results.

Package Structure The main interface functions are organsed as followed:

- **./core** The functions under `core` directories are the main analysis tools of the package. `train_network` allows network training and saving of the required statistics for subsequent spectrum learning. Based on the output of it, we additionally include tools for spectrum analysis (`compute_eigenspectrum`) and advanced loss statistics (such as covariance of gradients and second order information like Hessian variance) in `compute_loss_stats` and `build_loss_landscape`.

  We provide some pre-built network architectures (such as VGG and ResNet architectures) and optimizers apart from PyTorch natives (such as K-FAC, SWATS optimizers). We additionally support Stochastic Weight Averaging proposed in (Izmailov et al., 2018). However, it is worth noting that any PyTorch compatible networks and optimizers can be easily integrated in our framework.
- **./visualise** This directory defines the various pre-defined visualisation functions for different purposes, including the visualisation of training, spectrum and the loss landscape.

To facilitate a quick start of our package, we have included an illustrated example of analysis on the VGG-16 network on CIFAR-100 dataset.

3 An Illustrated Example

We give an illustration on an example of using the MLRG-DeepCurvature package and more details, including detailed documentation of each user function and the output of this particular example, can be found at our open-source repository. We begin by importing the necessary functions and packages:

```
from core import *
from visualise import *
import matplotlib.pyplot as plt
```

In this example, we train a VGG16 network on CIFAR-100 for 100 epochs. In a test computer with AMD Ryzen 3700X CPU and NVIDIA GeForce RTX 2080 Ti GPU, each epoch of training takes less than 10 seconds.

This step generates a number of training statistics files (starting with `stats-`) and checkpoint files (`checkpoint-00XXX.pt`, where `XXX` is the epoch number) that contain the `state_dict` of the optimizer and the model. We may additionally visualise the training processes by looking at the

[1]The computation of the GGN-vector product is similar with the computational cost of two backward passes in the network. Also, GGN uses *forward-mode automatic differentiation* (FMAD) in addition to the commonly employed *backward-mode automatic differentiation* (RMAD). In the current PyTorch framework, the FMAD operation can be achieved using two equivalent RMAD operations.

Network Training

```
train_network(
dir='result/VGG16-CIFAR100/',
dataset='CIFAR100',
data_path='data/',
epochs=100,
model='VGG16',
optimizer='SGD',
optimizer_kwargs={
'lr': 0.03,
'momentum': 0.9,
'weight_decay': 5e-4})
```

Eigenspectrum Computation

```
compute_eigenspectrum(
dataset='CIFAR100',
data_path='data/',
model='VGG16',
checkpoint_path='result/VGG16-CIFAR100/
    checkpoint-00100.pt',
save_spectrum_path='result/VGG16-CIFAR100/
    spectra/spectrum-00100-ggn_lanczos',
save_eigvec=True,
lanczos_iters=20,
curvature_matrix='ggn_lanczos',)
```

Table 1: Training a Neural Network and Calculating the Eigenspectrum using the Deep Curvature Suite package

Eigenspectrum Visualization

```
plot_spectrum('lanczos', path='
    result/VGG16-CIFAR100/spectra/
    spectrum-00100-ggn_lanczos.npz')
plt.show()
```

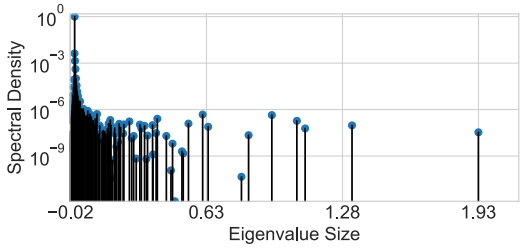


Table 2: Eigenspectrum plotting code and corresponding stem plot

basic statistics by calling `plot_training` function. With the checkpoints generated, we may now compute analyse the eigenspectrum of the curvature matrix evaluated at the desired point of training. For example, if we would like to evaluate the Hessian or Generalised Gauss Newton approximation to the Hessian matrix at the end of the training with 20 Lanczos iterations, we simply run the Eigenspetrum Computation code given in Table 1. This function call saves the spectrum results (including eigenvalues, eigenvectors and other related statistics) in the `save_spectrum_path` path string defined. To visualise the spectrum as stem plot similar to Figure 2, we simply call:

Finally, with the eigenvalues and eigenvectors computed, we might be interested in knowing how sensitive the network is to perturbation along these directions. To achieve this, we first construct a loss landscape by setting the number of query points and maximum perturbation to apply. To achieve that, we call the code given in Table 3. In this example, we set the maximum perturbation to be 1 (`dist` argument) and number of query points along each direction to be 21 (`n_points` argument). The corresponding plots in Training and Testing loss are also shown.

Loss Surface Visualization

```
build_loss_landscape(
dataset='CIFAR100',
data_path='data/',
model='VGG16',
dist=1., n_points=21,
spectrum_path='spectrum
    -100-hessian',
checkpoint_path='
    checkpoint-100.pt',
save_path='landscape-100.
    npz'
plot_loss_landscape('
    landscape-100.npz')
plt.show())
```

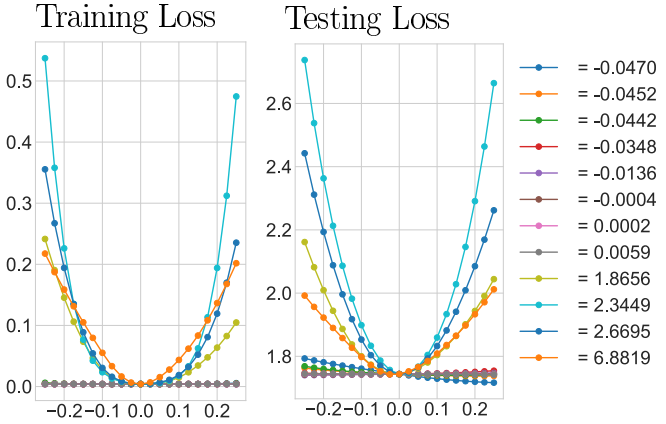


Table 3: Loss surface visualization along the sharpest Hessian eigenvectors with the Deep Curvature Suite

3.1 Simplicity of Second Order Optimisation

To enable researchers to experiment with second order optimisation algorithms for Deep Neural Networks, we implement a Lanczos based optimiser (which takes the absolute Hessian Dauphin et al.

SGD Training

```
train_network(
dir='VGG16-CIFAR100/',
dataset='CIFAR100',
data_path='data/',epochs=100,
model='VGG16', optimizer='SGD',
optimizer_kwargs={
'lr': 0.01,
'momentum': 0.9,
'weight_decay': 0
'batch_size': 128})
```

Lanczos 2nd Order Training

```
train_network(
dir='VGG16-CIFAR100/',
dataset='CIFAR100',
data_path='data/',epochs=100,
model='VGG16', optimizer='LancGN',
optimizer_kwargs={
'lr': 1,
'damping': 10,
'weight_decay': 0
'batch_size': 128
'curvature_batch_size': 128})
```

Table 4: Comparison of SGD training and Second Order Optimisation using the Deep Curvature Suite

(2014), or Generalised Gauss Newton as input). The code for running this optimiser is summarised in Table 4. We plot the training error of the VGG-16 network on CIFAR-100 dataset (Which has $16m$ parameters) against epoch in Figure 1. We keep the ratio of damping constant to learning rate constant, where $\delta = 10\alpha$, for a variety of learning rates in $\{1, 0.1, 0.01, 0.001, 0.0001\}$ with a batch size of 128 for both the gradient and the curvature, all of which post almost identical performance. We also compare against different learning rates of Adam and the best grid searched learning rate of SGD, both of which converge significantly slower per iteration compared to our stochastic Newton methods.

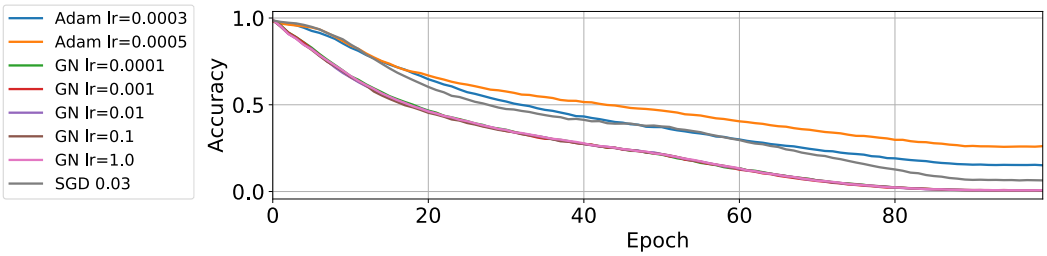


Figure 1: Training Error of stochastic Lanczos Newton methods on CIFAR-100 VGG-16 against baselines.

4 Learning to Love Lanczos

The Lanczos Algorithm, on which our package is based, (Algorithm 1) is an iterative algorithm for learning a subset of the eigenvalues/eigenvectors of any Hermitian matrix, requiring only matrix vector products. It can be regarded as a far superior adaptation of the power iteration method, where the Krylov subspace $\mathscr{K}(\boldsymbol{H}, \boldsymbol{v}) = \text{span}\{\boldsymbol{v}, \boldsymbol{H}^{\boldsymbol{v}}, \boldsymbol{H}^2\boldsymbol{v}...\}$ is orthogonalised using Gram-Schmidt. Beyond having improved convergence to the power iteration method (Bai et al., 1996) by storing the intermediate orthogonal vectors in the corresponding Krylov subspace, Lanczos produces estimates of the eigenvectors and eigenvalues of smaller absolute magnitude, known as Ritz vectors/values. Despite its known superiority to the power iteration method and relationship to orthogonal polynomials and hence when combined with random vectors the ability to estimate the entire spectrum of a matrix, these properties are often ignored or forgotten by practitioners, we hence include a full primer in Appendix E. We compare against diaognal approximations for random and synthetic matrices in Appendix C. We also explicitly debunk some key persistent myths, given below.

- We can learn the negative and interior eigenvalues by shifting and inverting the matrix sign $\boldsymbol{H} \rightarrow -\boldsymbol{H} + \mu\boldsymbol{I}$
- Lanczos learns the largest m $[\lambda_i, \boldsymbol{u}_i]$ pairs of $\boldsymbol{H} \in \mathbb{R}^{P \times P}$ with high probability (Dauphin et al., 2014)

Since these two related beliefs are prevalent, we disprove them explicitly in this section, with Theorems 1 and 2.

Theorem 1. *The shift and invert procedure $\boldsymbol{H} \rightarrow -\boldsymbol{H} + \mu\boldsymbol{I}$, changes the Eigenvalues of the Tridiagonal matrix $\boldsymbol{T}$ (and hence the Ritz values) to $\lambda_i = -\lambda_i + \mu$*

Proof. Following the equations from Algorithm 1

$$\begin{aligned}
&\boldsymbol{w}_1^T = (-\boldsymbol{H} + \mu\boldsymbol{I})\boldsymbol{v}_1 \ \& \ \alpha_1 = \boldsymbol{v}_1^T\boldsymbol{H}\boldsymbol{v}_1 + \mu\boldsymbol{I} \\
&\boldsymbol{w}_2 = \boldsymbol{w}_1 - \alpha_1\boldsymbol{v}_1 = (\boldsymbol{H} + \mu\boldsymbol{I})\boldsymbol{v}_1 - (\boldsymbol{v}_1^T\boldsymbol{H}\boldsymbol{v}_1 + \mu\boldsymbol{I})\boldsymbol{v}_1 \\
&\boldsymbol{w}_2 = (\boldsymbol{H} - \boldsymbol{v}_1^T\boldsymbol{H}\boldsymbol{v}_1)\boldsymbol{v}_1 \ \& \ \boldsymbol{v}_2 = \boldsymbol{w}_2/||\boldsymbol{w}_2|| \\
&\alpha_2 = \boldsymbol{v}_2^T(-\boldsymbol{H} + \mu\boldsymbol{I})\boldsymbol{v}_2 = -\boldsymbol{v}_2^T\boldsymbol{H}\boldsymbol{v}_2 + \mu \\
&\beta_2 = ||\boldsymbol{w}_2||
\end{aligned} \tag{4}$$

Assuming this for $m-1$, and repeating the above steps for m we prove by induction and finally arrive at the modified tridiagonal Lanczos matrix $\tilde{\boldsymbol{T}}$

$$\begin{aligned}
&\tilde{\boldsymbol{T}} = -\boldsymbol{T} + \mu\boldsymbol{I} \\
&\tilde{\lambda_i} = -\lambda_i + \mu \ \forall 1 \leq i \leq m
\end{aligned} \tag{5}$$

Remark. No new Eigenvalues of the matrix $\boldsymbol{H}$ are learned. Although it is clear that the addition of the identity does not change the Krylov subspace, such procedures are commonplace in code pertaining to papers attempting to find the *smallest eigenvalue*. This disproves the first misconception.

Theorem 2. *For any matrix $\boldsymbol{H} \in \mathbb{R}^{P\times P}$ such that $\lambda_1 > \lambda_2 > ..... > \lambda_P$ and $\sum_{i=1}^{m}\lambda_i < \sum_{i=m+1}^{P}\lambda_i$ in expectation over the set of random vectors $\boldsymbol{v}$ the m eigenvalues of the Lanczos Tridiagonal matrix $\boldsymbol{T}$ do not correspond to the top m eigenvalues of $\boldsymbol{H}$*

Proof. Let us consider the matrix $\tilde{\boldsymbol{H}} = \boldsymbol{H} - \frac{\lambda_{m+1}+\lambda_m}{2}\boldsymbol{I}$,

$$\begin{cases}
\lambda_i > 0, & \forall i \leq m \\
\lambda_i < 0, & \forall i > m
\end{cases} \tag{6}$$

Under the assumptions of the theorem, $\text{Tr}(\tilde{\boldsymbol{H}}) < 0$ and hence by Theorem 8 and Equation 24 there exist no $w_i > 0$ such that

$$\sum_{i=1}^{m} w_i\lambda_i^k = \frac{1}{P}\sum_{i=1}^{P}\lambda_i^k \forall \quad 1 \leq k \leq m \tag{7}$$

is satisfied for $k = 1$, as the LHS is manifestly positive and the RHS is negative. By Theorem 1 this holds for the original matrix $\boldsymbol{H}$. □

Remark. Given that Theorem 2 is satisfied over the expectation of the set of random vectors, which by the CLT is realised by Monte Carlo draws of random vectors as $d \to \infty$ the only way to really span the top m eigenvectors is to have selected a vector which lies in the m dimensional subspace of the P dimensional problem corresponding to those vectors, which would correspond to knowing those vectors *a priori*, defeating the point of using Lanczos at all.

Remark. Given the relationship between the moments of the spectral density and the expectation over the set of random vectors, one may be curious to ask how we expect the deviation to be depending on the number of random vectors actually used in practice. Whilst usually packages using Lanczos use a number of random vectors (increasing the corresponding computational cost), in Appendix A we demonstrate that we expect the difference to reduce as we increase the problem dimension P. This informs our choice of only using a single random vector in our package, we compare results for different random vectors in Appendix A and find minimal differences.

5 Neural Network Examples

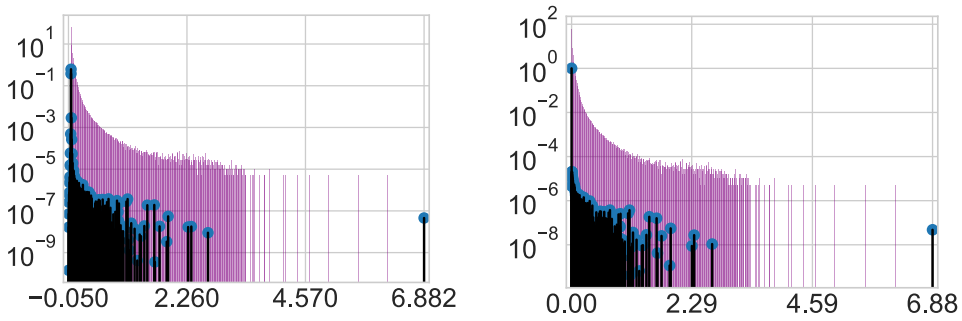
We showcase our spectral learning algorithm and visualization tool on real networks trained on real data-sets and we test on VGG networks (Simonyan and Zisserman, 2014). We train our neural networks using stochastic gradient descent with momentum $\rho = 0.9$, using a linearly decaying learning rate schedule. The learning rate at the t-th epoch is given by:

$$\alpha_t = \begin{cases}
\alpha_0, & \text{if } \frac{t}{T} \leq 0.5 \\
\alpha_0[1 - \frac{(1-r)(\frac{t}{T}-0.5)}{0.4}] & \text{if } 0.5 < \frac{t}{T} \leq 0.9 \\
\alpha_0 r, & \text{otherwise}
\end{cases} \tag{8}$$

where α_0 is the initial learning rate. $T = 300$ is the total number of epochs budgeted for all experiments. We set $r = 0.01$. We explicitly give an example code run in F We compare our method against recently developed open-source tools which calculate on the fly diagonal Hessian and Generalised Gauss-Newton diagonal approximations (Dangel et al., 2019).

5.1 VGG-16 CIFAR-100 DATASET

We train a 16-layer VGG network, comprising of $P = 15,291,300$ parameters on the CIFAR-100 dataset, using $\alpha_0 = 0.05$. Even for this relatively small model, the open-source Hessian and GGN exact diagonal computations require over 125GB of GPU memory and so to avoid re-implementing the library to support multiple GPUs and node communication we use the Monte Carlo approximation to the GGN diagonal against both our GGN-Lanczos and Hessian-Lanczos spectral visualizations. We plot a histogram of the Monte Carlo approximation of the diagonal GGN (Diag-GGN) against both the Lanczos GGN (Lanc-GGN) and Lanczos Hessian (Lanc-Hess) in Figure 2. Note that as the Lanc-GGN and Lanc-Hess are displayed as stem plots (with the discrete spectral density summing to 1 as opposed to the histogram area summing to 1).



(a) Diag-GGN & Lanc-GGN

(b) Diag-GGN & Lanc-Hess

Figure 2: Diagonal Generalised Gauss Newton monte carlo approximation (Diag-GGN) against $m = 100$ Lanczos using Gauss-Newton vector products (Lanc-GGN) or Hessian vector products (Lanc-Hess)

We note that the Gauss-Newton approximation quite closely resembles its Hessian counterpart, capturing the majority of the bulk and the outlier eigenvectors at $\lambda_1 \approx 6.88$ and the triad near $\lambda_i \approx 2.29$. The Hessian does still have significant spectral mass on the negative axis, around 37%. However most of this is captured by a Ritz value at -0.0003, with this removed, the negative spectral mass is only 0.05%. However the Diag-GGN gives a very poor spectral approximation. It vastly overestimates the bulk region, which extends well beyond $\lambda \approx 1$ implied by Lanczos and adds many spurious outliers between 3 and the misses the largest outlier of 6.88. *Computational Cost* Using a single NVIDIA GeForce GTX 1080 Ti GPU, the Gauss-Newton takes an average 26.5 seconds for each Lanczos iteration with the memory useage 2850Mb. Using the Hessian takes an average of 27.9 seconds for each Lanczos iteration with 2450Mb memory usage.

6 CONCLUSION

We introduce the **Deep Curvature** suite in **PyTorch** framework, based on the Lanczos algorithm implemented in GPyTorch (Gardner et al., 2018), that allows deep learning practitioners to learn spectral density information as well as eigenvalue/eigenvector pairs of the curvature matrices at specific points in weight space. Together with the software, we also include a succinct summary of the linear algebra, iterative method theory including proofs of convergence and misconceptions and stochastic trace estimation that form the theoretical underpinnings of our work. Finally, we also included various examples of our package of analysis of both synthetic data and real data with modern neural network architectures.