Epistemic Uncertainty Quantification in Deep Learning by the Delta Method

Geir Kjetil Nilsen

Thesis for the degree of Philosophiae Doctor (PhD)
University of Bergen, Norway
2022

UNIVERSITY OF BERGEN

Epistemic Uncertainty Quantification in Deep Learning by the Delta Method

Geir Kjetil Nilsen

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Date of defense: 16.05.2022

Year: 2022

Title: Epistemic Uncertainty Quantification in Deep Learning by the Delta Method

Name: Geir Kjetil Nilsen

Print: Skipnes Kommunikasjon / University of Bergen

Scientific Environment

This study was carried out at, and funded by, the Department of Mathematics, University of Bergen. Parts of the work have been done in the context of CEDAS (Center for Data Science, University of Bergen, Norway). Additional funding for a research visit to the Technical University of Denmark, was granted by the Meltzer Fund.

Advisory committee

- Antonella Z. Munthe-Kaas (University of Bergen, Department of Mathematics)
- Hans J. Skaug (University of Bergen, Department of Mathematics)
- Morten Brun (University of Bergen, Department of Mathematics)

Acknowledgments

This thesis is in memory of Andreas Våge Aspevik. I dedicate this work to you. I would like to thank Prof. Antonella Z. Munthe-Kaas, Prof. Hans J. Skaug and Assoc. Prof. Morten Brun for your excellent advisory. Thanks to our frequent meetings, your instant feedback and all our good talks — my three years at the Department of Mathematics, University of Bergen, has served me an invaluable experience. Also thanks to Dr. Berent Å. S. Lunde, Øyvind Lunde Rørtveit, Dr. Kristian Gundersen and Egil[1] Martinsen Aspevik for your helpful advice on various technical issues examined.

[1]1337

Nomenclature

Symbols

$a(\cdot)$	Neural network activation function
$C(\cdot)$	Cost function defining the objective of neural network training / optimization procedure; scalar function defining the P-dimensional cost landscape
$f(\cdot)$	Model function defining neural network architecture, mapping from T_1-dimensional space to T_L-dimensional space
F	Sensitivity matrix of size $T_L \times P$; Jacobian matrix of neural network outputs for a given inpu
G	Hessian OPG approximation matrix of size $P \times P$
H	Hessian matrix of size $P \times P$
K	Number of eigenpairs, scalar
L	Number of neural network layers, scalar
n	When used as subscript, example index
N	Number of training examples, scalar
P	Number of neural network parameters, scalar
Q	Eigenvector matrix of size $P \times K$ or $P \times P$
S	Number of Lanczos iteration steps, scalar
T	When used as superscript (e.g. Q^T), indicates transposed
T_1	Dimensionality of neural network input and data, scalar
T_L	Dimensionality of neural network output and data, scalar
x_n	T_1-dimensional input data point / example, vector or scalar
x_0	Denotes an arbitrary T_1-dimensional input data point / example, vector or scalar
y_n	T_L-dimensional output data point / example, vector or scalar
$\hat{y}_n$	T_L-dimensional prediction / estimated output data point, vector or scalar

λ	L_2-regularization rate parameter, scalar
ω	P-dimensional model parameter, vector or scalar
$\hat{\omega}$	P-dimensional estimated model parameter, vector or scalar
$\sigma(x_0)$	The T_L-dimensional predictive uncertainty associated with x_0 (i.e. one uncertainty value per neural network output)
Λ	Eigenvalue matrix of size $K \times K$
λ_k	Eigenvalue number k, $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_k \geq \ldots \geq \lambda_K$, scalar
Σ	Covariance matrix of size $P \times P$
$\wedge$	When used as superscript (e.g. $\hat{\omega}$), indicates estimated
$\sim$	When used as superscript (e.g. $\widetilde{\Sigma}$), indicates approximated
$\mathcal{N}$	Normal distribution

Abbreviations / Accronyms

CIFAR-10	Canadian Institute For Advanced Research; Small images dataset, 10 classes
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DRWI	Dynamic Random Weight Initialization
GB	Giga Byte
Github	Provider of Internet hosting for software development and version control
GNU	General Public License
GPU	Graphical Processing Unit
ImageNet	Large visual database of more than 14 million hand-annotated images
LeNet	LeCun Network; type of neural network architecture
MNIST	Modified National Institute of Standards and Technology; Handwritten digits image dataset, 10 classes
NumPy	The fundamental package for scientific computing with Python
OoD	Out-of-Distribution
OPG	Outer-Product of Gradients
PreResNet	Pre-activation Residual Network; type of neural network architecture
Python	The Python programming language
PyTorch	Deep learning software framework provided by Facebook
ReLU	Rectified Linear Unit; type of activation function
ResNet	Residual Network; type of neural network architecture
RMS	Root Mean Square
SciPy	Python-based ecosystem of open-source software for mathematics, science, and engineering
SGD	Stochastic Gradient Descent
sigmoid	Type of activation function
softmax	Type of activation function
SRWI	Static Random Weight Initialization
Std	Standard deviation
SVD	Singular Value Decomposition
TensorFlow	Deep learning software framework provided by Google
Var	Variance

Preface

"**When** will my daughter be back from school today to ruin the concentration I need for great scientific accomplishments?", the poor computer scientist asked himself miserably blaming his pandemic induced home office situation.

"It's **Tuesday**, so I know it will be **quite early**", he continued. "On the other hand, it's **spring** — so it's nice to stay outside and play around with friends. But she's probably hungry — since there is **no afternoon meal** served at school **this week**".

The poor computer geek concluded it would rather be **soon**, and started to finish up his work in order to get a clean stop. A few seconds later, the doorbell rang, and she was home. His mental model had made the right prediction based on his prior knowledge and the data available.

Almost any process you can imagine can be thought of as function computation. As long as we can learn the right function for the given task, we can use it to predict answers to all kinds of relevant questions. Learning from data plays a key role not only in everyday life, but in many areas of science, finance and industry. We make use of historic data to learn functions, and then compute these for arbitrary inputs to predict likely outcomes. From the frequency spectra of underwater acoustic emissions, we can estimate the amount of oil leaking into the ocean. Or we can predict the future price of a stock and get rich on the basis of company performance measures and economic data [Friedman et al., 2001; Goodfellow et al., 2016; Nielsen, 2015].

However, the simplest mathematical functions can fall short of representing complicated real-world phenomena. If we consider simple linear regression, and learn the two parameters given a set of observations, this simple linear model can quickly fail to describe the real underlying function. The classical example is the XOR function, for which we will find that a linear model cannot represent its non-linear relationship between the input and the output. We can still use it, but its predictive capabilities will lead to poor accuracy because it is *under-fitting* the data.

In contrast, it would be convenient if we could just rank functions by how well they fit our data, and then select the model with the best fit. But here we face the unfortunate difficulty that more complex models typically fits the data better. When we fit a curve to data, a quadratic curve with three parameters can always fit the data better than a linear model with just two parameters. And it is turtles all the way down — because a polynomial with hundred terms fits the data even better, and so on. Selecting the model with the best fit typically leads to implausibly detailed and *over-fitted* models, which generalize poorly for new unseen data.

One of the most striking facts about *neural networks* comes with the *universal approximation theorem* [Stinchombe, 1989]. It states that no matter what the underlying function is, there is guaranteed to be a neural network so that for every possible input, the expected output value (or some close approximation) is predicted by the network. In fact, the universality theorem even holds for networks with a single hidden layer — as long as they are allowed to be exponentially wide. However, even though these *wide but shallow* networks can be powerful on their own, they are generally prone to over-fitting. But fortunately, strong empirical evidence suggests that *deep* neural networks [Goodfellow et al., 2016] yet pose great generalization capabilities. When cleverly constructed using various forms of regularization and parameter optimizations strategies — deep networks are capable of avoiding the over-fitting regime, and can be adapted to learn the functions useful in solving very complicated real-world problems with unprecedented accuracy.

Deep learning models are generally considered to be black box models. This means that their model functions are so complex that no human interpretability is possible. One cannot simply have a closer look at one or more of the parameters in these networks and assess their role in the over-all model. Nor can we reach a conclusion of what the prediction will look like for a given input just by simple inspection. Consequently, when we make a prediction, we can neither explain in an easy manner why the model decided to give us this particular answer. These aforementioned shortcomings are closely tied to the problem of adversarial attacks and out-of-distribution examples [Goodfellow et al., 2014; Lee et al., 2018]. Deep learning models generally work best when applied on input data coming from the same distribution as the training set. In terms of image classification, quite easily one can construct both questionable and unquestionable images that will fool the network to predict a desired outcome.

Statistical uncertainty [Friedman et al., 2001] has long been thought of as a potential candidate to put these aforementioned shortcomings to an end. Traditionally, deep learning image classifiers are constructed to produce probability point predictions. However, as the Bayesian paradigm provides a coherent framework based on full probability distributions, these networks can be extended to produce full multivariate probability distributions. Their outputs can thus be seen as probability means and probability variances. This additional piece of information, the variance, is thus seen as a measure of the associated *predictive uncertainty* of the probability point predictions represented by the means. Predictive uncertainty is often divided in two components: the *aleatoric* component and the *epistemic* component [Hüllermeier and Waegeman, 2020]. The aleatoric uncertainty is regarded as irreducible and stochastic, and will in the context of image classification stem from the randomness associated with the labeling process. The other component, the epistemic uncertainty — also referred to as model uncertainty — is due to limited amount of training data, and can be reduced accordingly.

With a mixed success, predictive uncertainty has previously been explored and approximated by a range of methods in the deep learning literature: from classical sampling techniques such as *Bootstrap ensemble* methods [Khosravi and Creighton, 2011; Osband, 2016], *Markov Chain Monte Carlo* based methods [Andrieu et al., 2003; Krauth, 1998], and the more recent *MC-Dropout* method [Gal and Ghahramani, 2016; Kendall and Gal, 2017], to optimization based methods commonly referred to as *Variational Inference* [Jordan et al., 1999; Wainwright and Jordan, 2008].

This thesis deals with one of the most classical uncertainty quantification methods known in statistics. Throughout the thesis, it will be referred to as the *Delta method* [Hoef, 2012; Khosravi and Creighton, 2011; MacKay, 1992a; Newey and McFadden, 1994]. The Delta method is an analytical approach towards quantifying the variance of multivariate functions. In the deep learning prediction context, this amounts to evaluating an expression for the variance (i.e. uncertainty) of the neural network output which depends on the inversion of the *Fisher information* [Ly et al., 2017] matrix. Since the dimensionality of this matrix grows quadratically with the number of model parameters, the method has a long time been deemed computationally intractable in the deep learning domain. However, from first principles — and as a result of three years of systematic work, including more than two hundred documented Jira tasks — we show via three central papers that the method can in fact be successfully adapted to quantify predictive epistemic uncertainty in modern deep learning image classification.

Abstract

This thesis explores the Delta method and its application to deep learning image classification. The Delta method is a classical procedure for quantifying uncertainty in statistical models, but its direct application to deep neural networks is prevented by the large number of parameters P. We recognize the Delta method as a measure of epistemic as opposed to aleatoric uncertainty and break it into two components: the eigenvalue spectrum of the inverse Fisher information (i.e. inverse Hessian) of the cost function and the per-example sensitivities (i.e. gradients) of the model function. We mainly focus on the computational aspects, and show how to efficiently compute low and full-rank approximations of the inverse Fisher information matrix, which in turn reduces the computational complexity of the naïve Delta method from $O(P^2)$ space and $O(P^3)$ time, to $O(P)$ space and time. We provide bounds for the approximation error by a novel error propagating technique, and validate the developed methodology with a released TensorFlow implementation. By a comparison with the classical Bootstrap, we show that there is a strong linear relationship between the quantified predictive epistemic uncertainty levels obtained from the two methods when applied on a few well known architectures using the MNIST and CIFAR-10 datasets.

The thesis is organized as follows: Chapter 1 contains an introduction to deep learning, while Chapter 2 addresses the basic concepts of uncertainty, and we briefly review some of the existing methodology for uncertainty quantification in deep learning. In Chapter 3, we give an introduction and a summary of the three scientific papers included in the dissertation. Chapter 4 concludes the thesis and discusses potential topics for future work. In the last Chapter 5, we include the original manuscripts of the three scientific papers.

List of Publications

Paper 1 [Nilsen et al., 2019]

Geir K. Nilsen, Antonella Z. Munthe-Kaas, Hans J. Skaug and Morten Brun, *Efficient Computation of Hessian Matrices in TensorFlow*, arXiv preprint: 1905.05559, 2019, revised 2021.

Paper 2 [Nilsen et al., 2021a]

Geir K. Nilsen, Antonella Z. Munthe-Kaas, Hans J. Skaug and Morten Brun, *Epistemic Uncertainty Quantification in Deep Learning Classification by the Delta Method*, **Published in Neural Networks (Elsevier) October 2021**.

Paper 3 [Nilsen et al., 2021b]

Geir K. Nilsen, Antonella Z. Munthe-Kaas, Hans J. Skaug, Morten Brun, *A Comparison of the Delta Method and the Bootstrap in Deep Learning Classification*, arXiv preprint: 2107.01606, 2021.

List of Released Software Code and Datasets

1. `pyHessian`: A TensorFlow module with functionality for efficient computation of Hessian matrices [Nilsen, 2018-2021b], https://github.com/gknilsen/pyhessian.git, Copyright (c) 2018-2021 by Geir K. Nilsen and the University of Bergen.

   (a) `pyhessian.py`: Main code for the technology described in **Paper 1** [Nilsen et al., 2019].

   (b) `pyhessian_example.py`: Demonstration on how to apply the developed technology described in **Paper 1** [Nilsen et al., 2019].

2. `pyDeepDelta`: A TensorFlow module implementing the Delta method in deep learning classification [Nilsen, 2018-2021a], https://github.com/gknilsen/pydeepdelta.git, Copyright (c) 2018-2021 by Geir K. Nilsen and the University of Bergen.

   (a) `pydeepdelta.py`: Main code for the developed Delta methodology described in **Paper 2** [Nilsen et al., 2021a].

   (b) `pydeepboot.py`: Code for the Bootstrap algorithm used in **Paper 3** [Nilsen et al., 2021b].

   (c) `pydeepdelta_demo.ipynb`: Application demo for a MNIST LeNet.

   (d) `pydeepdelta_demo_resnet18.ipynb`: Application demo for a CIFAR-10 ResNet18.

   (e) `pydeepdelta_sampler_demo.ipynb`: Demonstration on how to combine the standard Laplace approximation with the key ideas from **Paper 2** [Nilsen et al., 2021a] / An efficient Laplace Approximation based Monte Carlo sampling algorithm.

   (f) `pydeepdelta_predictive_uncertainty_datasets.ipynb`: Data sets with predictive uncertainty based features, procured using the developed Delta methodology described in **Paper 2** [Nilsen et al., 2021a] based on MNIST and CIFAR-10 image classifiers.

Contents

Chapter 1

Deep Learning

Behind the tremendous array of conglomerative literature, supervised *deep learning* [Goodfellow et al., 2016; Nielsen, 2015] can simply be viewed as a (huge) *non-linear regression.* At the first step towards supervised deep learning, we find *simple linear regression.* We learn a two-dimensional *parameter* describing a straight line on the basis of example data consisting of single-dimensional inputs and outputs. Consecutively, we form linear combinations of the parameter and new single-dimensional inputs to predict single-dimensional outputs. At the second step, we arrive at *multiple linear regression.* Here, a multi-dimensional parameter describing a hyper plane is learned to predict single-dimensional outputs from multi-dimensional inputs. The next step takes us to *logistic regression*, where we allow for learning non-linear relationships between two-dimensional outputs and multi-dimensional inputs. The fourth step takes us to *multinomial logistic regression*, where non-linear relationships between multi-dimensional inputs and outputs are allowed to be learned. At the final fifth step, we find *supervised deep learning* — a modeling paradigm with few rules — where we learn multiple *deep* layers of non-linear relationships between multi-dimensional inputs and outputs.

Deep learning has successfully been applied in both *unsupervised* [Caron et al., 2018; Jing and Tian, 2020; Xie et al., 2016] and *supervised* forms. This introduction deals with supervised deep learning, which is the task of learning a *parameterized* function that maps an input to an output based on example pairs of *both* input and output. Quite differently, unsupervised deep learning deals with the task of learning an embedding that groups inputs similar to each other into clusters based on *just* input examples.

1.1 Deep Learning Hype

Although successfully applied by word-wide researchers, deep learning has received substantial criticism from several parties. To quote Andrew Ng — artificial intelligence pioneer, co-founder and former head of Google Brain, former Chief Scientist at Baidu, and today mostly known as the leading figure of Coursera — in a podcast hosted by DeepLearning.AI and Stanford HAI, reproduced and published by the IEEE Spectrum Magazine [Perry, 2021]:

“It turns out”, Ng said, “that when we collect data from Stanford Hospital, then we train and test on data from the same hospital, indeed, we can publish papers showing [the algorithms] are comparable to human radiologists in spotting certain conditions.”

“But”, he said, “It turns out [that when] you take that same model, that same AI system, to an older hospital down the street, with an older machine, and the technician uses a slightly different imaging protocol, that data drifts to cause the performance of AI system to degrade significantly. In contrast, any human radiologist can walk down the street to the older hospital and do just fine.”

“So even though at a moment in time, on a specific dataset, we can show this works, the clinical reality is that these models still need a lot of work to reach production.”

Indeed, industries such as Medical and Automotive are currently investing billions of dollars in deep learning research. The hype seems to be endless, but few companies are willing to risk missing what potentially represents the beginning of the greatest technological revolution of all times. But the third[1] wave of the current reincarnation of neural networks also comes with its dark side: no one to date has yet been able to prove that the technology is fundamentally reliable, and this poses a big risk when companies start to roll out automated diagnostics systems and self-driving cars.

However, the fundamental problem with deep learning seems to be the so-called out-of-distribution examples. In a real-world project, one literally never knows how much training data you will have to collect and annotate in order to uncover the true underlying data generating distribution. In many deep learning applications, it can take an endless effort to cover all possible corner-cases.

In a lecture provided by Bouvet Norway AS, they identify several key points to why about 80% of industrial machine learning projects tend to fail; a) *too high expectations*: most of

[1]Historically, there have been three main waves of development in the field: deep learning known as Cybernetics in the 40s-60s, deep learning known as Connectionism in the 80s-90s, and the current resurgence under the name deep learning beginning in 2006 [Goodfellow et al., 2016].

the enterprises tend to target hyper-ambitious moon-shot projects that will completely transform the company/product and give over-sized return or investment. Such projects can take forever to complete and pushes the teams to their limits. Ultimately, the business leaders lose their confidence in the project and stop the investment; b) *lack of clearly defined business goals*: often research is started without a clearly defined hypothesis. Team members often cannot answer what the goal of the project really is, what problem they are trying to solve, and which questions they want answered — and the leadership team often do not know how much time and money they are willing to spend; c) *lack of data and/or labeling of data*: The unavailability of labeled data is another key challenge that stalls many machine learning projects. From the MIT Sloan Management Review [Ransbotham et al., 2017], 76% of machine learning people combat this challenge by attempting to label and annotate training data on their own and 63% go so far as to try to build their own annotation automation technology. This means that a huge percentage of machine learning team expertise are lost for the labeling process.

1.2 The Learning in Deep Learning

In terms of linear regression models, the parameter can simply be obtained analytically by solving the *normal equations* [Bishop, 2006]. However, when the model becomes non-linear and increasingly complex like in deep learning, an iterative method is generally required to obtain an estimate of the parameter. The *learning* in deep learning refers to the process of estimating the *parameter* (Section 1.2.5) under a predefined *cost function* (Section 1.2.3) on the basis of a *training and test dataset* (Section 1.2.1), means for *regularization* (Section 1.2.4) and a *model function* (Section 1.2.2) using an *optimization* procedure (Section 1.2.6).

1.2.1 Training and Test Datasets

We define the *training dataset* as N input data points $x_n \in \mathbb{R}^{T_1}$ (e.g. image, sensor data, etc.) and N output data points $y_n \in \mathbb{R}^{T_L}$ (e.g. class, target measurement, etc.) where $n = 1, 2, ..., N$ is the example index. We use the same nomenclature for the *test dataset*, which consists of N_{test} examples not contained in the training dataset. The purpose of the test dataset is to assess how the model performs on new unseen data that has not been part of the training dataset. Usually, the test dataset is evaluated repeatedly throughout the learning process to monitor the generalization capability of the model. Whenever appropriate, we denote a new unseen example not contained in the training dataset by x_0.

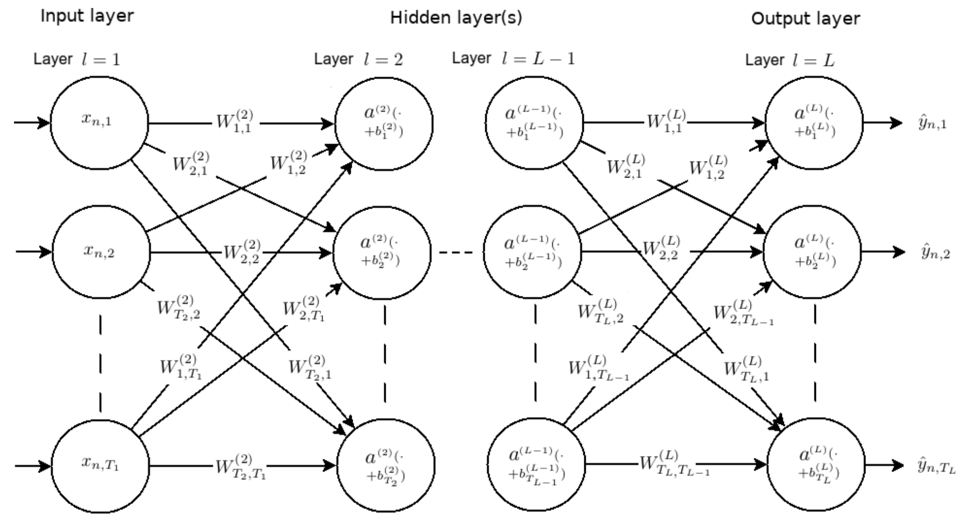


Figure 1.1: A feed-forward neural network with dense layers.

1.2.2 Model Function

The *architecture* of the deep learning model describes the relationship between inputs and outputs, and is formally expressed by a *model function* $f : \mathbb{R}^{T_1} \to \mathbb{R}^{T_L}$, which matches the dimensionalities of the input and output components of the training dataset. Usually, when $T_L = 1$ and the outputs are real numbers, we have a deep learning *regression* model. Whenever $T_L > 1$ and the outputs are categorical, we have a classification model. However, in principle we can also define a multi-output regression model with $T_L > 1$, and all other kinds of combinations. When the model function can be represented by a *directed acyclic graph* (DAG) [Bishop, 2006], we have a *feed-forward neural network* deep learning model, whereas if the graph is cyclic, we have a *recurrent neural network* deep learning model [Goodfellow et al., 2016].

The model function works through a series of L layers starting at the *input layer* and stopping at the *output layer* as shown in Figure 1.1. Usually this is conceptualized as drawing the DAG in a left-to-right fashion where the input is fed on the left hand side, and the output is obtained at the right hand side. The first layer, the input layer, has T_1 vertices with T_1 incoming edges and simply represents the input example $x_n \in R^{T_1}$ where n denotes the example index. The consecutive layers, the *hidden layers*, receive their inputs from the previous layers, and form linear combinations of these with their *weights* (in Figure 1.1 denoted by W, which is a part of the parameter) represented by their incoming edges. These linear combinations are often implemented as matrix multiplications, hence resulting in so-called *dense* layers in which every pair of distinct (input to output) vertices are connected by a unique edge (i.e. parameter value) [Bengio, 2009]. When the linear combinations are implemented using linear convolutions, the resulting layers are called *convolutional* layers [LeCun et al., 1995]. Convolutional layers have several interesting properties: *Sparse interaction* means that the output of the layer is dependent on a limited number of the inputs, but with multiple consecutive convolutional layers, the vertices become indirectly connected so series of convolutional layers can efficiently describe complicated interactions between several variables; *parameter sharing* limits the architecture of the model, reduces memory requirements and is more than often seen to improve the quality of models; *equivariant to translation* means that a convolutional layer can learn patterns regardless of shifts in the input.

Other types of layers also exists: *batch-normalization* layers [Ioffe and Szegedy, 2015] are commonly used for stabilization through normalization of the layers' inputs by re-centering and re-scaling, *pooling* layers [Weng et al., 1993] can be deployed for dimensionality reduction, *dropout* layers [Srivastava et al., 2014] can be used as regularization and *residual* layers [He et al., 2016] allow for training deeper networks by dealing with the *vanishing gradients* [Hochreiter et al., 2001] problem in terms of *skip connections* [Drozdzal et al., 2016].

As a last step per hidden layer, the *biases* (in Figure 1.1 denoted by b, which can be viewed as the intercepts of the parameter) are added to the layers' outputs before a non-linear *activation function* (in Figure 1.1 denoted by a) is applied in an element-wise fashion. Non-linear activation functions are what makes deep learning models non-linear, and is generally what fuels neural networks' powerful expressiveness. A multitude of activation functions currently exists, such as the *rectified linear unit* (ReLU) [Glorot et al., 2011], the *sigmoid* and the *tanh* [Goodfellow et al., 2016], to name a few. At this stage, it can be useful to conceptualize that a feed-forward neural network with dense layers utilizing only linear activation functions would simply collapse into multiple linear regression.

Finally, the last layer, the output layer, has T_L vertices and produces the final output of the network (in Figure 1.1 denoted by $\hat{y}$). The output-layer closely resembles the hidden layers in terms of that it also applies its own weights and biases, but the difference lies in the type of activation function utilized. For classification models, the *softmax* function [Bridle, 1989] is typically utilized, as it normalizes the outputs relative to the different classes and thus provides a *probability* output. For regression models, a linear activation function is usually applied.

1.2.3 Cost Function

The cost function measures the distance between the output components of the training dataset and the outputs of the model function given the corresponding input components of the training set. As the model function depends on the parameter, the cost function can be *minimized* with respect to the parameter to achieve a minimum total distance for the entire training set. For regression models, the *mean squared error* cost function is typically utilized, whereas for classification models, the *cross entropy* [Goodfellow et al., 2016] cost function is commonly deployed.

1.2.4 Regularization

Regularization is a general technique with the goal of making an ill-posed problem well-posed. As over-fitting is one example of an ill-posed problem, regularization is commonly applied in deep learning to balance the so-called *bias-variance trade-off*. The bias-variance dilemma is the conflict in trying to minimize the cost function while simultaneously maximizing the model's generalization capability for new unseen input data.

A multitude of regularization techniques exist in the deep learning domain. Ranging from classical *penalizing* based methods such as L_1/L_2-regularization [Krogh and Hertz, 1991] to more sophisticated approaches such as *dropout regularization* [Srivastava et al., 2014]. Penalizing methods typically works by introducing a parameter-dependent penalty term on the cost function, and can be thought of as a way of shrinking the parameter towards zero during training. In turn, this will lead to simpler models that are less likely to over-fit. Quite differently, dropout regularization is achieved by altering the architecture by introducing *dropout layers*. Dropout layers work by randomly setting some of their outputs to zero during training. As a consequence, the network becomes less sensitive to specific weights and biases, and this in turn results in networks with better generalization capabilities which are less likely to over-fit.

1.2.5 Initializing the Parameter

In order for the optimization procedure to start evaluating the cost function, the parameter must first be initialized. There are two main heuristics for this single purpose, and they both generally lead to a drastic reduction in the amount of required optimization steps. While the parameter consists of both weights and biases, both these methods were derived to deal exclusively with the weights — and assumes that the biases are initialized to zero. The process is therefore referred to simply as *weight initialization*.

Depending on the type of activation functions used within the individual layers, *xavier/glorot* initialization [Glorot and Bengio, 2010] was derived for the weights associated with *sigmoid/tanh* activation functions, while *he* initialization [He et al., 2015] is intended for weights associated with *ReLU* activation functions. The two methods are distinguished by that the former draws its initial weight values from an uniform probability distribution, while the latter from a Gaussian probability distribution. Although technically different, both methods parameterize their respective probability distributions according to the number of weights in the individual layers.

1.2.6 Optimization

Having all the required building blocks in place, the optimization procedure can finally take place in order for the network to learn the parameter from the training dataset. The central optimization algorithm in deep learning is *gradient descent* [Bottou et al., 2018]. As gradient descent is a first-order iterative optimization algorithm for finding a local minimum of differentiable functions, it can efficiently be applied to minimize the cost function with respect to the parameter as long as estimates of its gradient can be computed efficiently. Thanks to the *back-propagation* algorithm [Linnainmaa, 1976], this is indeed possible in deep learning. The back-propagation algorithm has later been identified as a special case of *automatic differentiation* [Wengert, 1964]. Today, most deep learning software frameworks, such as TensorFlow [Abadi et al., 2015] and PyTorch [Paszke et al., 2017], include simple functions for calculating the gradient of arbitrary expressions based on automatic differentiation. In fact, the whole philosophy behind these deep learning frameworks is built upon the idea of decomposing the model architecture into graph(s) sequentially representing all the involved mathematical operations. Gradients are then computed by automatic differentiation which works by applying the chain-rule of calculus repeatedly to these operations, and therefore derivatives of arbitrary order can be computed automatically and accurately to working precision.

Gradient descent simply works by taking repeated steps in the opposite direction of the gradient at the current point. Because this is the direction of steepest descent, the method is guaranteed to follow a down-hill path and converge to a point in parameter space where the gradient is (approximately) zero. Although such points need not necessarily be true global nor local minima, the success of gradient descent in deep learning prominently lies in the fact that it works very well in practice. Nevertheless, second-order optimization is also an active research topic in deep learning, but has generally not found much popularity mainly due to the prohibitively expensive computational cost that comes with repeated Hessian evaluations [Anil et al., 2020].

Gradient descent comes in a range of different variants. The primary source of variance lies in how the training data is used when estimating the gradient. In its original form, the full training set is utilized at every step. But as repeated gradient estimates based on the full training dataset is computationally demanding, *stochastic* gradient descent [Robbins and Monro, 1951] performs a parameter update for each single training example. A natural extension and more cost-efficient approach is *mini-batch* gradient descent, where the gradient is approximated by calculating the mean of the gradients on subsets (e.g. mini-batches) of the entire training dataset. The latter two methods often introduce yet another level of stochasticity by periodically shuffling the training dataset so that the gradient always will be based on a different sample.

The second source of variance is rooted in subtle details around how to best scale the gradient when performing the down-hill steps. This scaling factor is widely known as the *learning rate*. Here we find a range of methods specifically developed for deep learning. Common for most of these methods is that they utilize an *adaptive* adjustment of the learning rate. Besides stochastic gradient descent *with momentum* [Rumelhart et al., 1986], the two most used optimization methods in deep learning are *Adam* [Kingma and Ba, 2014] and *root mean square propagation* (*RMSProp*) [Tieleman and Hinton, 2017]. While the latter adaptively adjusts the learning rate based on a running average for each of the individual gradient components, the former operates by adjusting separate learning rates for each of the gradient components in terms of both the running average and the running variance.

1.3 Famous Architectures and Datasets

The *LeNet* architecture [LeCun et al., 1995, 1998] introduced in 1995, is famous due to its historical importance. As the first successful neural network with convolutional layers, it achieved state-of-the-art classification performance on the *MNIST* hand-written digit dataset. Thanks to the internet, the tiny color image datasets *CIFAR-10* and *CIFAR-100* [Krizhevsky et al., 2009] subsequently came to life. With the introduction of *GPUs*, further breakthroughs followed. A range of successful deep network architectures including *AlexNet* [Krizhevsky et al., 2012], *VGG* [Simonyan and Zisserman, 2014], *GoogLeNet* [Szegedy et al., 2015], *ResNet* [He et al., 2016], and *DenseNet* [Huang et al., 2017] now operates on large scale datasets consisting of millions of high resolution images such as *ImageNet* [Deng et al., 2009] and *OpenImage* [Kuznetsova et al., 2020].

Chapter 2

Uncertainty Quantification in Deep Learning

This chapter briefly summarizes uncertainty quantification methodology applicable (and inapplicable) to deep learning, before we round off with a more in-depth introduction to the Delta method which lies the foundation for this thesis. For an excellent and comprehensive survey of uncertainty quantification in deep learning, we refer to [Gawlikowski et al., 2021]. For the broader machine learning perspective, we refer to [Hüllermeier and Waegeman, 2020].

2.1 Introduction

In statistical modeling, there are two major schools of thought. These are known as the *Frequentist* and the *Bayesian* school of thought. In the Frequentist's view of statistical modeling, the true, unobservable value ω^* of the model parameter ω is assumed fixed and unknown, and the estimate $\hat{\omega}$ is based on a random sample of data points (the training dataset). The values of $\hat{\omega}$ will thus vary from sample to sample, making $\hat{\omega}$ itself a random variable. Therefore, the uncertainty about the point estimate of ω^* will be reflected in the variance of $\hat{\omega}$. Consequently, this variance will indicate how much $\hat{\omega}$ is expected to change with respect to a different sampling of data. The uncertainty of predictions can thus be seen as the result of propagating the uncertainty of the parameter estimate through the model function when applied to new input x_0, hence producing an output y_0 with its own variance, i.e. the *predictive uncertainty*.

In Bayesian statistics [Bolstad and Curran, 2016], statements about the parameter are made in terms of probability distributions that are conditioned on observed data (the training dataset), denoted by the *posterior distribution.* Uncertainty about the parameter is thus expressed in terms of probability, which is why the Bayesian framework is often referred to as *probabilistic modeling.* The Bayesian equivalent to predictive uncertainty is thus described by the variance of the *posterior predictive distribution* which can be viewed as an ensemble of models generated from the posterior distribution.

Regardless of the statistical approach taken, there are two types of uncertainty [Hüllermeier and Waegeman, 2020]: *Epistemic uncertainty* is commonly understood as the reducible component of uncertainty — the uncertainty of the model itself, or its parameters. While the epistemic uncertainty can be reduced by increasing the amount of training data, the other component of uncertainty known as *aleatoric uncertainty*, is a random and irreducible quantity and stems from the uncertainty in the output (label/target) assignment process. For this reason, the aleatoric uncertainty is also often called *data uncertainty.*

The *predictive uncertainty* consists of the sum of both the epistemic uncertainty and the aleatoric uncertainty — when propagated onto predictions. Common for most of the Bayesian methods is that they quantify the sum of both these components, and as a consequence, further actions must be taken in order to separate the two [Kwon et al., 2020]. However, our Delta methodology introduced in Section 2.2 is concentrated on the epistemic part, and we therefore refer to it as the *predictive epistemic uncertainty.* The meaning is therefore different from the standard interpretation [Devore et al., 2012].

2.1.1 The Goal of Uncertainty Quantification

After having trained a neural network to obtain the parameter point estimate $\hat{\omega}$, the traditional approach proceeds by computing only predictive point estimates $\hat{y}_0 = f(x_0, \hat{\omega})$ for new inputs x_0, simply by plugging x_0 into the model function along with $\hat{\omega}$. But in reality we know there will be two sources of uncertainty in this process: a) the epistemic component of uncertainty baked into our model as a result from having trained using only a limited amount of training data; and b) the aleatoric component of uncertainty as a result of noisy target measurements. Therefore, we can take these sources of uncertainty into account and rather express predictions as $\hat{y}_0 = f(x_0, \hat{\omega}) \pm \widetilde{\sigma}_0$ where $\widetilde{\sigma}_0$ is an estimate of predictive uncertainty associated with x_0.

To illustrate this further, we now consider a data generating function $g(x) = \sin(x) + x$ from which we sample a training dataset in two different regions of x (around $x = -0.5$ and $x = 0.5$). We define a heteroskedastic measurement noise function $\epsilon(x)$ that we add to $g(x)$ when we obtain samples from it. By hetereoskedasticity we here mean that the emulated measurement noise we add is itself a function of x. We emulate the measurement noise by letting $\epsilon(x)$ be normally distributed with zero mean and with a variance proportional to x. Hence, the training data we generate will be randomly be corrupted with low noise values for low values of x — and high noise values for high values of x. With reference to Figure 2.1, we can now explain the two involved sources of predictive uncertainty as follows: a) the epistemic component of $\widetilde{\sigma}$ will be high in regions where there is little or no training data (x not around -0.5 nor 0.5) simply because the model lacks the ability to safely deal with these values of x (for which it has no training experience); and b) the aleatoric component of $\widetilde{\sigma}$ will be low around $x = -0.5$ and high around $x = 0.5$ because here the training data suggests low and high measurement noise, respectively. Thus, generally, the aleatoric component of uncertainty simply accounts for the stochasticity of data generating processes which may (e.g. heteroskedastic) or may not (e.g. homoskedastic) be a function of x.

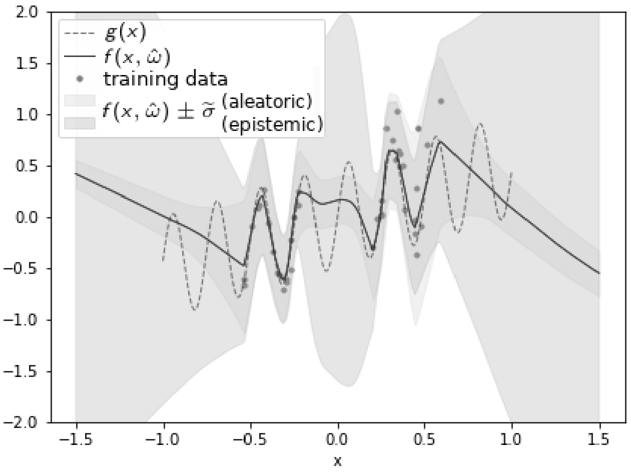


Figure 2.1: A data generating function $g(x) = \sin(x) + x$ (dotted gray line) corrupted by heteroskedastic noise, is sampled in two different regions of x (around $x = -0.5$ and $x = 0.5$) to obtain a training dataset (red dots). A neural network is trained using this training dataset, and outputs predictive point estimates $f(x, \hat{\omega})$ as shown by the solid blue line. The orange shaded interval around the point estimate corresponds to the epistemic component of uncertainty, while the green shaded interval corresponds to the aleatoric component of the uncertainty. Original image credit: Martin Krasser

2.1.2 The Bootstrap and Deep Ensembles

A classical frequentist approach to obtain uncertainty estimates is known as the Bootstrap [Efron, 1992; Lakshminarayanan et al., 2017; Osband, 2016]. In terms of deep learning, the Bootstrap procedure is often referred to by the name *Deep Ensembles*.

While this method can be extended to quantify also aleatoric uncertainty [Khosravi and Creighton, 2011], the standard method deals with the epistemic counterpart. Essentially, the method requires an *ensemble* of neural network models. By assessing the variance over model ensemble predictions, predictive epistemic uncertainty estimates can thus be obtained. The simplicity of the approach is the key advantage, while the disadvantage is the need to build and maintain multiple models in parallel. The classical Bootstrap method starts by creating B datasets from the original training dataset by sampling with replacement. Subsequently, B models are trained separately on each of the bootstrapped datasets. The variance captured by the Bootstrap is thus seen caused by the change in $\hat{\omega}$ relative to the different samples of the training dataset. In the deep learning domain, the procedure is straightforward although several subtleties naturally arise. Given the implicit stochasticity of gradient descent through *mini-batching* and *training data shuffling* (Chapter 1, Section 1.2.6), these sources of variance will also affect the outcome. Clearly, another source of variation is *random weight initialization* (Chapter 1, Section 1.2.5), not to mention the so-called *indeterminism* [Nagarajan and Warnell, 2019] present in many deep learning software frameworks. We have included a deterministic TensorFlow implementation of the Bootstrap algorithm in the `pyDeepDelta` [Nilsen, 2018-2021a] provision under the name `pydeepboot.py`, and further details can be found in **Paper 3** [Nilsen et al., 2021b].

2.1.3 Bayesian Methods

Bayesian statistics provides a coherent framework for representing uncertainty in neural networks [Blei et al., 2017; Brooks et al., 2011; Gal and Ghahramani, 2016; Gundersen et al., 2020; MacKay, 1992a; Murray, 2018]. The Bayesian approach makes inferences based on the full distribution of a model output y_0, for an input example x_0. This is called the *posterior predictive distribution*, given by

$$p(y_0|x_0, X, Y) = \int p(y_0|x_0, \omega)p(\omega|X, Y)d\omega, \tag{2.1}$$

where the *likelihood distribution*, $p(y_0|x_0, \omega)$, is the distribution of the model output y_0, given an input x_0 and a model function parameterized by ω. The second factor, $p(\omega|X, Y)$, is called the *posterior distribution*, and is the distribution of the model parameter itself given the training dataset here denoted by X (input) and Y (output).

The posterior predictive distribution can thus be viewed as an ensemble of models with different settings of ω, weighted by the posterior. Thus, the most likely parameter values will contribute the most to the probability of y_0, taking on a particular value. Making statements about parameter values and predictions in terms of probability highlights one of the key features of the Bayesian approach: it yields an intuitive and coherent way of expressing uncertainties in terms of the variances of the different distributions.

The (co)variance of the posterior distribution can thus be seen as the uncertainty of the model parameter, and can therefore be regarded as the source of the epistemic component of uncertainty. In contrast, the variance of the likelihood distribution can be regarded as the source of the aleatoric component of uncertainty. When combined via Equation (2.1), the variance of the predictive posterior distribution quantifies the predictive uncertainty associated with x_0 as a combination of both the aleatoric and the epistemic counterparts. At this point it can be useful to realize that with an infinite amount of training data (i.e. $N \to \infty$), the posterior distribution would approach a Dirac delta function centered around the true unknown parameter value ω^* (i.e. $p(\omega|X, Y) \to \delta(\omega - \omega^*)$). The integral (2.1) would therefore collapse, and only the irreducible variance of the likelihood at ω^* (e.g. aleatoric uncertainty) would contribute to the total predictive uncertainty.

Most of the uncertainty quantification methods in the Bayesian regime seek to approximate the integral represented by Equation (2.1). To this end, the classical *Monte Carlo method* [Krauth, 1998] proceeds by drawing samples from the posterior distribution and propagate these to obtain samples from the posterior predictive distribution. Finally, to obtain predictive point estimates $\hat{y}_0$ with predictive uncertainty $\sigma(x_0)$, the mean and standard deviation of these samples are computed.

If the sampling of the posterior distribution becomes too computationally costly (which is usually the case in deep learning), one can turn to other approximation methods. These includes, but are not limited to, the *Laplace approximation* [Neal, 2012], *Stochastic expectation propagation* [Li et al., 2015] and *Langevin diffusion* methods [Welling and Teh, 2011]. In particular, the Laplace approximation is obtained by taking the second-order Taylor expansion around the mode of the posterior ($\hat{\omega}$). This leads to a Gaussian distributed (approximate) posterior, which in turn can more easily be sampled. However, this approach leads to expressions involving the inverse Fisher information matrix and is therefore in its direct form still prohibitively expensive in deep learning due its quadratic space and cubic time computational complexity in the number of parameters. A line of research seeking to further reduce the computational complexity of the Laplace approximation is discussed in more detail in Section 2.1.4.

Another major drawback with the pure Monte Carlo method in deep learning is its poor convergence properties (i.e. high variance) for high-dimensional parameter spaces. Other sampling methods intended to solve this exist, such as *Importance sampling* [Skaug and Fournier, 2006], the *Metropolis-Hasting Algorithm*, the *Gibbs sampler* and *Hamiltonian Monte Carlo* (HMC) [Neal et al., 2011]. These methods are all known as Markov Chain

Monte Carlo (MCMC) methods and they might improve the convergence properties compared to random sampling, but are usually still not tractable in the deep learning domain. However, a viable alternative is *Variational Inference* [Blei et al., 2017].

Like the Laplace approximation, Variational Inference addresses the sampling issue by approximating complex posteriors using much simpler distributions (i.e. *variational distributions*) parameterized by so-called *variational parameters*. This method turns the problem into an optimization problem with the goal of estimating the variational parameter by minimizing the corresponding *Kullback-Leibler divergence* [Kullback, 1997] between the variational distribution and the true posterior.

The *Dropout* method [Gal and Ghahramani, 2016], often referred to as *Monte Carlo Dropout* or simply *MC-Dropout*, can be viewed as approximate Variational Inference [Gundersen et al., 2020]. Given deep learning architectures regularized (and trained) using *dropout layers* (Chapter 1, Section 1.2.4), MC-Dropout generates a distribution over the model parameter by leaving the dropout-layers activated also during prediction. The process is thus similar to the Bootstrap procedure, where the empirical variance can be computed based on the resulting ensemble of predictions.

In the original works of MacKay [MacKay, 1992a,b], a Bayesian neural network training procedure based on the Laplace approximation is proposed. While this method allows for separate quantification of aleatoric and epistemic uncertainty, it relies on repeated computations of the full inverse Fisher information matrix at training time. It can thus be regarded as a second-order optimization method, and is therefore in its original form intractable in deep learning.

2.1.4 Approximating the Laplace Approximation

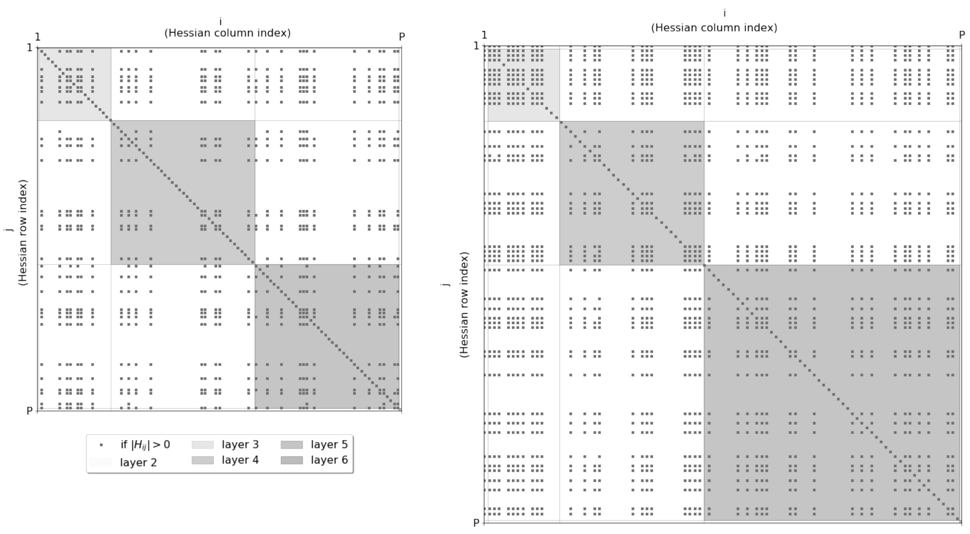
The yet so high computational complexity of the Laplace approximation can be mitigated by several approximations of the Fisher information matrix. Different approximations of the Fisher have been proposed in various contexts. This includes its *diagonal approximation* [Becker and Le Cun, 1988; Salimans and Kingma, 2016], the *last-layer approximation* [Kristiadi et al., 2020], the *block-diagonal approximation* [Lee et al., 2020; Ritter et al., 2018] and others [Hennig, 2013; Le Roux and Fitzgibbon, 2010; Liu and Nocedal, 1989]. A line of research focusing on Kronecker-factorization [Botev et al., 2017; Martens and Grosse, 2015] of the block-diagonal approximation starts with [Lee et al., 2020; Ritter et al., 2018].

To illustrate these approximations, we in Figure 2.2 visualize the full Fisher information matrix (i.e. Hessian) for the LeNets discussed in **Paper 2** [Nilsen et al., 2021a]. To be clear, this sort of computation and visualization is tractable only for the smallest networks discussed in the paper. To see this, consider that the full Fisher of the CIFAR-10 LeNet occupies 56 GBs of memory and takes 78 hours to compute; whereas in contrast, the full Fisher of the large CIFAR-10 ResNet (also discussed in **Paper 2**) would occupy 500 TBs of memory, and take an estimated 35 years to compute!

With reference to Figure 2.2, the diagonal approximation thus concentrates solely on the values along the main diagonals, while the last-layer approximation concentrates on the values contained in the barely visible purple squares (i.e. layer 6) in the lower right corners. As the block-diagonal approximation assumes layer-wise independence of the parameters, it is limited to the values contained in the color-shaded squares representing the individual layers of the neural networks (i.e. no cross-terms involved).

These plots suggest that there is no specific structure to the Fisher in deep learning image classification. Except from that L_2-regularization imposes a relatively strong diagonal structure, the other non-zero entries are more or less uniformly distributed across the whole matrix. We find that the sparsity level of the Fisher is extremely high in general. The number of non-zero entries is just 1% for MNIST and 2% for CIFAR-10. Nevertheless, to highlight the structure in these visualizations we had to make the blue dots where $|H_{ij}| > 0$ large enough to be visible, and this in a sense gives a misleading perception of the true sparsity levels.

In the next Section 2.2, we introduce the Delta method and connect it to the Laplace approximation.



(a) MNIST (b) CIFAR-10

Figure 2.2: Visualization of the structure and sparsity level of the Fisher information matrix for the MNIST ($P = 93,322$) and CIFAR-10 ($P = 122,570$) LeNets. The raw matrices were first computed exactly by Pearlmutter's technique, followed by down-sampling to 100×100 entries using bicubic interpolation for visualization purposes. Finally, to preserve the relative scale of the two different sized networks, the final images were scaled accordingly. A blue dot is shown in column i and row j when the absolute value of the down-sampled Hessian in location ij is greater than zero. The neural network layers are visualized by the color-shaded squares. Note that layer 2 (upper left corners) and layer 6 (lower right corners) are barely visible due to their relative small number of parameters.

2.2 The Delta Method

Under the framework of *Maximum Likelihood Estimation* [Le Cam, 1990], the deep learning model parameter estimate $\hat{\omega}$ is called the *Maximum Likelihood Estimator* (MLE). Alternatively, when the cost function is regularized using parameter-dependent penalty terms (e.g. L_1/L_2-regularization), $\hat{\omega}$ is called the *Maximum A Posteriori* (MAP) *Estimator*. With the theory of maximum likelihood comes a truly remarkable theoretical result. It has been referred to as "arguably the most important result in theoretical statistics" [Geyer, 2003], and has been discussed in thousands of research papers ever since its first appearance in the original work of R. A. Fisher — the inventor of the method of maximum likelihood. The derivation of the result builds on *the law of large numbers*, the *central limit theorem*, *Taylor expansions* and repeated use of *Slutsky's theorem*. We point to [Geyer, 2003] for arguably the best intuition building treatment of the topic available on the internet to date. Despite the fact that the MLE or MAP estimates usually cannot be described by an explicit formula (this is why deep learning utilizes a optimization procedure), maximum likelihood continues to give an explicit *large sample approximation* [Van der Vaart, 2000] to the sampling distributions of these estimators. In particular, this means that estimates of the mean and variance of the distributions describing the MLE or MAP estimates can be obtained on the basis of the training dataset — hence allowing for epistemic uncertainty quantification. The large sample approximation is as follows

$$\hat{\omega} \sim \mathcal{N}(\omega^*, \Sigma). \tag{2.2}$$

In words, for a large (enough) training dataset size N — the distribution of the MLE or MAP estimates are approximately normal with mean ω^* (the true unknown parameter value) and approximate covariance Σ. When the likelihood distribution for which the MLE or MAP estimate is based upon is of the form of the true likelihood distribution of the training dataset, the covariance matrix Σ can be estimated by the *Hessian estimator* defined H^{-1} or the *Outer-Product of Gradients* (OPG) *estimator* defined G^{-1}. In contrast, when the true likelihood distribution of the data is *not* of the form of the one used for any ω, the covariance matrix can be estimated by the *Sandwich estimator* defined $H^{-1}GH^{-1}$ [Freedman, 2006; Geyer, 2003; Schulam and Saria, 2019]. In the literature, the two matrices H and G are both referred to as the empirical *Fisher information* matrix, and can be estimated based on the N examples in the training dataset: the matrix H is the empirical Hessian with respect to ω of the cost function evaluated at $\hat{\omega}$, and the matrix G is the empirical covariance of the gradients with respect to ω of the cost function evaluated at $\hat{\omega}$.

For convenience, we now restrict the reminder of this introduction to deep learning regression models, with scalar model functions $f(x_0, \omega) : \mathbb{R}^{T_1} \to \mathbb{R}$. An extension to the classification setting can be found in **Paper 2** [Nilsen et al., 2021a]. It can be shown by a first-order Taylor approximation [Grosse, 2020; Khosravi and Creighton, 2011] that the variance (i.e. squared predictive epistemic uncertainty) associated with the prediction of x_0 can be obtained by

$$Var(\hat{y}_0) = \nabla f^T \Sigma \nabla f, \tag{2.3}$$

where $\hat{y}_0 = f(x_0, \hat{\omega})$ is the neural network output, Σ is the parameter covariance estimate in (2.2), and where ∇f is the gradient with respect to ω of the model function evaluated at $\hat{\omega}$. The two expressions (2.2) and (2.3) together form what is known as the *Delta method.* At this point, we clearly see that the Delta method is closely related to the Laplace approximation. Both methods build upon the large sample approximation (2.2), but the difference lies in how the predictive uncertainty estimates are obtained. The standard Laplace approximation uses (2.2) to obtain samples of the posterior distribution, and plug these into an approximation of the variance of (2.1) (i.e. sum), while the Delta method simply evaluates the differential expression (2.3).

The main obstacle with the classical Delta method applied in deep learning is its high computational cost. The space complexity is quadratic as the Fisher information matrix grows quadratically with the dimensionality of the parameter, and the time complexity is cubic due to its inversion. In the next Chapter 3 we introduce the three papers included in the dissertation which together addresses and details how to get around these aforementioned computational difficulties. In this perspective, it is well worth to realize that the methodology we introduce can be seen as a new way of approximating the **full Fisher information matrix with bounded approximation errors**, and thus fits into the Laplace approximation line of research discussed in Section 2.1.4.

In the next and final Section 2.2.1, we round off this chapter by developing a hybrid approach combining the standard Laplace approximation with the key ideas from **Paper 2** [Nilsen et al., 2021a] — resulting in an efficient Laplace Approximation based Monte Carlo sampling algorithm not published elsewhere. A small demonstration of its capabilities is further presented in Chapter 4.

2.2.1 An Efficient Laplace Approximation based Monte Carlo Sampler

The covariance approximation developed in **Paper 2** [Nilsen et al., 2021a] is repeated here for convenience. It is defined by

$$\widetilde{\Sigma} = \frac{1}{N}\left[Q_{\mathrm{L}}\Lambda_{\mathrm{L}}^{-1}Q_{\mathrm{L}}^{T} + \widetilde{\lambda}^{-1}(I - Q_{\mathrm{L}}Q_{\mathrm{L}}^{T})\right], \tag{2.4}$$

where $Q_{\mathrm{L}}\Lambda_{\mathrm{L}}Q_{\mathrm{L}}^{T}$ is the low-rank approximation of the Fisher information matrix based on its top K eigenpairs, and where $\widetilde{\lambda}$ is the approximation constant used for the remaining eigenvalues. The first step in the standard approach [Gentle, 2009] to sampling the normal distribution (2.2) is to factorize the covariance matrix as $\widetilde{\Sigma} = AA^{T}$. Conveniently, it turns out that under the approximation (2.4) the matrix A is simply given by

$$A = \frac{1}{\sqrt{N}}\left[Q_{\mathrm{L}}\sqrt{\Lambda_{\mathrm{L}}^{-1}}Q_{\mathrm{L}}^{T} + \sqrt{\widetilde{\lambda}^{-1}}(I - Q_{\mathrm{L}}Q_{\mathrm{L}}^{T})\right]. \tag{2.5}$$

Obtaining T samples $\hat{\omega}^{(t)}$ from the posterior can now be done by repeatedly perturbing the parameter estimate $\hat{\omega} \in \mathbb{R}^{P}$ with matrix vector products between A and vectors

$z^{(t)} \sim \mathcal{N}(0, I) \in \mathbb{R}^P,\ t = 1, 2, \ldots, T,$

$$\hat{\omega}^{(t)} = \hat{\omega} + Az^{(t)} = \hat{\omega} + \frac{1}{\sqrt{N}} \left[Q_{\mathrm{L}} \sqrt{\Lambda_{\mathrm{L}}^{-1}} Q_{\mathrm{L}}^T z^{(t)} + \sqrt{\widetilde{\lambda}^{-1}} (z^{(t)} - Q_{\mathrm{L}} Q_{\mathrm{L}}^T z^{(t)}) \right]. \tag{2.6}$$

Finally, the predictive mean and variance for an arbitrary input example x_0 can be obtained by evaluating the mean and variance of the model function for x_0 across the posterior samples

$$\hat{y}_0 = \frac{1}{T} \sum_{t=1}^{T} f(x_0, \hat{\omega}^{(t)}), \qquad \widetilde{\sigma}^2(x_0) = \frac{1}{T-1} \sum_{t=1}^{T} \big(f(x_0, \hat{\omega}^{(t)}) - \hat{y}_0 \big)^2. \tag{2.7}$$

Bounds for the approximation error can readily be obtained by using the error propagating technique presented in **Paper 2**, Section 4.3.1. A Python implementation of the proposed algorithm is included in the `pyDeepDelta` [Nilsen, 2018-2021a] provision under the name `pydeepdelta_sampler_demo.ipynb`.

Chapter 3

Introduction to the Papers

The fundamental research question we seek to answer in this dissertation is twofold: a) can the classical Delta method from statistics be applied in modern deep learning to quantify model (epistemic) uncertainty? And b) can it be useful in image classification?

While the short answer of a) above turns out to be yes, this chapter summarizes the path of work that was conducted in order to answer the question, and places the three papers denoted by **Paper 1, 2 and 3** along this route. Regarding b), a discussion of its usefulness in image classification is presented in the next Chapter 4. We start off by a little background section which allows this chapter to be read as a stand-alone piece of text.

3.1 Background

A modern deep neural network can be described by a complicated random multivariate function $y = f(\omega, x)$ of a P-dimensional random variable $\omega = \begin{pmatrix} \omega_1 & \omega_2 & \dots & \omega_P \end{pmatrix}^T$ which maps constant inputs x to random outputs y. Traditionally, by a training procedure and a set of input/output examples, we get a constant point estimate $\hat{\omega}$ of the function variable so that we later on can use it to predict point estimates $\hat{y}_0$ of the outcome y_0 for arbitrary input examples x_0 simply by plugging $\hat{\omega}$ into f along with x_0.

The question is, are there other viable alternatives in which we compute not only point estimates of ω and y but rather their full probability distributions? Yes, conveniently the Delta method is an analytical approach towards quantifying both the mean and variance of multivariate functions. In principle we can use it to compute the mean and variance of both ω and y and hence get vital information about their respective probability distributions. If we regard $\hat{\omega}$ as the mean of ω, a first-order Taylor expansion of y around $\hat{\omega}$ yields an approximation of the mean of y simply given by $f(\hat{\omega}, x)$. In this view, we recognize that at this point nothing deviates from the traditional approach: point estimates of predictions correspond to first-order Taylor approximations of the mean of y. But what about the variance? Indeed, an analogous first-order Taylor approximation of the variance of y is given by the expression $\nabla y^T \Sigma \nabla y$ where Σ is the covariance matrix of ω. Thus, if we manage to compute this new expression, we can supplement point estimates of predictions with the underlying predictive variance, i.e. the predictive uncertainty. As we saw in Chapter 2, the covariance matrix Σ can be estimated by several different procedures. Common for these methods are that they all involve the inverse of the Fisher information matrix, i.e. the inverse Hessian matrix and/or the inverse Hessian OPG approximation matrix. In other words, the variance expression mainly consists of gradients and the Hessian (i.e. gradients of gradients), and the scene is set by the fact that modern deep learning software frameworks are built around automatic differentiation.

3.2 Summary

With the basic background information presented in the beginning of this chapter in mind, in **Paper 1** [Nilsen et al., 2019], we introduce the required methodology and software code for efficient computation of Hessian matrices within the modern deep learning software framework TensorFlow [Abadi et al., 2015]. Firstly, we demonstrate how to efficiently compute the exact Hessian matrix using Pearlmutter's technique [Pearlmutter, 1994]. Secondly, we introduce a per-example gradients technique serving as the foundation for efficiently computing the OPG approximation. Thirdly, we introduce and implement the Lanczos iteration and the Incremental SVD [Cardot and Degras, 2015; Levy and Lindenbaum, 2000; Trefethen and III, 1997] algorithm for efficient computation of the eigenvalue decomposition of the two aforementioned matrices. In particular, this addresses the difficulty with the large dimensionality of the Hessian matrix which grows quadratically with the number of variables/parameters in the deep learning model. Fourthly, we show how to obtain full-rank approximations based on low-rank approximations exploiting the well-known flatness [Keskar et al., 2016] of deep learning cost landscape minima.

The importance and motivation of the contributions in **Paper 1** is two-fold: a) to help relevant communities by providing a quick route to get Hessian estimates using TensorFlow, and b) to enable and support the calculations required by **Paper 2**. Regarding a) we evidently observe that the paper and the belonging released software code has been and still is quite popular on Github. In Chapter 4, Section 4.6 we take a closer look at the current impact of this work.

In **Paper 2** [Nilsen et al., 2021a], we build upon the methodology presented in **Paper 1**, and show how to implement an approximation of the Delta method using the same software framework. Firstly, we recognize the Delta method as a measure of epistemic as opposed to aleatoric uncertainty and break it into two components: the eigenvalue spectrum of the Fisher information (i.e. Hessian) of the cost function and the per-example sensitivities (i.e. gradients) of the model function. Secondly, we show how to approximate the naïve Delta method and thereby reducing the computational complexity in the number of parameters from quadratic in space and cubic in time, to linear in both space and time. Thirdly, we elaborate on how L_2-regularization counteracts the fact that the Hessian and OPG matrices tend to be singular in deep learning after training, and therefore virtually are not invertible. Furthermore, we discuss the well-known fact that the Hessian after training in deep learning usually is not positive definite, and show how to deal with this. Fourthly, based on the full-rank approximation introduced in **Paper 1** [Nilsen et al., 2019], we develop rigorous bounds of the approximation error of the proposed Delta methodology. We consider this error propagating technique as the most novel contribution of the dissertation. We show by examples that only a relatively small number of eigenpairs of the Fisher contribute to the predictive uncertainty for examples contained in the training set. Furthermore, we show that when the smallest estimated eigenvalue of the Fisher information matrix λ_K is close to the regularization rate parameter λ, the approximation errors for any example will be close to zero even when $K \ll P$. Fifth, we compare the three covariance estimators discussed, and show that they yield almost identical results. Sixth, we provide an accompanying TensorFlow implementation [Nilsen, 2018-2021a] of the methodology, and demonstrate how to it can be applied on a few well known architectures using the MNIST and CIFAR-10 datasets.

The importance and motivation of **Paper 2** can best be described by recognizing that this paper deals with a lot of different but interconnected topics, and acts as a reference filled with enlightening information which we expect can bridge the gap between people from the fields of statistics and machine/deep learning.

In **Paper 3** [Nilsen et al., 2021b], we compare the Delta method approximation introduced in **Paper 2** [Nilsen et al., 2021a] with the classical Bootstrap method introduced in Section 2.1.2. For the Bootstrap, we discard training data shuffling (and in-determinism by the use of an deterministic implementation) from the procedure, and consider the sole effect of training data bootstrapping plus two variants of random weight initialization: *dynamic random weight initialization* (DRWI) and *static random weight initialization* (SRWI). The former is based on using a different (i.e. dynamic) random seed for each of the networks in the ensemble, while the latter uses the same (i.e. static) seed across all of the networks. By a comparison with the Delta method, we show that the best correspondence between the predictive epistemic uncertainty estimates is achieved with the DRWI variant of the Bootstrap. We show that there is a strong linear relationship between the quantified predictive epistemic uncertainty levels obtained from the two methods when applied on two LeNet-based neural network classifiers using the MNIST and CIFAR-10 datasets. By the use of regressions, we quantify the similarity in terms of the relative and absolute uncertainty reflected respectively via the squared correlation coefficients (R^2) and the slopes (β). While the relative uncertainty similarity measured by the R^2 tends to be high (~ 0.9), the slopes β indicate that the absolute uncertainty levels differ slightly between the methods. In Chapter 4 we come back to this and make some further reflections regarding absolute and relative uncertainty. Lastly, we demonstrate by timings that the Delta method offers a five times computation time reduction compared to the Bootstrap.

The motivation of **Paper 3** is to validate the methodology proposed in **Paper 2**. This piece of work therefore first and foremost acts as a supplement to **Paper 2**, but can also be seen as the result of a need for an internal validation tool.

Chapter 4

Concluding Remarks and Future Work

We have shown via the three central papers introduced in Chapter 3 that the Delta method from statistics can be adapted to the modern deep learning context. However, this chapter is devoted to discuss its usefulness in image classification, and we take a critical look at the proposed methodology and focus on identified shortcomings and limitations. Secondly, we round off by pointing at several relevant research topics which could be investigated in the future as well as taking a brief look at the current impact and consequences of our research.

4.1 Implementation

Despite that the Delta method offers a relatively simple recipe to quantify epistemic uncertainty, the practical aspect of implementing the technology within a modern deep learning software framework was at the time of writing nothing but involved. This can be surprising, since at first glance the involved calculations in our methodology fits well with automatic differentiation.

As we saw in **Paper 1**, Pearlmutter's technique is in retrospect relatively simple to implement, but tremendous effort was spent before this simple TensorFlow formulation was realized and validated. Another surprisingly difficult aspect is the per-example gradients involved in the OPG approximation. This is counter-intuitive, since one of the motivations for the OPG approximation in the first place is to avoid second-order derivatives, and hence to simplify the required calculations. However, as per-example gradients cannot leverage from the parallelism offered by backward-mode automatic differentiation [Goodfellow, 2015], much effort was spent on developing the efficient per-example gradients technique introduced in **Paper 1** for this sole purpose.

Our efficient implementation in TensorFlow [Nilsen, 2018-2021a] serves as a basic starting point for trying out the Delta method in deep learning. Nevertheless, despite that the implementation currently consists of more than 1500 lines of code, its applicability to ever-changing architectures with new types of layers etc., can still be limited in terms of adaption needs. At a late stage in this study, however, our library also got company by a PyTorch [Paszke et al., 2017] implementation [Daxberger et al., 2021] of the closely related Laplace approximation and its variants.

4.2 Computing Considerations and Larger Architectures

Despite the fact that we have reduced the naïve Delta method's computational complexity to be linear in the number of parameters P, the presented methodology still requires considerable amount of computing power when P grows very large. Since the computational complexity also scales (linearly) with the number of eigenpairs K and the number of training examples N, it seems that with today's computing power, the bottleneck of our methodology is reached when the number of parameters is in the order of 10^7. In **Paper 2**, we showed that the computation time for a CIFAR-10 PreResNet-11 with $P \approx 11M$, $N = 50,000$ and $K = 200$ is about 15 hours using a AMD Ryzen 5 2600 CPU @ 3.4 GHz with eight cores and 32 GBs memory along with an NVIDIA RTX 2080 Ti based GPU with 11 GBs memory. The belonging memory consumption is hence close to 9 GBs assuming single precision floating point arithmetic. More generally, if the largest affordable K yields a λ_K far from λ, it can render the methodology intractable as the approximation errors can be too large.

However, on the positive side, the effect of computational complexity reduction can also be seen this way: computing the naïve Delta method using the Hessian estimator for the CIFAR-10 LeNet would take about 78 hours, and require about 57 GB of memory. Since in practice one would need to store both the Hessian and its inverse, as well as temporary variables depending on the type of inversion algorithm, the effective memory consumption can be as much as 320 GBs. With our approximate Delta methodology, the computation time is reduced to just about one hour with a memory requirement of about 500 MBs assuming single precision.

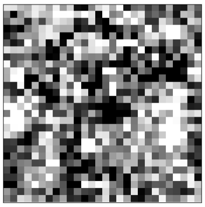


Figure 4.1: Resulting MNIST OoD example x_0. The predicted class '8' probability of this image is 0.9998, while the belonging predictive epistemic uncertainty is 0.0002.

4.3 The Usefulness of the Methodology in Image Classification

The last limiting aspect we discuss regards the usefulness of the uncertainty measure obtained by the Delta method in deep learning image classification. In **Paper 2** we demonstrated that false positives have on average a higher level of predictive uncertainty than true positives. However, using this fact for the better good has shown to be difficult: we have not managed to somehow boost[1] the classification accuracy based on the uncertainty measure nor have we found any other obviously useful applications. This finding is controversial as predictive uncertainty usually is framed as the holy grail when it comes to out-of-distribution (OoD) example detection. While we saw in **Paper 2** that the predictive uncertainty can be high for OoD examples, one can per definition of the Delta method readily construct OoD examples with a class probability of one, and a belonging predictive uncertainty of zero. This can be seen by inspecting Equation (2.3): the predictive uncertainty (as estimated by the Delta method) will always be zero as long as the class probability is either equal to zero or one. The explanation for this phenomenon is attributed to the softmax activation function whose gradient (i.e. sensitivity F) will always be weighted by a quantity which is negative quadratic in probability (i.e. $\hat{y}(1-\hat{y})$).

To further illustrate this, we now demonstrate how to construct an OoD example with a certain class probability close to one, and with a belonging predictive uncertainty close to zero using the MNIST LeNet defined in **Paper 2**. We make use of TensorFlow and flip things around from the standard perspective: after having trained the network as usual, we let x_0 be a variable image while keeping the parameter estimate $\hat{\omega}$ fixed. We initialize x_0 with random uniform noise, and minimize the uncertainty score $\sigma_{\text{score}}(x_0)$ defined in **Paper 2** with respect to x_0. We also constrain the pixels of x_0 to stay within

[1]For example by swapping the classification in terms of the highest and next highest predicted probability when the predictive uncertainty of the former exceeds a predefined threshold.

the normalized range (i.e. between zero and one). The resulting image from this simple constrained optimization procedure is shown in Figure 4.1. Clearly the resulting image is OoD as it does not (at all) resemble any digit in the range zero to nine. However, the predicted class '8' probability for this image is still 0.9998, and the belonging predictive uncertainty is 0.0002.

What can be conclude from this experiment? As long as the class probability of a prediction is overconfident (i.e. close to zero or one), the predictive epistemic uncertainty estimated by the Delta method is forced to be close to zero. This accounts for both in-distribution and OoD examples, and it is therefore not reliable to use the Delta method and the predictive epistemic uncertainty to detect all OoD images.

A natural question is now to ask whether this is just a weakness of the Delta method, or if the overconfidence is an intrinsic property of all deep learning classifiers? If the latter turns out to be true, it could make uncertainty quantification in the deep learning classification less attractive in general. However, we round off this discussion by demonstrating that also the Bootstrap and the Bayesian equivalent of our Delta method approximation, namely the Laplace sampler introduced in Section 2.2.1, fails to fully fix the overconfidence problem. We use the same MNIST LeNet and the same OoD image as above (Figure 4.1) and use $B = 100$ DRWI Bootstrap replicates. This leads to a predicted class '8' probability of 0.9993 and a predictive uncertainty of 0.0003 for the Bootstrap. Furthermore, for the Laplace sampler, we draw $S = 1,000$ samples from the posterior using Equation (2.6). We evaluate (2.7) and get the following results: the predicted class '8' probability is now marginally reduced to 0.9973, and the belonging predictive uncertainty has increased slightly to 0.0053.

We have behind the scene also explored the so-called 'red flag'-application of predictive uncertainty in the image classification setting whose story goes like: if the predictive uncertainty level exceeds a predefined threshold, a red flag is raised and the example is sent to a human expert for evaluation. However, based on the MNIST and CIFAR-10 LeNets, we found no evidence that it is advantageous to use the predictive epistemic uncertainty estimated by the Delta method as opposed to standard probability in our experiments. We invite the public to explore our released MNIST/CIFAR-10 predictive epistemic uncertainty datasets contained in the `pyDeepDelta` Github repository [Nilsen, 2018-2021a]. Documentation on how to read and interpret these datasets is included in the file `pydeepdelta_predictive_uncertainty_datasets.ipynb`.

4.4 Absolute vs. Relative Predictive Uncertainty

As we point out in **Paper 2**, the *absolute level* of the predictive uncertainty for the ResNets is larger than for the LeNets, and exceeds the theoretical maximum standard deviation of 0.5 for softmax-based neural networks. In particular, this highlights that there are no guarantees (due to the approxmations involved) that the Delta method will correctly quantify absolute predictive uncertainty levels. This has several consequences: a) the absolute level of predictive uncertainty must be used carefully, and b) comparing the level of predictive uncertainty for different models can be challenging. However, within the same model, we observe that the *relative level* of predictive uncertainty is preserved by the Delta method. This is evident by several means: a) the high R^2s in **Paper 3**, b) the raised level for false positives regardless of the model (**Paper 2**), and c) that meaningful rankings based on the uncertainty score still can be obtained (also **Paper 2**). These observations are interesting, because in the classification setting — the relative predictive uncertainty (among images) is arguably more important than the absolute counterpart. For example, by a preserved relative predictive uncertainty we can state that within model M the image X has a greater predictive uncertainty than image Y. In contrast, knowing the absolute uncertainty level for image X or Y in model M can be less important. However, as a consequence, in the regression setting (although we have not explored this territory), the absolute uncertainty levels estimated by the Delta method must be used carefully.

As a final remark, we note that the Monte Carlo based Laplace sampling algorithm building on **Paper 2**, developed in Chapter 2, Section 2.2.1, will ensure that the absolute predictive uncertainty levels under classification at least will stay within theoretical maximum bounds. This is simply because the variance estimate (2.7) is calculated as the sample variance of the normalized model outputs.

4.5 Future Work

In the following, we discuss ideas which may improve the computational efficiency of our methodology, and highlight topics which could be explored to extend its usefulness in deep learning.

The stochastic Lanczos algorithm [Lin et al., 2016; Yao et al., 2020] and its variants can potentially be adapted to our setting in order to improve the computational efficiency of Pearlmutter's technique, i.e. the Hessian vector products. We have experimented by using small stochastic mini-batches of the training dataset with size $B \ll N$ when evaluating the Hessian vector products (i.e. analogously to stochastic gradient descent) and find that the convergence is slower (requires more iterations S) but as $B \ll N$ the net computational time is typically reduced. However, we also find that this introduces a bias in terms of that the Hessian eigenvalue spectra are overestimated, but that their corresponding variance is very low (i.e. does not change much for different random seeds).

As we saw in Chapter 2, the Laplace Approximation and in particular the Kronecker-factored [Botev et al., 2017; Kristiadi et al., 2020; Lee et al., 2020; Martens and Grosse, 2015; Ritter et al., 2018] Laplace Approximation and its variants are closely related to the Delta method. Potential synergies between the methodologies should be explored. By using our released software [Nilsen, 2018-2021a], it would be straight forward to assume layer-wise independence of the parameters (or just simply consider the last layer) and thus only combine the diagonal blocks (or just the last-layer block) of the Fisher to further increase the computational efficiency. As an example, the computational time complexity of a block-diagonal approximation of the Hessian estimator in **Paper 2** will be reduced to $O(\max_l\{S^{(l)}P^{(l)}\}N)$ where $S^{(l)}$ and $P^{(l)}$ denotes the number of Lanczos steps for layer l and the number of parameters in layer l, respectively. This approach also paves the way for studying the actual impact of assuming a block-diagonal structure of the Fisher (or just considering the last layer).

In Chapter 2, Section 2.2.1 we introduced an efficient Laplace Approximation based Monte Carlo sampler (the Bayesian equivalent of the Delta method) based on the methodology in **Paper 1** and **Paper 2**. Beyond providing an efficient implementation in [Nilsen, 2018-2021a] (`pydeepdelta_sampler_demo.ipynb`) and the small OoD demonstration in the very end of Section 4.3, we have not explored whether this approach can be beneficial or not. This calls for further experiments. In particular, it could be interesting to extend the approach with importance sampling [Skaug and Fournier, 2006].

In **Paper 2**, we mostly focus on Equation (10) which is the standard deviation of the T_L network outputs, i.e. the predictive uncertainty per class. However, as can be seen by the Equation (8), the full $T_L \times T_L$ covariance matrix of the network outputs can readily be obtained. The research question is, can there be useful information also in the off-diagonal elements of this matrix?

In **Paper 3**, we compared the Delta method with the Bootstrap. Further work in this direction could be to compare the Delta method to some of the other methods outlined in Chapter 2. Moreover, as our work has been focused on the classification task, a natural extension is to see how the framework behaves under deep learning regression.

4.6 Impact of Work

We expect that this research can contribute to bridge the gap between statisticians and machine/deep learning researchers and practitioners. We have explored a range of interconnected topics from both these fields, and provided computationally viable solutions in order to try out the Delta method in deep learning. The main consequence of this work can thus be seen as a way of enabling faster and deeper research in the same direction. Although we have demonstrated the practicality of our methodology, the emphasis has always been on the computational aspects, and so the real value of applying the methodology has yet to be uncovered.

At the time of writing, several independent authors have cited the papers included in this dissertation; a computational method for empirical characterization of the training loss level-sets of deep neural networks is discussed in [Tahir and Katz, 2020], a study of gradients flow pathologies in physics-informed neural networks can be found in [Wang et al., 2021], the implicit bias effect of deep linear networks for binary classification using the logistic loss in the large learning rate regime is characterized in [Huang et al., 2020], while [van den Dool, 2020] proposes a mathematical definition for the concept of an explanation in the context of understanding deep learning decisions, [Gundersen et al., 2020] explores the use of deep learning and variational inference in Carbon Capture and Storage (CCS) monitoring. A Long Short Term Memory (LSTM) cell parameter optimization procedure is proposed by [Chia et al., 2021].

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine learning*, 50(1):5–43, 2003.

R. Anil, V. Gupta, T. Koren, K. Regan, and Y. Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

S. Becker and Y. Le Cun. Improving the Convergence of Back-Propagation Learning with Second-Order Methods, 1988.

Y. Bengio. *Learning deep architectures for AI.* Now Publishers Inc, 2009.

C. M. Bishop. *Pattern recognition and Machine Learning.* Springer, 2006.

D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

W. M. Bolstad and J. M. Curran. *Introduction to Bayesian statistics.* John Wiley & Sons, 2016.

A. Botev, H. Ritter, and D. Barber. Practical gauss-newton optimisation for deep learning. In *International Conference on Machine Learning*, pages 557–565. PMLR, 2017.

L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. SIAM Rev., vol. 60, no. 2, pp. 223-311, 2018.

J. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems*, 2, 1989.

S. Brooks, A. Gelman, G. Jones, and X.-L. Meng. *Handbook of Markov Chain Monte Carlo*. CRC press, 2011.

H. Cardot and D. Degras. Online Principal Component Analysis in High Dimension: Which Algorithm to Choose? https://arxiv.org/abs/1511.03688 arXiv:1511.03688 [stat.ML], 2015.

M. Caron, P. Bojanowski, A. Joulin, and M. Douze. Deep clustering for unsupervised learning of visual features. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 132–149, 2018.

Z. C. Chia, K. H. Lim, and T. P. L. Tan. Two-phase switching optimization strategy in lstm model for predictive maintenance. In *2021 International Conference on Green Energy, Computing and Sustainable Technology (GECOST)*, pages 1–6. IEEE, 2021.

E. Daxberger, A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig. Laplace Redux–Effortless Bayesian Deep Learning. *arXiv preprint arXiv:2106.14806*, 2021.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

J. L. Devore, K. N. Berk, and M. A. Carlton. *Modern mathematical statistics with applications, 2nd edition*. Springer, 2012.

M. Drozdzal, E. Vorontsov, G. Chartrand, S. Kadoury, and C. Pal. The importance of skip connections in biomedical image segmentation. In *Deep learning and data labeling for medical applications*, pages 179–187. Springer, 2016.

B. Efron. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer, 1992.

D. A. Freedman. On the So-Called "Huber Sandwich Estimator" and "Robust Standard Errors". https://www.jstor.org/stable/27643806 The American Statistician, Vol. 60, No. 4 (Nov., 2006), pp. 299-302, 2006.

J. Friedman, T. Hastie, R. Tibshirani, et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

Y. Gal and Z. Ghahramani. Dropout as a Approximation: Representing Model Uncertainty in Deep Learning. https://arxiv.org/pdf/1506.02142, arXiv:1506.02142v6 [stat.ML], 2016.

J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. Kruspe, R. Triebel, P. Jung, R. Roscher, et al. A survey of uncertainty in deep neural networks. *arXiv preprint arXiv:2107.03342*, 2021.

J. E. Gentle. *Computational statistics*, volume 308. Springer, 2009.

C. J. Geyer. 5601 Notes: The Sandwich Estimator. http://www.stat.umn.edu/geyer/5601/notes/sand.pdf, 2003.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

I. Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. http://www.deeplearningbook.org, MIT Press, 2016.

I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.

R. Grosse. Lecture 2: Taylor Approximations. https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2021/readings/L02_Taylor_approximations.pdf, 2020.

K. Gundersen, G. Alendal, A. Oleynik, and N. Blaser. Binary Time Series Classification with Bayesian Convolutional Neural Networks When Monitoring for Marine Gas Discharges. *Algorithms*, 13(6):145, 2020.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

P. Hennig. Fast probabilistic optimization from noisy gradients. In *International conference on machine learning*, pages 62–70. PMLR, 2013.

S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. A field guide to dynamical recurrent neural networks. IEEE Press, 2001.

J. M. V. Hoef. Who Invented the Delta Method? `https://www.researchgate.net/publication/254329376_Who_Invented_the_Delta_Method`, The American Statistician, 66:2, 124-127, 2012.

G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

W. Huang, W. Du, R. Y. Da Xu, and C. Liu. Implicit bias of deep linear networks in the large learning rate phase. *arXiv preprint arXiv:2011.12547*, 2020.

E. Hüllermeier and W. Waegeman. Aleatoric and Epistemic Uncertainty in Machine Learning: An Introduction to Concepts and Methods. `https://arxiv.org/abs/1910.09457`, arXiv:1910.09457v2 [cs.LG], 2020.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

L. Jing and Y. Tian. Self-supervised visual feature learning with deep neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.

A. Kendall and Y. Gal. What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision? `https://arxiv.org/pdf/1703.04977`, arXiv:1703.04977v2 [cs.CV], 2017.

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. *arXiv preprint arXiv:1609.04836*, 2016.

A. Khosravi and D. Creighton. A Comprehensive Review of Neural Network-based Prediction Intervals and New Advances. `https://www.researchgate.net/publication/51534965_Comprehensive_Review_of_Neural_Network-Based_Prediction_Intervals_and_New_Advances`, IEEE Transactions On Neural Networks, Vol. 22, No. 9, 2011.

D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In Proc. 3rd Int. Conf. Learn. Representations, 2014.

W. Krauth. *Introduction to Monte Carlo algorithms*. Springer, 1998.

A. Kristiadi, M. Hein, and P. Hennig. Being Bayesian, even just a bit, fixes overconfidence in ReLU networks. In *International Conference on Machine Learning*, pages 5436–5446. PMLR, 2020.

A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Citeseer, 2009.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25: 1097–1105, 2012.

A. Krogh and J. Hertz. A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4, 1991.

S. Kullback. *Information theory and statistics*. Courier Corporation, 1997.

A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, pages 1–26, 2020.

Y. Kwon, J.-H. Won, B. J. Kim, and M. C. Paik. Uncertainty quantification using Bayesian neural networks in classification: Application to biomedical image segmentation. *Computational Statistics & Data Analysis*, 142:106816, 2020.

B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. `https://arxiv.org/pdf/1612.01474`, arXiv:1612.01474v3 [stat.ML], 2017.

L. Le Cam. *Maximum likelihood: an introduction*. JSTOR, 1990.

N. Le Roux and A. W. Fitzgibbon. A fast natural Newton method. In *ICML*, 2010.

Y. LeCun, L. D. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard, et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective*, 261(276):2, 1995.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

J. Lee, M. Humt, J. Feng, and R. Triebel. Estimating model uncertainty of neural networks in sparse information form. In *International Conference on Machine Learning*, pages 5702–5713. PMLR, 2020.

K. Lee, K. Lee, H. Lee, and J. Shin. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. *arXiv preprint arXiv:1807.03888*, 2018.

A. Levy and M. Lindenbaum. Sequential KarhunenLoeve Basis Extraction and its Application to Images. `http://www.cs.technion.ac.il/~mic/doc/skl-ip.pdf` IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 9, NO. 8, 2000.

Y. Li, J. M. Hernández-Lobato, and R. E. Turner. Stochastic expectation propagation. *arXiv preprint arXiv:1506.04132*, 2015.

L. Lin, Y. Saad, and C. Yang. *Approximating spectral densities of large matrices*, volume 58. SIAM, 2016.

S. Linnainmaa. *Taylor expansion of the accumulated rounding error*, volume 16. Springer, 1976.

D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

A. Ly, M. Marsman, J. Verhagen, R. P. Grasman, and E.-J. Wagenmakers. A tutorial on Fisher information. *Journal of Mathematical Psychology*, 80:40–55, 2017.

D. J. MacKay. A practical Bayesian framework for backpropagation networks. `http://www.inference.org.uk/mackay/PhD.html#PhD`, Neural Computation, 4(3):448472, 1992., 1992a.

D. J. MacKay. The evidence framework applied to classification networks. *Neural computation*, 4(5):720–736, 1992b.

J. Martens and R. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

S. M. Murray. An exploratory analysis of multi-class uncertainty approximation in bayesian convolutional neural networks, 2018.

P. Nagarajan and G. Warnell. Deterministic Implementations for Reproducibility in Deep Reinforcement Learning. `https://arxiv.org/abs/1809.05676`, arXiv:1809.05676 [cs.AI], 2019.

R. M. Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

R. M. Neal et al. MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo*, 2(11):2, 2011.

W. K. Newey and D. McFadden. Handbook of Econometrics. `https://www.sciencedirect.com/science/article/pii/S1573441205800054`, 1994.

M. A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, 2015.

G. K. Nilsen. pyDeepDelta: A TensorFlow Module Implementing the Delta Method in Deep Learning Classification. `https://github.com/gknilsen/pydeepdelta.git`, 2018-2021a.

G. K. Nilsen. pyHessian: A TensorFlow module with functionality for efficient computation of Hessian matrices. `https://github.com/gknilsen/pyhessian.git`, 2018-2021b.

G. K. Nilsen, A. Z. Munthe-Kaas, H. J. Skaug, and M. Brun. Efficient Computation of Hessian Matrices in TensorFlow. `https://arxiv.org/abs/1905.05559`, arXiv:1905.05559v1 [cs.LG], 2019.

G. K. Nilsen, A. Z. Munthe-Kaas, H. J. Skaug, and M. Brun. Epistemic uncertainty quantification in deep learning classification by the delta method. *Neural Networks*, 2021a. URL `https://www.sciencedirect.com/science/article/pii/S0893608021004056`.

G. K. Nilsen, A. Z. Munthe-Kaas, H. J. Skaug, and M. Brun. A Comparison of the Delta Method and the Bootstrap in Deep Learning Classification. `http://arxiv.org/abs/2107.01606`, arXiv:2107.01606 [cs.LG], 2021b.

I. Osband. Risk versus Uncertainty in Deep Learning: Bayes, Bootstrap and the Dangers of Dropout. `http://bayesiandeeplearning.org/2016/papers/BDL_4.pdf`, NIPS Workshop on Bayesian Deep Learning, 2016.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

B. A. Pearlmutter. Fast Exact Multiplication by the Hessian. `http://www.bcl.hamilton.ie/~barak/papers/nc-hessian.pdf`, Neural Computation, 6 (1) (1994), pp. 147-160, 1994.

T. S. Perry. Andrew Ng X-Rays the AI Hype. `https://spectrum.ieee.org/view-from-the-valley/artificial-intelligence/machine-learning/andrew-ng-xrays-the-ai-hype`, IEEE Spectrum, 2021.

S. Ransbotham, D. Kiron, P. Gerbert, and M. Reeves. Reshaping business with artificial intelligence. `https://sloanreview.mit.edu/projects/reshaping-business-with-artificial-intelligence/`, MIT Sloan Management Review, 2017.

H. Ritter, A. Botev, and D. Barber. A scalable Laplace approximation for neural networks. In *6th International Conference on Learning Representations, ICLR 2018-Conference Track Proceedings*, volume 6. International Conference on Representation Learning, 2018.

H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

T. Salimans and D. P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, 29:901–909, 2016.

P. Schulam and S. Saria. Can You Trust This Prediction? Auditing Pointwise Reliability After Learning. `https://arxiv.org/abs/1901.00403` arXiv:1901.00403 [stat.ML], 2019.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

H. J. Skaug and D. A. Fournier. Automatic approximation of the marginal likelihood in non-Gaussian hierarchical models. *Computational Statistics & Data Analysis*, 51(2): 699–709, 2006.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

M. Stinchombe. Universal approximation using feed-forward networks with nonsigmoid hidden layer activation functions. *Proc. IJCNN, Washington, DC, 1989*, pages 161–166, 1989.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

N. Tahir and G. E. Katz. Numerical Exploration of Training Loss Level-Sets in Deep Neural Networks. *arXiv preprint arXiv:2011.04189*, 2020.

T. Tieleman and G. Hinton. Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. *Technical Report.*, 2017.

L. N. Trefethen and D. B. III. *Numerical Linear Algebra*. Siam, 1997.

W. van den Dool. Understanding Deep Learning Decisions: the Explanatory Vector Decomposition (EVD) method. `https://dspace.library.uu.nl/handle/1874/393405`, Utrecht University Repository, 2020.

A. W. Van der Vaart. *Asymptotic Statistics*, volume 3. Cambridge university press, 2000.

M. J. Wainwright and M. I. Jordan. *Graphical models, exponential families, and variational inference*. Now Publishers Inc, 2008.

S. Wang, Y. Teng, and P. Perdikaris. Understanding and Mitigating Gradient Flow Pathologies in Physics-Informed Neural Networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688. Citeseer, 2011.

J. J. Weng, N. Ahuja, and T. S. Huang. Learning recognition and segmentation of 3-D objects from 2-D images. In *1993 (4th) International Conference on Computer Vision*, pages 121–128. IEEE, 1993.

R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

J. Xie, R. Girshick, and A. Farhadi. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, pages 478–487. PMLR, 2016.

Z. Yao, A. Gholami, K. Keutzer, and M. W. Mahoney. PyHessian: Neural networks through the lens of the Hessian. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 581–590. IEEE, 2020.

Chapter 5

The Papers

Paper 1 [Nilsen et al., 2019]

5.1 Efficient Computation of Hessian Matrices in TensorFlow

Geir K. Nilsen, Antonella Z. Munthe-Kaas, Hans J. Skaug, Morten Brun, arXiv preprint: 1905.05559, https://arxiv.org/abs/1905.05559, 2019, revised 2021.

Efficient Computation of Hessian Matrices in TensorFlow

Geir K. Nilsen[1,2], Antonella Z. Munthe-Kaas[1], Hans J. Skaug[1], and Morten Brun[1]

[1] *Department of Mathematics, University of Bergen*
[2] *geir.kjetil.nilsen@gmail.com*

Abstract

This paper deals with the practical aspects of efficiently computing Hessian matrices in the context of deep learning using the Python programming language and the TensorFlow library. We define a general feed-forward neural network model and show how to efficiently compute two quantities: the cost function's exact Hessian matrix, and the cost function's approximate Hessian matrix, known as the Outer Product of Gradients (OPG) matrix. Furthermore, as the number of parameters P in deep learning usually is very large, we show how to reduce the quadratic space complexity by an efficient implementation based on approximate eigendecompositions.

1 Introduction

The Hessian matrix has a number of important applications in a variety of different fields, such as optimzation, image processing and statistics. Geometrically, the Hessian matrix describes the local curvature of scalar functions $f : \mathbb{R}^P \to \mathbb{R}$, and is for this reason perhaps mostly known in the field of optimization [8]. Nevertheless, the Hessian matrix also has an important role in statistics, since its inverse is related to the powerful concept of uncertainty quantification [9].

In this technical note we mostly focus on the practical aspects of efficiently computing Hessian matrices in the context of deep learning [7] using the Python [10] programming language and the TensorFlow [1] library. We define a general feed-forward neural network model and show how to efficiently compute two quantities: the cost function's exact Hessian matrix, and the cost function's approximate Hessian matrix, known as the Outer Product of Gradients (OPG) matrix. Furthermore, as the number of parameters P in deep learning usually is very large, we show how to reduce the quadratic space complexity by efficient approximate eigendecompositions. Although we here use a feed-fordward neural network architecture to introduce terminology, the theory and implementation presented is still directly applicable on more general neural network architectures using convolutional layers, pooling and regularization.

The paper is organized as follows: In Section 2 we give definitions which will be used throughout the paper. In Section 3 we present the problem statement, and discuss three complications which need to dealt with in order to achieve a successful TensorFlow implementation: 1) `tf.hessians()` is fundamentally inadequate since it only calculates a subset of all the partial derivatives (Section 3.3), 2) computing Hessian matrices essentially requires per-example gradients of the cost function with respect to model parameters, and unfortunately, the differentiation functionality provided by TensorFlow does not support computing gradients with respect to individual examples efficiently [2] (Section 3.1), and 3) when differentiating a function with respect to several variables represented by a list of tensors, the result is also a list of tensors (Section 3.2). In Section 5 we show how to overcome the aforementioned complications and introduce our Python module `pyhessian` [11] which is released as open source licensed under GNU GPL on GitHub. In Section 6 we summarize the paper and give some concluding remarks.

2 Deep Neural Networks

A feed-forward neural network is shown in Figure (1). There are L layers $l = 1, 2, ..., L$ with T_l neurons in each layer. The input layer $l = 1$, is represented by the input vector $x_n = \begin{bmatrix} x_{n,1} & x_{n,2} & \dots & x_{n,T_1} \end{bmatrix}^T$ where $n = 1, 2, ..., N$ is the input index. Furthermore, there are $L-2$ dense hidden layers, $l = 2, 3, ..., L-1$, and a dense output layer $l = L$, all represented by weight matrices $W^{(l-1)} \in \mathbb{R}^{T_l \times T_{l-1}}$, bias vectors $b^{(l)} \in \mathbb{R}^{T_l}$ and vectorized activation functions $\sigma^{(l)}$.

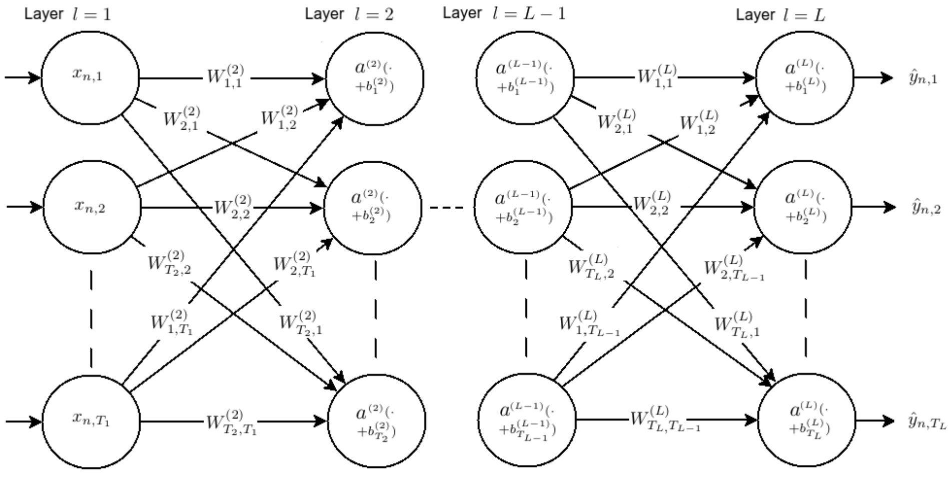


Figure 1: A Feed-Forward Neural Network with Dense Layers

Let the cost function C coincide with TensorFlow's built-in softmax cross-entropy function[1],

$$C = \frac{1}{N} \sum_{n=1}^{N} C_n(y_n, \hat{y}_n) \tag{1}$$

$$= \frac{1}{N} \sum_{n=1}^{N} \left(- \sum_{m=1}^{T_L} y_{n,m} \log \hat{y}_{n,m} \right). \tag{2}$$

It is defined as the average of N per-example cross-entropy cost functions $C_n(y_n, \hat{y}_n)$, where y_n represents the one-hot target vector for the nth

[1]TensorFlow API r1.13: tf.losses.softmax_cross_entropy()

$$\hat{y}_n = f(x_n, \omega) = \sigma^{(L)}(W^{(L-1)}\sigma^{(L-1)}(\cdots\sigma^{(2)}(W^{(1)}x_n + b^{(2)}) + \cdots) + b^{(L)}) \quad (3)$$

example, and where $\hat{y}_n$ represents the corresponding prediction vector. The prediction vector is obtained by evaluating the model function (3) using the input vector x_n and a flat vector of model parameters $\omega \in \mathbb{R}^P$ defined by

$$\omega = \begin{bmatrix}\omega_1 & \omega_2 & \dots & \omega_P\end{bmatrix}^T \quad (4)$$
$$= \underset{l=2,3,\dots,L}{\text{flatten}}(W^{(l-1)}, b^{(l)}). \quad (5)$$

The function flatten$(\cdot)$ denotes a row-wise flattening operation to transform the collection of model parameters represented by the weight matrices $W^{(l-1)}$ and bias vectors $b^{(l)}, l = 2, 3, ..., L$ into a flat column vector of dimension $P = T_1T_2 + T_2 + \ldots + T_{L-1}T_L + T_L$. Further, the activation function in the output layer is the vectorized softmax function

$$\sigma^{(L)}(z) = \text{softmax}(z) \quad (6)$$
$$= \frac{\exp(z)}{\sum_{m=1}^{T_L}\exp(z_m)}, \quad (7)$$

where $z \in \mathbb{R}^{T_L}$, and where $\exp(\cdot)$ denotes the vectorized exponential function. Finally, training of the neural network can be defined as finding an 'optimal' parameter vector $\hat{\omega}$ by minimizing the cost function (1),

$$\hat{\omega} = \arg\min_{\omega \in \mathbb{R}^P} C(\omega). \quad (8)$$

3 Computing Hessian Matrices in TensorFlow

Given the cost function C defined in Section 2, the Hessian matrix $H \in \mathbb{R}^{P\times P}$ is defined[2]

$$H = \left.\frac{\partial^2 C}{\partial\omega\partial\omega^T}\right|_{\omega=\hat{\omega}} \quad (9)$$
$$= \frac{1}{N}\sum_{n=1}^{N}\left.\frac{\partial^2 C_n}{\partial\omega\partial\omega^T}\right|_{\omega=\hat{\omega}}. \quad (10)$$

The approximation to the Hessian matrix, known as the Outer Product of Gradients (OPG) matrix $G \in \mathbb{R}^{P\times P}$, is defined

$$G = \frac{1}{N}\sum_{n=1}^{N}\left.\frac{\partial C_n}{\partial\omega}\frac{\partial C_n}{\partial\omega}^T\right|_{\omega=\hat{\omega}} \quad (11)$$
$$\neq \left.\frac{\partial C}{\partial\omega}\frac{\partial C}{\partial\omega}^T\right|_{\omega=\hat{\omega}}. \quad (12)$$

Letting $J = \begin{bmatrix}\frac{\partial C_1}{\partial\omega} & \frac{\partial C_2}{\partial\omega} & \dots & \frac{\partial C_N}{\partial\omega}\end{bmatrix}$, yields

$$G = \left.\frac{1}{N}J^TJ\right|_{\omega=\hat{\omega}}. \quad (13)$$

We notice that H in Equation (10) is formed by summing over N per-example Hessian matrices, and that G in Equation (11) is formed by summing over N per-example OPG matrices. We also note that H can be obtained by differentating the cost function directly, whereas this property does not hold for G as seen by (12). Finally, we note that G can be written as a per-example cost Jacobian matrix product (13).

In order to proceed, we now need to consider three complications regarding gradients and Hessians in TensorFlow: the limitations of TensorFlow's built-in `tf.hessians()` function is discussed in Section 3.3, per-example gradients will be discussed in Section 3.1, and gradient representation will be discussed in Section 3.2.

[2]The notation used means that $H_{i,j} = \left.\frac{\partial^2 C}{\partial\omega_i\partial\omega_j}\right|_{\substack{\omega_i=\hat{\omega}_i \\ \omega_i=\hat{\omega}_i}}$

3.1 Per-Example Gradients

A per-example gradient of the cost function with respect to model parameters means to differentiate C_n in (10) and (11) with respect to model parameters for a single example n. However, when TensorFlow compute gradients (e.g. `tf.gradients()`) it performs back propagation, which never actually computes the per-example gradients, but instead directly obtains the sum of per-example gradients. To see what this means, consider the following dummy multiple linear regression model (for simplicity with no bias term):

```
In [1]: import tensorflow as tf
In [2]: import numpy as np
In [3]: W = tf.Variable([3., 4., 5., 2.])
In [4]: X = tf.placeholder('float32', shape=(None, 4))
In [5]: yhat = tf.tensordot(X, W, axes = 1)
In [6]: init = tf.global_variables_initializer()
In [7]: sess = tf.InteractiveSession()
In [8]: sess.run(init)
```

We have model parameters represented by the variable tensor `W` (`In [3]`), and we use the placeholder tensor `X` (`In [4]`) as the model input. For simplicity, we do not define a cost function here, but instead conduct several differentiation experiments directly on the scalar model function `yhat` (`In [5]`) with $N = 2$:

```
 In [9]: sess.run(yhat, feed_dict={x:np.array([[1.,2.,3.,4.],
                                              [2.,3.,4.,5.]])})
Out [1]: array([34., 48.], dtype=float32)
```

We get back two values (`Out [1]`) corresponding to the two inner products as expected. We now take the gradient of the model function with respect to the model parameters for a single example:

```
In [10]: sess.run(tf.gradients(yhat, W),
                  feed_dict={X:np.array([[1.,2.,3.,4.]])})
Out [2]: [array([1., 2., 3., 4.], dtype=float32)]
```

We get back the per-example gradient as expected (`Out [2]`). We do the same for the second example:

```
In [11]: sess.run(tf.gradients(yhat, W),
                  feed_dict={X:np.array([[2.,3.,4.,5.]])})
Out [3]: [array([2., 3., 4., 5.], dtype=float32)]
```

But when we try to feed two examples:

```
In [12]: sess.run(tf.gradients(yhat, W),
                  feed_dict={X:np.array([[1.,2.,3.,4.],
                                         [2.,3.,4.,5.]])})
Out [4]: [array([3., 5., 7., 9.], dtype=float32)]
```

we notice that we do not get back two per-example gradients, but rather the sum of the two per-example gradients (`Out [4]`). The important observation is here that in order to obtain per-example gradients we seemingly need to run `tf.gradients()` once per example, which in turn is well known to be very inefficient when N grows large. We will get back to this and discuss solutions in Sections (5.1) and 5.2).

3.2 Gradient Representation

In practice, the P model parameters are represented by a list of tensors (e.g. `[tf.Variable(),...]`) corresponding to the different layers of the model architecture. On the other hand, the Hessian matrix is only one `(P, P)`-shaped tensor (matrix) formed by every single variable element contained in the list of variable tensors.

When differentiating a function represented by a computational graph with respect to some variable(s) in that graph, the variable tensors we pass to the differentiation function (`tf.gradients()`) must be kept in their original form as upon defining the graph. One can still pass on the whole collection of variables as a list to get hold of the full gradient, but the result will not be a flat gradient vector – it will rather be a list of sub-gradients represented by multiple tensors. This means that in order to end up with the `(P, P)`-shaped Hessian matrix we want, we need to keep all the variables in a list during differentiation, and only afterwards reshape the result into the desired flat form.

3.2.1 Flattening of Gradients

To illustrate the concept of lists of sub-gradients vs. flat gradients, consider a dummy multinomial logistic regression model:

```
In [13]: import tensorflow as tf
In [14]: T1 = 64
In [15]: T2 = 32
In [16]: P = T1*T2 + T2 # Total number of model parameters
In [17]: W = tf.Variable(tf.ones((T1, T2)), 'float32')
In [18]: b = tf.Variable(tf.ones((T2,)), 'float32')
In [19]: params = [W, b]
In [20]: params
Out [5]: [<tf.Variable 'Variable...' shape=(64, 32) ...>,
          <tf.Variable 'Variable...' shape=(32,) ...]
In [21]: X = tf.placeholder(dtype='float32', shape=(None, T1))
In [22]: y = tf.placeholder(dtype='float32', shape=(None, T2))
In [23]: def model_fun(X, params):
             return tf.add(tf.matmul(X, params[0]), params[1])
In [24]: yhat_logits = model_fun(X, params)
In [25]: yhat = tf.nn.softmax(yhat_logits)
In [26]: def cost_fun(y, yhat_logits, params):
             return tf.losses.softmax_cross_entropy(y,
                                                      yhat_logits)
In [27]: cost = cost_fun(y, yhat_logits, params)
```

We thus have model parameters `W` (`In [17]`) and `b` (In [18]) with shapes `(T1, T2)` and `(T2,)`, respectively. We can differentiate the cost function represented by the tensor `cost` (`In [27]`) with respect to the individual variables, or the full list `params` (`In [19]`):

```
In [28]: tf.gradients(cost, W)
Out [6]: [<tf.Tensor 'gradients...' shape=(64, 32) ...>]
```

```
In [29]: tf.gradients(cost, b)
Out [7]: [<tf.Tensor 'gradients...' shape=(32,) ...>]
In [30]: tf.gradients(cost, params)
Out [8]: [<tf.Tensor 'gradients...' shape=(64, 32) ...>,
          <tf.Tensor 'gradients...' shape=(32,) ...>]
```

But if we try to reshape our parameters into a flat vector and then differentiate:

```
 In [31]: params_flat = tf.concat([tf.reshape(W, [-1]), b],
                                  axis=0)
 In [32]: params_flat
 Out [9]: <tf.Tensor 'concat...' shape=(2080,) ...>
 In [33]: tf.gradients(cost, params_flat)
Out [10]: [None]
```

We get `[None]` (`Out [10]`) because the new tensor `params_flat` (`In [31]`) is not part of the `cost` function graph (`In [27]`). We solve the issue by first differentiating with respect to the full list, and then flattening the resulting tensor:

```
 In [34]: grads = tf.gradients(cost, params)
 In [35]: grads
Out [12]: [<tf.Tensor 'gradients_...' shape=(64, 32) ...>,
           <tf.Tensor 'gradients_...' shape=(32,) ...>]
 In [36]: grads_flat = tf.concat([tf.reshape(grads[0],[-1]),
                                  grads[1]],
                                 axis=0)
 In [37]: grads_flat
Out [13]: <tf.Tensor 'concat...' shape=(2080,) dtype=float32>
```

3.3 The built-in TensorFlow function `tf.hessians()`

The fundamental question is, why can we not simply use the built-in TensorFlow function `tf.hessians()`? To see why, consider the following:

```
 In [38]: tf.hessians(cost, params)
Out [14]: [<tf.Tensor 'Reshape_...' shape=(64, 32, 64, 32) ...>,
           <tf.Tensor 'Reshape_...' shape=(32, 32) ...>]
```

We observe that we get back two tensors (`Out [14]`). Let us name the two H_U and H_L, respectively. Their respective shapes are `(T1, T2, T1, T2)` and `(T2, T2)`. Firstly, if we reshape H_U into a `(T1*T2, T1*T2)`-shaped tensor, it will correspond to the full Hessian's upper block diagonal matrix $\in \mathbb{R}^{T_1T_2 \times T_1T_2}$. Secondly, the tensor H_L corresponds to the full Hessian's lower block diagonal matrix $\in \mathbb{R}^{T_2 \times T_2}$. In other words, we get no information about the full Hessian's two off-diagonal block matrices $\in \mathbb{R}^{T_1T_2 \times T_2}$ and $\mathbb{R}^{T_2 \times T_1T_2}$. Equation (14) illustrates the concept.

$$H = \begin{bmatrix} H_U \in \mathbb{R}^{T_1T_2 \times T_1T_2} & ? \in \mathbb{R}^{T_1T_2 \times T_2} \\ ? \in \mathbb{R}^{T_2 \times T_1T_2} & H_L \in \mathbb{R}^{T_2 \times T_2} \end{bmatrix} \tag{14}$$

The two missing off-diagonal block matrices[3] represented by question marks

[3]The two matrices are equal up to transposition, since H is symmetric

in Equation (14) correspond to the partial derivatives involving variable entities from different tensors in the parameter list `params` (`In [19]`). The same principle applies for all `params` with `len(params)` > 1.

4 Approximate Hessian Eigendecompositions

In deep learning, the number of parameters P is usually so large that the full Hessian matrix will be prohibitively expensive to compute and store. In this section we present methodology addressing the issue in terms of approximate eigendecompositions based on K eigenpairs. Thus leading to a space complexity of $O(KP)$ rather than $O(P^2)$. As the time complexity is somewhat more involved, we leave this discussion for Sections 5.3 and 5.4.

4.1 Low-rank Approximation

A low-rank approximation of the Hessian matrix can be obtained by a eigendecomposition utilizing only K eigenpairs corresponding to the K largest eigenvalues of H (or G),

$$\widetilde{H} = Q\Lambda Q^T \in \mathbb{R}^{P\times P}, \qquad (15)$$

where $Q \in \mathbb{R}^{P\times K}$ is the matrix whose kth column is the eigenvector q_k of H (or G), and $\Lambda \in \mathbb{R}^{K\times K}$ is the diagonal matrix whose elements are the corresponding eigenvalues, $\Lambda_{kk} = \lambda_k$. We assume that the eigenvalues are algebraically sorted so that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_K$.

4.2 Full-rank Approximation

A full-rank approximation of the Hessian matrix can be obtained by an extrapolation of its smallest eigenvalues. Assuming that $\lambda_{K+1} = \lambda_{K+2} = \ldots = \lambda_P = \widetilde{\lambda} > 0$, a full-rank approximation is given by

$$\widetilde{\widetilde{H}} = \widetilde{H} + \widetilde{\lambda}(I - QQ^T) \in \mathbb{R}^{P\times P}, \quad (16)$$

where $\widetilde{H}$ is the low-rank approximation (15) and where we have used that Q is an orthonormal basis. Details can be found in the Appendix 7.2. One particular choice for $\widetilde{\lambda}$ is to set it equal to the smallest eigenvalue in the low-rank approximation, e.g. $\widetilde{\lambda} = \lambda_K$.

5 Implementation

We will now address how to overcome the basic complications discussed in Sections 3.3, 3.1 and 3.2. The current section is divided into four parts: we first discuss how to compute the matrix H in Equation (10), and afterwards move on to the matrix G in Equation (11). Finally, in Sections 5.3 and 5.4 we address how to compute the aforementioned approximate eigendecompositions of both H and G.

5.1 Computing H

We compute the matrix H based on Hessian vector products [6]. A practial implementation of Equation (10) is essentially to form P Hessian vector products using the full set of basis vectors in $\mathbb{R}^P$. As a bonus, the resulting implementation can easily be paralellized because the columns of the Hessian matrix can be computed independently.

In the following we describe the essential parts of this paper's accompanying Python module `pyhessian` [11]. The Hessian vector product function `get_Hv_op(v)` can be described as follows:

1. Differentiates the cost function with respect to the model param-

eters contained in the list `params` and flattens the result

2. Performs elementwise multiplication of the flattened gradient and the vector `v`; `tf.stop_gradient()` ensures that `v` is treated as a constant during differentiation. This is important if the vector `v` is a function of the model parameters ω.

3. Differentiates the resulting elementwise vector product with respect to the model parameters (to get second order derivatives) and flattens the result. As this step can appear subtle, see the Appendix 7.1 for a rigorous derivation.

Note that the function `get_Hv_op(v)` uses the function `flatten()` which is based on the insights from Section 3.2.1 and the mathematical operation defined in Equation (5). Furthermore, we have defined a parallellized function `get_H_op()` to create the full Hessian matrix operation based on forming `P` Hessian vector products using `get_Hv_op(v)` for all v's in $\mathbb{R}^P$. The function `get_H_op()` sets up a parallelized operation using `tf.map_fn()` to get hold of all the `P` columns of the full Hessian matrix as defined in Equation (10). It works by applying `Hv_op` on all basis vectors in $\mathbb{R}^P$ represented by `tf.eye(self.P, self.P)`, where `P` is the total number of parameters in the model.

The important remark is now to realize that, by definition, the matrix H in Equation (10) is the sum of per-example Hessian matrices. It means that we can directly leverage from the fact that `tf.gradients()` returns the sum of per-example gradients discussed in Section 3.1. In other words, when we run the resulting `H_op` in a graph session, we get per-example Hessians (below `In [43]`) if we feed single examples, and the average of per-example Hessians if we feed more than one example. Thus, we can get a mini-batch (below using size `batch_size_H`) Hessian matrix if we feed a mini-batch (below `In [45]`), or we can obtain the full Hessian matrix directly by feeding the complete training set. However, to avoid excessive memory consumption for large N, we can sum over mini-batch Hessians and divide by the number of mini-batches (`In [46] - In [56]`):

```
In [39]: from pyhessian import HessianEstimator
In [40]: hest = HessianEstimator(...)
In [41]: H_op = hest.get_H_op()
In [42]: # Per-example
In [43]: H = sess.run(H_op, feed_dict={X:[X_train[0]],
                                        y:[y_train[0]]})
In [44]: # Mini-batch
In [45]: H = sess.run(H_op, feed_dict={X:X_train[:batch_size_H],
                                        y:y_train[:batch_size_H]})
In [46]: # Full
In [47]: B = int(N/batch_size_H)
In [48]: H = np.zeros((hest.P, hest.P), dtype='float32')
In [49]: for b in range(B):
In [50]:     H = H + sess.run(H_op,
In [51]:                      feed_dict={ \
In [52]:                      X: X_train[b*batch_size_H: \
In [53]:                                 (b+1)*batch_size_H],
In [54]:                      y: y_train[b*batch_size_H: \
In [55]:                                 (b+1)*batch_size_H]})
In [56]: H = H/B
```

Listing 1: Computing H

5.2 Computing G

Due to the inequality sign in Equation (12), the computation of G (unlike H) cannot exploit the implicit sum of gradients as discussed in Section 3.1. Instead, we will pursuit another efficient technique based on parallized per-example gradients. Although the technique we present here has been reformulated and adapted to our needs, the original implementation idea is to our knowledge originating from the author of [2]. The OPG matrix operation function `get_G_op()` can be described as follows:

1. Creates `batch_size_G` copies of the model parameters
2. Splits the model input variable `X`, and the model output variable `y` into respective lists of `batch_size_G` elements
3. Creates a list of `batch_size_G` elements holding model output tensors resulting from evaluating the model function using respective inputs and parameter copies
4. Creates a list of `batch_size_G` elements holding cost output tensors resulting from evaluating the cost using respective labels, model outputs and parameter copies
5. Stacks up a flat per-example gradient tensor by paralell differentiation of per-example costs with respect to the corresponding model parameter copy
6. Forms the OPG matrix operation by matrix multiplication of per-example cost Jacobians as in Equation (13)

Note that the function `get_G_op()` utilizes the function `flatten()` which is based on the insights from Section 3.2.1 and the mathematical operation defined in Equation (5). Also note that the function `get_G_op()` requires itself to maintain redundant model parameter copies which size scale with `batch_size_G`. To avoid excessive memory consumption, we can sum over mini-batch OPGs and divide by the number of mini-batches (`In [64] - In [68]`):

```
In [57]: hest = HessianEstimator(..., batch_size_G)
In [58]: G_op = hest.get_G_op()
In [59]: # Per-example
In [60]: sess.run(G_op, feed_dict={X:[X_train[0]],
                                   y:[y_train[0]]})
In [61]: # Mini-batch
In [62]: sess.run(G_op, feed_dict={X:X_train[:batch_size_G],
                                   y:y_train[:batch_size_G]})
In [63]: # Full
In [64]: B = int(N/batch_size_G)
In [65]: G = np.zeros((hest.P, hest.P), dtype='float32')
In [66]: for b in range(B):
In [67]:     G = G + sess.run(G_op,
                              feed_dict={ \
                              X: X_train[b*batch_size_G:\
                                         (b+1)*batch_size_G],
                              y: y_train[b*batch_size_G:\
                                         (b+1)*batch_size_G]})
In [68]: G = G/B
```

Listing 2: Computing G

5.3 Computing Eigenpairs of H

The Lanczos iteration [5] can be applied to find $K < P$ eigenvalues (and corresponding eigenvectors) in $O(SNP)$ time and $O(KP)$ space when Pearlmutter's technique [6] is applied inside the iteration. Pearlmutter's technique can simply be described as a procedure based on two-pass back-propagations of complexity $O(NP)$ time and $O(P)$ space to obtain exact Hessian vector products without requiring to keep the full Hessian matrix in memory. The number S denotes the number of Lanczos iterations to reach convergence. Typically the convergence of the Lanczos algorithm will be fast enough so that S is orders of magnitude less than P.

Essentially, we select the number of eigenapirs K and use `LinearOperator` from the `scipy` distribution in combination with the Lanczos implementation `eigsh`, and setup the former to compute Hessian vector products using `get_Hv_op()` from `pyhessian` (`In [69] - In [74]`). The `LinearOperator` (`In [83]`) is initialized with a callback function `Hv()` (`In [75] - In [82]`) where the actual graph session is executed. The `eigsh` argument `which='LA'` (`In [84]`) ensures that the eigenpairs returned corresponds to the algebraically largest eigenvalues of H, and the lines `In [85] - In [87]` sorts the eigenpairs in descending eigenvalue order.

```
In [69]: from scipy.sparse.linalg import LinearOperator
In [70]: from scipy.sparse.linalg import eigsh
In [71]: K = 10
In [72]: hest = HessianEstimator(...)
In [73]: _v = tf.placeholder(shape=(hest.P,), dtype='float32')
In [74]: Hv_op = hest.get_Hv_op(_v)
In [75]: def Hv(v):
In [76]:     B = int(N/batch_size_H)
In [77]:     Hv = np.zeros((hest.P))
In [78]:     Bs = batch_size_H
In [79]:     for b in range(B):
In [80]:         Hv = Hv + sess.run(Hv_op,
                                        feed_dict={X:X_train[b*Bs:\
                                                         (b+1)*Bs],
                                                y:y_train[b*Bs:\
                                                         (b+1)*Bs],
                                                _v:np.squeeze(v)})
In [81]:     Hv = Hv / B
In [82]:     return Hv
In [83]: H = LinearOperator((hest.P, hest.P), matvec=Hv,
                              dtype='float32')
In [84]: L, Q = eigsh(H, k=K, which='LA')
In [85]: sinds = np.flip(np.argsort(L))
In [86]: L = L[sinds]
In [87]: Q = Q[:,sinds]
```

Listing 3: Computing the Eigendecomposition of H

5.4 Computing Eigenpairs of G

For the OPG approximation (12), a slightly different approach can be applied. Since the OPG matrix can be written as a Jacobian matrix product (13), we get by the singular value decomposition that its eigenvectors will be the right singular vectors of the Jacobian, and its eigenvalues the squared singular values

$$\begin{aligned} NG = J^TJ &= (U\Sigma V^T)^T U\Sigma V^T \\ &= V\Sigma U^T U\Sigma V^T \\ &= V\Sigma^2 V^T \end{aligned} \tag{17}$$

However, even the $N \times P$-dimensional Jacobian matrix J is prohibitively expensive to store. Luckily, mini-batches of J can easily be obtained, and so an incremental singular value decomposition [3, 4] can be applied to each mini-batch. The computational cost is thus $O(KNP)$ time and $O(KP)$ space. We select the number of eigenapirs K and use `IncrementalPCA` from the `sklearn` distribution (`In [88] - In [94]`). We then make use of J in (17) which is available via the function `get_J_op()`. The `get_J_op()` implementation is similar to `get_G_op()` except from that it excludes the final matrix product $J^T J$ and just returns J (`In 95`). Essentially, the rest of the details are tied to filling up the buffer `J` in a mini-batch fashion and also ensuring that the number of examples per mini-batch is large enough to support the selected K (`In [96] - In [103]`). Finally, the eigenpairs are computed based on (17) (`In [104]`).

```
In  [88]: from sklearn.decomposition import IncrementalPCA
In  [89]: K = 10
In  [90]: Bs = batch_size_G
In  [91]: hest = HessianEstimator(...)
In  [92]: _N = int(np.ceil(K / Bs))
In  [93]: assert N % _N != 0, 'N must be divisible by \
                               K/batch_size_G!'
In  [94]: ipca = IncrementalPCA(n_components=K, batch_size=Bs*N,
                                copy=False)
In  [95]: J_op = hest.get_J_op()
In  [96]: J = np.zeros((Bs*_N, hest.P), dtype='float32')
In  [97]: B = int(N/Bs)
In  [98]: for b in range(B):
In  [99]:     s1 = Bs*(b%_N)
In [100]:     s2 = Bs*(b%_N+1)
In [101]:     J[s1:s2] = sess.run(J_op,
                          feed_dict={X: X_train[b*Bs:\
                                                (b+1)*Bs],
                                     y: y_train[b*Bs:\
                                                (b+1)*Bs]})
In [102]:     if (b+1) % _N == 0:
In [103]:         ipca.partial_fit(J)
In [104]: L, Q = np.float32(ipca.singular_values_**2 / N),\
                 np.float32(ipca.components_.T)
```

Listing 4: Computing the Eigendecomposition of G

5.5 Low-Rank Approximations

Given the implementations of the eigendecompositions of H and G in Sections 5.3 and 5.4, low-rank approximations can be computed by

```
Q@np.diag(L)@Q.T
```

Listing 5: Computing the Low-Rank Approximation

However, the primary motivation of this approximation is to avoid storing the full Hessian in memory. For example, if the intent is to evaluate $y = x^T H x$ for $x \in \mathbb{R}^P$ then we can use

```
y = (x.T@Q)@np.diag(L)@(Q.T@x)
```

Listing 6: Implicit Application of the Low-Rank Approximation

where we have intentionally introduced superfluous parenthesis to illustrate that this expression avoids to form a full $P \times P$ matrix as an intermediate step.

5.6 Full-Rank Approximations

Given the implementations of H and G in Sections 5.3 and 5.4, full-rank approximations (using $\widetilde{\lambda} = \lambda_K$) can be computed by

```
Q@np.diag(L)@Q.T \
+ L[-1]*(np.eye(hest.P) \
         - Q@Q.T)
```

Listing 7: Computing the Full-Rank Approximation

Analogously to the low-rank example, if we wish to evaluate $y = x^T Hx$ using the full-rank approximation with no intermediate formation of the full Hessian (nor I), we can use

```
y = (x.T@Q)@np.diag(L)@(Q.T@x)\
    + L[-1]*x.T@x \
    - L[-1]*(x.T@Q)@(Q.T@x)
```

Listing 8: Implicit Application of the Full-Rank Approximation

6 Summary and Concluding Remarks

We have presented a practical and efficient TensorFlow implementation for computing Hessian matrices in a deep learning context. The naive methods have a complexity of $O(NP^2)$ time and $O(P^2)$ space where N is the number of examples in the training set and P is the number of parameters in the model. Furthermore, we have introduced means for efficient computation of approximate Hessian eigendecompositions based on K eigenpairs, and shown how these can be applied as both low-rank and full-rank operators. The complexity of the approximate eigendecompositon of the Hessian is $O(SNP)$ and $O(KP)$ space where S represents the number of required Lanczos steps, whereas for the OPG approximation $O(KPN)$ time and $O(KP)$ space. The novelty of the naive methodology presented prominently lies in the implementation technique rather than in the asymptotic bound analysis point of view. As noted by [2], a naive method running back propagation N times with a mini-batch of size 1 is very inefficient because TensorFlow's back propagation implementation will not be able to exploit the parallelism of mini-batch operations by efficient matrix operation implementations. An usage example of the `pyhessian` module [11] applied on a feed-forward neural network TensorFlow model can be found in the included file `pyhessian_example.py`.

7 Appendix

7.1 Derivation of the Hessian Vector Product Implementation

Let y be the product between the Hessian matrix and an arbitrary vector v,

$$y = v^T H(\omega)|_{\omega=\hat{\omega}} \in \mathbb{R}^P, \tag{18}$$

$$H(\omega)|_{\omega=\hat{\omega}} = \begin{bmatrix} \frac{\partial^2 C(\omega)}{\partial^2 \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_p} \\ \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial^2 \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial^2 \omega_p} \end{bmatrix}_{\omega=\hat{\omega}} \in \mathbb{R}^{P\times P}, \quad v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_P \end{bmatrix} \in \mathbb{R}^P, \tag{19}$$

where $C(\omega)$ is the scalar cost function (1), $\omega \in \mathbb{R}^P$ denotes the model parameter vector, and where $\hat{\omega}$ is the point in parameter space where we would like to evaluate the Hessian. The implementation of `get_Hv_op()` in `pyhessian` is as follows

```
y = flatten(tf.gradients(tf.math.multiply(flatten(tf.gradients(C, ω̂)),
                                           tf.stop_gradient(v)),
                params))
```

Listing 9: get_Hv_op() implementation

The inner-most differentiation (e.g. `tf.gradients()`) will return the gradient of the scalar function $C(\omega)$ evaluated at $\omega = \hat{\omega}$, which we will denote by $\nabla_\omega C(\omega)|_{\omega=\hat{\omega}} \in \mathbb{R}^P$. Furthermore, this gradient is multiplied element-wise by the vector v, and we get

$$\nabla_\omega C(\omega) \circ v|_{\omega=\hat{\omega}} = \begin{bmatrix} \frac{\partial C(\omega)}{\partial \omega_1} v_1 \\ \frac{\partial C(\omega)}{\partial \omega_2} v_2 \\ \vdots \\ \frac{\partial C(\omega)}{\partial \omega_P} v_P \end{bmatrix}_{\omega=\hat{\omega}} . \tag{20}$$

Therefore the first argument of the outer-most differentiation (e.g. `tf.gradients()`), will be a vector function rather than a scalar function as was not the case in the inner-most differentiation. Since differentiation of tensors in TensorFlow will evaluate to the <u>sum</u> of the gradients of the individual elements (of the tensor which is differentiated), we get

$$\nabla_\omega \nabla_\omega C(\omega) \circ v|_{\omega=\hat{\omega}} = \begin{bmatrix} \frac{\partial}{\partial \omega_1}\frac{\partial C(\omega)}{\partial \omega_1} v_1 + \frac{\partial}{\partial \omega_1}\frac{\partial C(\omega)}{\partial \omega_2} v_2 + \ldots + \frac{\partial}{\partial \omega_1}\frac{\partial C(\omega)}{\partial \omega_P} v_P \\ \frac{\partial}{\partial \omega_2}\frac{\partial C(\omega)}{\partial \omega_1} v_1 + \frac{\partial}{\partial \omega_2}\frac{\partial C(\omega)}{\partial \omega_2} v_2 + \ldots + \frac{\partial}{\partial \omega_2}\frac{\partial C(\omega)}{\partial \omega_P} v_P \\ \vdots \\ \frac{\partial}{\partial \omega_P}\frac{\partial C(\omega)}{\partial \omega_1} v_1 + \frac{\partial}{\partial \omega_P}\frac{\partial C(\omega)}{\partial \omega_2} v_2 + \ldots + \frac{\partial}{\partial \omega_P}\frac{\partial C(\omega)}{\partial \omega_P} v_P \end{bmatrix}_{\omega=\hat{\omega}} \tag{21}$$

$$= \begin{bmatrix} \frac{\partial^2 C(\omega)}{\partial^2 \omega_1} v_1 + \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_2} v_2 + \ldots + \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_P} v_P \\ \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_1} v_1 + \frac{\partial^2 C(\omega)}{\partial^2 \omega_2} v_2 + \ldots + \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_P} v_P \\ \vdots \\ \frac{\partial^2 C(\omega)}{\partial \omega_P \partial \omega_1} v_1 + \frac{\partial^2 C(\omega)}{\partial \omega_P \partial \omega_2} v_2 + \ldots + \frac{\partial^2 C(\omega)}{\partial^2 \omega_P} v_P \end{bmatrix}_{\omega=\hat{\omega}} \tag{22}$$

$$= \begin{bmatrix} \frac{\partial^2 C(\omega)}{\partial^2 \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_1 \partial \omega_p} \\ \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial^2 \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial \omega_2 \partial \omega_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_1} & \frac{\partial^2 C(\omega)}{\partial \omega_p \partial \omega_2} & \cdots & \frac{\partial^2 C(\omega)}{\partial^2 \omega_p} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_P \end{bmatrix}_{\omega=\hat{\omega}} \tag{23}$$

$$= v^T H(\omega)|_{\omega=\hat{\omega}} \quad \square \tag{24}$$

7.2 Derivation of the Full-rank Approximation

The full eigendecomposition of the Hessian matrix can be written

$$H = Q_L \Lambda_L Q_L^T + Q_R \Lambda_R Q_R^T, \tag{25}$$

where $Q_L \in \mathbb{R}^{P\times K}$ is the matrix whose kth column is the eigenvector q_k of H, and $\Lambda_L \in \mathbb{R}^{K\times K}$ is the diagonal matrix whose elements are the corresponding eigenvalues, $\Lambda_{Lkk} = \lambda_k$. Further, $Q_R \in \mathbb{R}^{P\times(P-K)}$ is the matrix whose kth column is the eigenvector q_{K+k} of H, and $\Lambda_R \in \mathbb{R}^{(P-K)\times(P-K)}$ is the diagonal matrix whose elements are the corresponding eigenvalues, $\Lambda_{Rkk} = \lambda_{K+k}$. We assume that the eigenvalues are algebraically sorted so that $\lambda_1 \geq \lambda_2 \geq \lambda_K \geq \ldots \geq \lambda_P$. Assuming that the eigenvalues $\lambda_{K+1} = \lambda_{K+2} = \ldots = \lambda_P = \widetilde{\lambda} > 0$, we get

$$\widetilde{\widetilde{H}} = Q_L \Lambda_L Q_L^T + Q_R \widetilde{\lambda} I Q_R^T \tag{26}$$

$$= Q_L \Lambda_L Q_L^T + \widetilde{\lambda} Q_R Q_R^T. \tag{27}$$

Since the columns of Q_L and Q_R forms an orthonormal basis, it follows that $I = Q_L Q_L^T + Q_R Q_R^T$, and thus

$$\widetilde{\widetilde{H}} = Q_L \Lambda_L Q_L^T + \widetilde{\lambda}(I - Q_L Q_L^T). \tag{28}$$

Consequently, $\widetilde{\widetilde{H}}$ will be full-rank since all its eigenvalues are greater than zero. □

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from `https://tensorflow.org`

[2] Ian Goodfellow *Efficient Per-Example Gradient Computations* Technical report, Google, Inc, 2015, `http://arxiv.org/abs/1510.01799v2`

[3] A. Levy and M. Lindenbaum *Sequential Karhunen?Loeve Basis Extraction and its Application to Images* IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 9, NO. 8, 2000, `http://www.cs.technion.ac.il/~mic/doc/skl-ip.pdf`

[4] H. Cardot and D. Degras *Online Principal Component Analysis in High Dimension: Which Algorithm to Choose?*, arXiv:1511.03688 [stat.ML], 2015 `https://arxiv.org/abs/1511.03688`

[5] , L. N. Trefethen and D. Bau III *Numerical Linear Algebra, pp. 243-284*, Siam, 1997

[6] Barak A. Pearlmutter *Fast Exact Multiplication by the Hessian* Neural Computation, 1993, `http://www.bcl.hamilton.ie/~barak/papers/nc-hessian.pdf`

[7] Ian Goodfellow and Yoshua Bengio and Aaron Courville *Deep Learn-*

ing. MIT Press, 2016, http://www.deeplearningbook.org

[8] Jorge Nocedal and Stephen Wright *Numerical Optimization* Springer Verlag, 2000, http://www.bioinfo.org.cn/~wangchao/maa/Numerical_Optimization.pdf

[9] Whitney K. Newey and Daniel McFadden *Handbook of Econometrics, Volume 4, Chapter 36 Large sample estimation and hypothesis testing, Pages 2111-2245* Elsevier, 1994, https://www.sciencedirect.com/science/article/pii/S1573441205800054

[10] *Python* https://www.python.org

[11] *pyhessian Python Module* https://github.com/gknilsen/pyhessian.git

Paper 2 [Nilsen et al., 2021a]

5.2 Epistemic Uncertainty Quantification in Deep Learning Classification by the Delta Method

Geir K. Nilsen, Antonella Z. Munthe-Kaas, Hans J. Skaug, Morten Brun, **Published in Neural Networks (Elsevier), October 2021.**

Neural Networks 145 (2022) 164–176

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

Epistemic uncertainty quantification in deep learning classification by the Delta method

Geir K. Nilsen *, Antonella Z. Munthe-Kaas, Hans J. Skaug, Morten Brun

Department of Mathematics, University of Bergen, Norway

ARTICLE INFO

Article history:
Received 28 February 2021
Received in revised form 18 August 2021
Accepted 18 October 2021
Available online 23 October 2021

Keywords:
Uncertainty quantification
Predictive epistemic uncertainty
Neural networks
Deep learning
Hessian
Fisher information

ABSTRACT

The Delta method is a classical procedure for quantifying epistemic uncertainty in statistical models, but its direct application to deep neural networks is prevented by the large number of parameters P. We propose a low cost approximation of the Delta method applicable to L_2-regularized deep neural networks based on the top K eigenpairs of the Fisher information matrix. We address efficient computation of full-rank approximate eigendecompositions in terms of the exact inverse Hessian, the inverse outer-products of gradients approximation and the so-called Sandwich estimator. Moreover, we provide bounds on the approximation error for the uncertainty of the predictive class probabilities. We show that when the smallest computed eigenvalue of the Fisher information matrix is near the L_2-regularization rate, the approximation error will be close to zero even when $K \ll P$. A demonstration of the methodology is presented using a TensorFlow implementation, and we show that meaningful rankings of images based on predictive uncertainty can be obtained for two LeNet and ResNet-based neural networks using the MNIST and CIFAR-10 datasets. Further, we observe that false positives have on average a higher predictive epistemic uncertainty than true positives. This suggests that there is supplementing information in the uncertainty measure not captured by the classification alone.

1. Introduction

The predictive probabilities at the output layer of neural network classifiers are often misinterpreted as model (epistemic) uncertainty (Gal & Ghahramani, 2016). Bayesian statistics provides a coherent framework for representing uncertainty in neural networks (Goodfellow, Bengio, & Courville, 2016; MacKay, 1992), but has not so far gained widespread use in deep learning — presumably due to the high computational cost that traditionally comes with second-order methods. Recently, Gal and Ghahramani (2016) developed a theoretical framework which casts dropout at test time in deep neural networks as approximate Bayesian inference. Due to its mathematical elegance and negligible computational cost, this work has caught great interest in a variety of different fields (Litjens et al., 2017; Loquercio, Segu, & Scaramuzza, 2020; Yan, Gong, Wei, & Gao, 2020; Zhu & Laptev, 2017), but has also generated questions as to what types of uncertainty these approximations actually lead (Osband, 2016; Osband, Blundell, Pritzel, & Roy, 2016) and what types are relevant (Kendall & Gal, 2017). For a general treatment of uncertainty in machine learning, we refer to Hüllermeier and Waegeman (2020).

Epistemic uncertainty is commonly understood as the reducible component of uncertainty — the uncertainty of the model itself, or its parameters. In our context this amounts to the uncertainty in the estimated class probabilities due to limited amount of training data. While the epistemic uncertainty can be reduced by increasing the amount of training data, the other component of uncertainty known as aleatoric uncertainty, is irreducible and stems from the uncertainty in the label assignment process (Song, Kim, Park, & Lee, 2020). However, in this paper we only address the epistemic part, and treat the labels as constant when estimating uncertainty.

Our approach goes back to the work of MacKay (1992), and we show that the above reasoning leads to the method known as the Delta method[1] (Hoef, 2012; Khosravi & Creighton, 2011; Newey & McFadden, 1994) in statistics. However, as the Delta method depends on the empirical Fisher information matrix which grows quadratically with the number of neural network parameters P – its direct application in modern deep learning is prohibitively expensive. We therefore propose a low cost variant of the Delta method applicable to L_2-regularized deep neural networks based on the top K eigenpairs of the Fisher information matrix. We

* Corresponding author.
E-mail addresses: geir.kjetil.nilsen@gmail.com, geir.nilsen@uib.no (G.K. Nilsen), antonella.zanna@uib.no (A.Z. Munthe-Kaas), hans.skaug@uib.no (H.J. Skaug), morten.brun@uib.no (M. Brun).

[1] Also known as the Laplace approximation.

https://doi.org/10.1016/j.neunet.2021.10.014

address efficient computation of full-rank approximate eigendecompositions in terms of either the exact inverse Hessian, the inverse outer-products of gradients (OPG) approximation or the so-called Sandwich estimator. Further, we exhibit the fact that deep learning classifiers tend to be heavily over-parameterized. This leads to flat Fisher information eigenvalue spectra which we show can be exploited in terms of a simple linearization.

Another classical epistemic uncertainty quantification procedure is the Bootstrap (Efron, 1979; Khosravi & Creighton, 2011). A comparison of the Delta methodology presented in this paper and the classical Bootstrap procedure applied to deep learning classification can be found in Nilsen, Munthe-Kaas, Skaug, and Brun (2021).

The theoretical Fisher information matrix is always positive (semi)-definite, and we constrain our empirical counterpart to be the same. Recent research (Alain, Roux, & Manzagol, 2019; Ghorbani, Krishnan, & Xiao, 2019; Sagun, Bottou, & LeCun, 2017; Sagun, Evci, Guney, Dauphin, & Bottou, 2018), consistent with our own observations, show that the exact Hessian after training is rarely positive definite in deep learning. To mitigate this, we propose a simple correction of the right tail of the Hessian eigenvalue spectrum to achieve positive definiteness. We corroborate our choice with two observations: a) negative eigenvalues of the Hessian matrix are highly stochastic across different weight initialization values, and b) correcting the eigenvalue spectrum to achieve positive definiteness yields stable predictive epistemic uncertainty estimates which are perfectly correlated with the estimates based on the OPG approximation — which by construction is always positive (semi)-definite (Martens, 2020).

As the computational cost of the exact inverse Hessian matrix or its full eigendecomposition is prohibitively expensive in deep learning, we propose to use the Lanczos iteration (Trefethen & III, 1997) in combination with Pearlmutter's technique (Pearlmutter, 1994) to compute the needed eigenpairs. Consequently, the matrix inversion will be straightforward, and the net computational complexity will be $O(SPN)$ time and $O(KP)$ space, where N is the number of training examples and S is the number of Lanczos–Pearlmutter steps required to compute K eigenpairs.

Also the inverse OPG approximation or its full eigendecomposition is prohibitively costly in deep learning. Even if we disregard the cubic time inversion and the quadratic space complexity, one is first left to compute and store the $N \times P$-dimensional Jacobian matrix. In deep learning software provisions based on backward-mode automatic differentiation, only the sum of mini-batch gradients can be computed efficiently. We therefore propose to compute mini-batches of the Jacobian using efficient per-example gradients (Nilsen, Munthe-Kaas, Skaug, & Brun, 2019) in combination with incremental singular value decompositions (Levy & Lindenbaum, 2000). Since the OPG approximation can be written as a Jacobian matrix product, its eigenvectors will be the right singular vectors of the Jacobian, and its eigenvalues the squared singular values. This leads to a computational complexity of $O(KPN)$ time and $O(KP)$ space, also accounting for the inversion. The Sandwich estimator requires both the inverse Hessian and the OPG approximation, and is thus $O(max\{K, S\}PN)$ time and $O(KP)$ space.

This work is a continuation of Nilsen et al. (2019), and we here introduce the fully deterministic (Nagarajan & Warnell, 2019) open sourced TensorFlow module `pydeepdelta` (pyDeepDelta, 2018-2021), and illustrate the methodology on two LeNet and ResNet-based convolutional neural network classifiers using the MNIST and CIFAR-10 datasets. The main contributions of the paper can be summarized as follows:

- We recognize the Delta method as a measure of epistemic as opposed to aleatoric uncertainty and break it into two components: the eigenvalue spectrum of the Fisher information (i.e. Hessian) of the cost function and the per-example sensitivities (i.e. gradients) of the model function.
- We show how to approximate the naïve Delta method and thereby reducing the computational complexity in P from quadratic in space and cubic in time, to linear in both space and time. Bounds of the approximation error are provided.
- We provide an accompanying TensorFlow implementation, and demonstrate how it can be applied on a few well known architectures using the MNIST and CIFAR-10 datasets.

The paper is organized as follows: In Section 2 we give definitions which will be used throughout the paper. In Section 3 we review the Delta method in a deep learning classification context, and in Section 4 we outline the details of the proposed methodology. In Sections 5 and 6 we demonstrate the method, and finally, in Section 7 we summarize the paper and give some concluding remarks and ideas of future work.

2. Deep neural networks

We use a feed-forward neural network architecture with dense layers to introduce terminology and symbols, but emphasize that the theory presented in the paper is directly applicable to any L_2-regularized architecture.

2.1. Architectural

A feed-forward neural network is shown in Fig. 1. There are L layers $l = 1, 2, \dots, L$ with T_l neurons in each layer. The input layer $l = 1$, is represented by the input vector $x_n = \begin{pmatrix} x_{n,1} & x_{n,2} & \dots & x_{n,T_1} \end{pmatrix}^T$ where $n = 1, 2, \dots, N$ is the input index. Furthermore, there are $L-2$ dense hidden layers, $l = 2, 3, \dots, L-1$, and a dense output layer $l = L$, each represented by weight matrices $W^{(l-1)} \in \mathbb{R}^{T_l \times T_{l-1}}$, bias vectors $b^{(l)} \in \mathbb{R}^{T_l}$ and vectorized activation functions $a^{(l)}$.

2.2. Parameter vectors

The total number of parameters in the model shown in Fig. 1 can be written,

$$P = \sum_{l=2}^{L} P^{(l)} = \sum_{l=2}^{L} T_{l-1}T_l + T_l, \tag{1}$$

where $P^{(l)}$ denotes the number of parameters in layer l. By definition, $P^{(1)} = 0$ since the input layer contains no weights or biases. Furthermore, we define parameter vectors representing the layer-wise weights and biases as follows,

$$\omega^{(l)} = \begin{bmatrix} \mathrm{vec}(W^{(l)}) \\ b^{(l)} \end{bmatrix} \in \mathbb{R}^{P^{(l)}}, \tag{2}$$

for $l = 2, 3, \dots, L$, with components $\omega_i^{(l)}$, $i = P^{(l-1)} + 1, P^{(l-1)} + 2, \dots, P^{(l)}$. The notation vec$(W)$ denotes a row-wise vectorization[2] of the matrix $W^{A \times B}$ into a column vector of dimension $\mathbb{R}^{AB}$. In the rest of the paper, we consider the full model and define the parameter vector,

$$\omega = \begin{bmatrix} \omega^{(2)} \\ \omega^{(3)} \\ \vdots \\ \omega^{(L)} \end{bmatrix} \in \mathbb{R}^{P}. \tag{3}$$

[2] Standard method in TensorFlow: tf.reshape(W, [−1]).

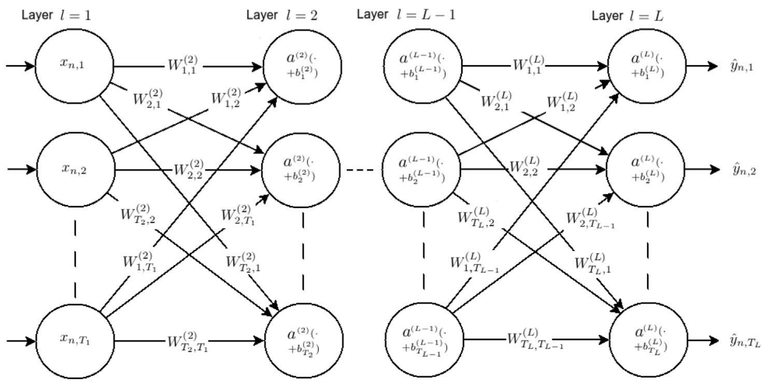


Fig. 1. A feed-forward neural network with dense layers.

2.3. Training, model and cost function

The *model function* $f : \mathbb{R}^{T_1 \times P} \to \mathbb{R}^{T_L}$ associated to the architecture shown in Fig. 1 is defined as

$$f(x_n, \omega) = a^{(L)}[W^{(L)}a^{(L-1)}(\cdots a^{(2)}\{W^{(2)}x_n + b^{(2)}\} + \cdots) + b^{(L)}]. \quad (4)$$

We use a softmax cross-entropy *cost function* $C : \mathbb{R}^P \to \mathbb{R}$ and require L_2-regularization with a rate factor $\lambda > 0$,

$$\begin{aligned} C(\omega) &= \frac{1}{N}\sum_{n=1}^{N} C_n(y_n, \hat{y}_n) + \frac{\lambda}{2}\sum_{p=1}^{P}\omega_p^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}\left(-\sum_{m=1}^{T_L} y_{n,m}\log \hat{y}_{n,m}\right) + \frac{\lambda}{2}\sum_{p=1}^{P}\omega_p^2, \end{aligned} \quad (5)$$

where y_n represents the target vector for the nth example (N examples), and where $\hat{y}_n = f(x_n, \omega)$ represents the corresponding prediction vector obtained by evaluating the *model function* (4) using the input vector x_n and the parameter vector (3). The activation function $a^{(L)} : \mathbb{R}^{T_L} \to \mathbb{R}^{T_L}$ in the output layer is the vectorized softmax function defined as

$$\begin{aligned} a^{(L)}(z) &= \text{softmax}(z) \\ &= \frac{\exp(z)}{\sum_{m=1}^{T_L} \exp(z_m)}, \end{aligned} \quad (6)$$

where $\exp(\cdot)$ denotes the vectorized exponential function. Training of the neural network can be defined as finding an 'optimal' parameter vector $\hat{\omega}$ by minimizing the cost function (5),

$$\hat{\omega} = \arg\min_{\omega \in \mathbb{R}^P} C(\omega). \quad (7)$$

3. The delta method

The Delta method (Hoef, 2012) views a modern deep neural network as a (huge) non-linear regression. In our classification setting, we regard the labels as constant, and thus the epistemic component of the uncertainty associated with predictions of an arbitrary input example x_0 reduces to the evaluation of the covariance matrix of the network outputs (Khosravi & Creighton, 2011). By a first-order Taylor expansion (Grosse, 2020), it can be shown that the covariance matrix of the network outputs $\hat{y}_0$, i.e. the model function (4), can be approximated by

$$\text{Cov}(\hat{y}_0) \approx F\Sigma F^T \in \mathbb{R}^{T_L \times T_L}, \quad (8)$$

where

$$F = \left[F_{ij}\right] \in \mathbb{R}^{T_L \times P}, \; F_{ij} = \frac{\partial}{\partial \omega_j} f_i(x_0, \omega)\bigg|_{\omega = \hat{\omega}} \quad (9)$$

is the Jacobian matrix of the model function, and where Σ is the covariance matrix of the model parameter estimate $\hat{\omega}$. For a given x_0, an approximate standard deviation of $\hat{y}_0$ is provided by the formula

$$\sigma(x_0) \approx \sqrt{\text{diag}\big(F\Sigma F^T\big)} \in \mathbb{R}^{T_L}. \quad (10)$$

Eq. (10) means that when the neural network predicts for an input x_0, the associated epistemic uncertainty per class output is determined by a linear combination of parameter sensitivity (i.e. F) and parameter uncertainty (i.e. Σ). Parameter sensitivity (F) prescribes the amount of change in the neural network output for an infinitesimal change in the parameter estimates, whereas the parameter uncertainty (Σ) prescribes the amount of uncertainty in the parameter estimates themselves.

We apply and compare three different approximations to Σ. The first one is called the **Hessian estimator**, and is defined by

$$\Sigma^{\text{H}} = \frac{1}{N}H^{-1} = \frac{1}{N}\left[\frac{1}{N}\sum_{n=1}^{N}\frac{\partial^2 C_n}{\partial\omega\partial\omega^T}\bigg|_{\omega=\hat{\omega}} + \lambda I\right]^{-1} \in \mathbb{R}^{P\times P}, \quad (11)$$

where H is the empirical Hessian matrix of the cost function evaluated at $\hat{\omega}$.

The second estimator is called the **Outer-Products of Gradients (OPG) estimator** and is defined by

$$\Sigma^{\text{G}} = \frac{1}{N}G^{-1} = \frac{1}{N}\left[\frac{1}{N}\sum_{n=1}^{N}\frac{\partial C_n}{\partial\omega}\frac{\partial C_n}{\partial\omega}^T\bigg|_{\omega=\hat{\omega}} + \lambda I\right]^{-1} \in \mathbb{R}^{P\times P}, \quad (12)$$

where the summation part of G corresponds to the empirical covariance of the gradients of the cost function evaluated at $\hat{\omega}$. Finally, the third estimator is known as the **Sandwich estimator** (Freedman, 2006; Schulam & Saria, 2019) and is defined by

$$\Sigma^{\text{S}} = \frac{1}{N}H^{-1}GH^{-1} \in \mathbb{R}^{P\times P}. \quad (13)$$

Across various fields and contexts, the two famous Eqs. (11) and (12) are often presented and interpreted differently, and the inconsistency in the vast literature is nothing but intriguing. We therefore feel that their appearance in this paper requires some elaboration. Firstly, for the Hessian estimator (11), we note that the differentials act only on the data dependent part of the cost function (5), C_n, so the second term, λI, here comes from the

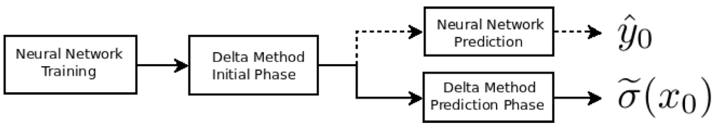

Neural Network Training
Delta Method Initial Phase
Neural Network Prediction
Delta Method Prediction Phase
$\hat{y}_0$
$\widetilde{\sigma}(x_0)$

Fig. 2. The Delta method for quantifying the predictive epistemic uncertainty $\widetilde{\sigma}(x_0)$ of $\hat{y}_0 = f(x_0, \hat{w})$ in deep learning (solid line).

second-order derivatives of the L_2-regularization term. Secondly, for the OPG estimator (12), also here the differentials act on the data dependent part of the cost function, but the crucial detail often confused or let out in the literature comes with the second term, λI: under L_2-regularization it must be added explicitly in order for G to be asymptotically equal to H (See Appendix for a proof) – as is the primary motivation of the OPG estimator as a plug-in replacement of the Hessian estimator in the first place. If let out, G will almost always be singular (Murfet et al., 2020; Watanabe, 2007), and thus cannot be used in (12).

At this point, we can see that two fundamental difficulties arise when applying the Delta method in deep learning: (a) the sheer size of the covariance matrix grows quadratically with P, and (2) the covariance matrix must be positive definite. In other words, we are virtually forced to compute and store the full covariance matrix, and are in terms of the Hessian estimator dependent on that the optimizer can find a true local (or global) minimum of the cost function. Nevertheless, with the OPG and the Sandwich estimators, the second obstacle is virtually inapplicable since they by definition always will be positive definite when $\lambda > 0$.

In the next section we present methodology that addresses both these aspects. We present an indirect correction leaving the Hessian estimator positive definite, and introduce methodology with computational time and space complexity which is linear in P.

4. The delta method in deep learning

We present our approach to the Delta method in deep learning as a procedure carried out in two phases after the neural network has been trained. See Fig. 2.

The first phase – the ‘initial phase’ – is carried out only once, with the scope of indirectly computing full-rank, positive definite approximations of the covariance matrices (11), (12) or (13) based on approximate eigendecompositions of H and G. The second phase – the ‘prediction phase’ – is carried out hand in hand with the regular neural network prediction process (4), and is used to approximate the epistemic component of the predictive uncertainty governed by (10) using the indirect covariance matrix approximation found in the ‘initial phase’.

In the next sections, we address the following aspects of the proposed methods: (a) how to efficiently compute eigenvalues and eigenvectors of the Hessian estimator via the Lanczos iteration and exact Hessian vector products, (b) how to efficiently compute eigenvalues and eigenvectors of the OPG estimator via incremental singular value decompositions, (c) how to combine the former two to obtain an approximation of the Sandwich estimator, and (d) how to apply these estimators to efficiently compute an approximation of (10).

4.1. Computing eigenvalues and eigenvectors of the covariance matrix

The full eigendecomposition of the covariance matrix in (10) is defined by

$$\Sigma = Q\Lambda^{-1}Q^T \in \mathbb{R}^{P\times P}, \qquad (14)$$

Table 1
The computational complexity of the outlined methodology is linear in P across both phases.

	Initial phase		Prediction phase (Per-Example)	
	Time	Space	Time	Space
Hessian	$O(SPN)$	$O(KP)$	$O(T_LPK + T_L^2K + K^2T_L)$	$O(max\{K, T_L\}P)$
OPG	$O(KPN)$			
Sandwich	$O(max\{K, S\}PN)$			

where $Q \in \mathbb{R}^{P\times P}$ is the matrix whose kth column is the eigenvector q_k of Σ, and $\Lambda \in \mathbb{R}^{P\times P}$ is the diagonal matrix whose elements are the corresponding eigenvalues, $\Lambda_{kk} = \lambda_k$. We assume that the eigenvalues are algebraically sorted so that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_P$. Note that in terms of the Hessian estimator, the eigenvalues are precisely the second derivatives of the cost function along the principal axes of the ellipsoids of equal cost, and that Q is a rotation matrix which defines the directions of these principal axes (LeCun, Simard, & Pearlmutter, 1993).

For the Hessian estimator (11), the Lanczos iteration (Trefethen & III, 1997) can be applied to find $K < P$ eigenvalues (and corresponding eigenvectors) in $O(SNP)$ time and $O(KP)$ space when Pearlmutter’s technique (Pearlmutter, 1994) is applied inside the iteration (Nilsen et al., 2019). Pearlmutter’s technique can simply be described as a procedure based on two-pass back-propagations of complexity $O(NP)$ time and $O(P)$ space to obtain exact Hessian vector products without requiring to keep the full Hessian matrix in memory. The number S denotes the number of Lanczos iterations to reach convergence. We observe that the convergence of the Lanczos algorithm is quite fast in our experiments, and we find that S is practically orders of magnitude less than P.

For the OPG estimator (12), a slightly different approach can be applied. Since the OPG estimator can be written as a Jacobian matrix product (Nilsen et al., 2019), we get by the singular value decomposition that its eigenvectors will be the right singular vectors of the Jacobian, and its eigenvalues the squared singular values. Mini-batches of the Jacobian matrix can easily be obtained by standard back-propagation, and so an incremental singular value decomposition (Cardot & Degras, 2015; Levy & Lindenbaum, 2000) can be applied to each mini-batch. The computational cost is thus $O(KNP)$ time and $O(KP)$ space. The Sandwich estimator combines the Hessian and the OPG approximation via the product (13), and thus has a computational complexity of $O(max\{K, S\}NP)$ time and $O(KP)$ space. The computational complexity of the outlined methodology is summarized in Table 1.[3]

Our TensorFlow module `pydeepdelta` (pyDeepDelta, 2018-2021) utilizes the Lanczos implementation available in the SciPy distribution (SciPy), as well as the incremental singular value decomposition available in the scikit-learn distribution (scikit-learn).

[3] Assuming naive matrix multiplication.

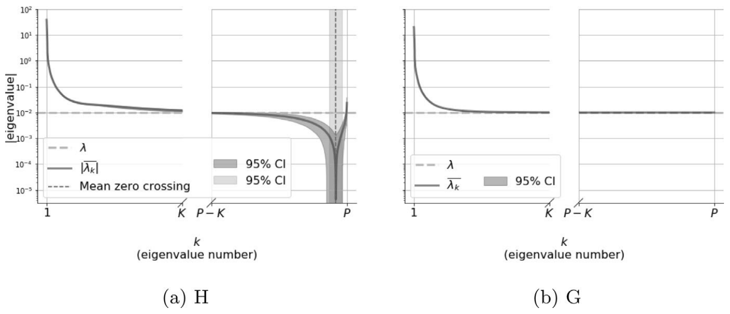


Fig. 3. Log scale eigenvalue magnitude spectra of H and G showing the $K = 1500$ largest (left tail subspace) and the $K = 1500$ smallest (right tail subspace) eigenvalues and their variation across sixteen trained instances of the MNIST network distinguished only by a different random weight initialization. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.2. The eigenvalue spectra of H and G

To better understand the proposed covariance approximations, we first need to explore the prototypical deep learning eigenvalue spectrum of the empirical Hessian matrix H (11) and the empirical covariance of the gradients G (12). To this end, we introduce two LeNet-based convolutional neural network classifiers using the MNIST and CIFAR-10 datasets, and draw parallels to the findings in the literature.

4.2.1. Classifier architectures, parameters and training

The MNIST classifier has $L = 6$ layers, layer $l = 1$ is the input layer represented by the input vector. Layer $l = 2$ is a $3 \times 3 \times 1 \times 32$ convolutional layer followed by max pooling with stride equal to 2 and with a ReLU activation function. Layer $l = 3$ is a $3 \times 3 \times 32 \times 64$ convolutional layer followed by max pooling with a stride equal to 2, and with ReLU activation function. Layer $l = 4$ is a $3 \times 3 \times 64 \times 64$ convolutional layer with ReLU activation function. Layer $l = 5$ is a 576×64 dense layer with ReLU activation function, and the output layer $l = 6$ is a $64 \times T_L$ dense layer with softmax activation function, where the number of classes (outputs) is $T_L = 10$. The total number of parameters is $P = 93{,}322$.

The CIFAR-10 classifier has $L = 6$ layers, layer $l = 1$ is the input layer represented by the input vector. Layer $l = 2$ is a $3 \times 3 \times 3 \times 32$ convolutional layer followed by max pooling with stride equal to 2 and with a ReLU activation function. Layer $l = 3$ is a $3 \times 3 \times 32 \times 64$ convolutional layer followed by max pooling with a stride equal to 2, and with ReLU activation function. Layer $l = 4$ is a $3 \times 3 \times 64 \times 64$ convolutional layer with ReLU activation function. Layer $l = 5$ is a 1024×64 dense layer with ReLU activation function, and the output layer $l = 6$ is a 64×10 dense layer with softmax activation function, where the number of classes (outputs) is $T_L = 10$. The total number of parameters is $P = 122,570$.

We apply random normal weight initialization and zero bias initialization. We use (5) as the cost function with a L_2-regularization rate $\lambda = 0.01$. We utilize the Adam optimizer (Bottou, Curtis, & Nocedal, 2018; Kingma & Ba, 2014) with a batch size of 100, and apply no form of randomized data shuffling. To ensure convergence (i.e. $\|\nabla C(\hat{\omega})\|_2 \approx 0$) we apply the following learning rate schedules given by the following (step, rate) pairs: MNIST = $\{(0, 10^{-3}), (60\text{k}, 10^{-4}), (70\text{k}, 10^{-5}), (80\text{k}, 10^{-6})\}$ and CIFAR-10 = $\{(0, 10^{-3}), (55\text{k}, 10^{-4}), (85\text{k}, 10^{-5}), (95\text{k}, 10^{-6}), (105\text{k}, 10^{-7})\}$. For MNIST, we stop the training after 90,000 steps – corresponding to a training accuracy of 0.979, test accuracy 0.981, training cost $C(\hat{\omega}) = 0.257$ and a gradient norm $\|\nabla C(\hat{\omega})\|_2 = 0.016$. For CIFAR-10, we stop the training after 115,000 steps – corresponding to a training accuracy of 0.701, test accuracy 0.687, training cost $C(\hat{\omega}) = 1.284$ and a gradient norm $\|\nabla C(\hat{\omega})\|_2 = 0.030$.

4.2.2. The eigenvalue spectrum approximation

The general assumption in deep learning is that H after training is not positive definite and mostly contain eigenvalues close to zero (Alain et al., 2019; Ghorbani et al., 2019; Granziol et al., 2019; Sagun et al., 2017, 2018; Watanabe, 2007). The same holds true for G although it by definition must at least be positive semi-definite (Martens, 2020). However, given the discussion in Section 3, we know that L_2-regularization with rate $\lambda/2$ has the effect of shifting the eigenvalues of H and G upwards by λ.

To test this hypothesis, we study the $K = 1500$ algebraically largest and the $K = 1500$ algebraically smallest eigenvalues of H and G for 16 trained instances of the MNIST network defined in Section 4.2.1. These sixteen networks are thus only distinguished from each other by a different random weight initialization prior to training. The two corresponding log-scale eigenvalue magnitude spectra are shown in Fig. 3.

Firstly, we note that in the midpoint gaps of the spectra, there are $P - 2K = 90,195$ 'missing' central eigenvalues which we have not computed. Since the eigenvalues are sorted in decreasing order, all the central eigenvalues must be close to the L_2-regularization rate λ. We refer to this part of the eigenvalue spectrum as the gap. Secondly, we note that the confidence intervals in the plots are taken across instance space, thus telling how the eigenvalue spectrum change based on the 16 random weight initializations. In both plots, the blue confidence interval tells that the largest eigenvalues of H and G (called left tail) are stable across the 16 trained networks, but the smallest eigenvalues of H are changing dramatically (called right tail, left plot). On the contrary, all the eigenvalues of G are stable. Thirdly, as shown by the green vertical dotted line in the upper plot representing the mean zero-crossing, H is clearly not positive definite — even with L_2-regularization. The green confidence interval around the zero-crossing shows that the number of negative eigenvalues also change across the networks.

In Granziol et al. (2019) it was hypothesized that negative Hessian eigenvalues are caused by a discrepancy between the empirical Hessian (i.e. H) and its theoretical counterpart (expected Hessian) in which the summation of (11) is replaced with an expectation so that effectively $N \to \infty$. They showed that as N grows (holding $\hat{\omega}$ fixed), the empirical right tail grows toward λ whereas the rest of the spectrum is stable. Supported by the fact that H and G will be equal in expectation (Appendix), the expected Hessian eigenvalue spectrum might be more similar to that of G where all the eigenvalues are greater than equal to λ.

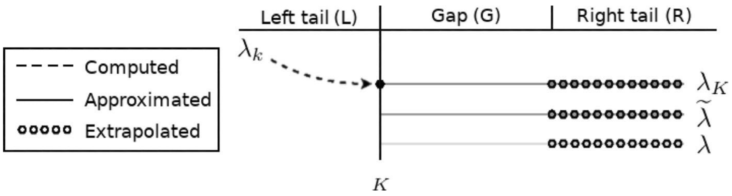


Fig. 4. In terms of its eigenvalue spectrum, the covariance matrix can be partitioned as given by Eq. (15): the left tail subspace (eigenpairs computed), the gap subspace (eigenvalues approximated, eigenvectors implicitly found by orthonormality) and the right tail subspace (eigenvalues extrapolated, eigenvectors implicitly found using orthonormality). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

In line with these ideas and the empirical evidence presented in Fig. 3, we assume that all the smallest eigenvalues of H in the right tail are inherently noisy, and should not be used by the Hessian estimator. Therefore, with reference to Fig. 4, for the Hessian estimator, we (a) calculate all the eigenpairs in the left tail, (b) approximate all the eigenvalues in the gap and (c) extrapolate the eigenvalues from the gap into the right tail. The eigenvectors corresponding to the gap and right tail can implicitly be accounted for by orthonormality as discussed in the next section.

For the OPG estimator, the same principle applies apart from that the extrapolation inherently becomes a part of the gap subspace approximation because we know that G always is positive definite when $\lambda > 0$. Finally, for the Sandwich estimator, we simply apply the aforementioned procedures and estimate the product (13).

4.3. Closing the gap

Based on the observations in the previous section, we now propose a partitioning of the eigendecomposition which reveals that full-rank, positive definite approximations of the Hessian and OPG estimators can be obtained by computing only the eigenpairs corresponding to the K algebraically largest eigenvalues of H and G respectively. Finally, we show how to use these approximations to construct an approximation of the Sandwich estimator.

4.3.1. The Hessian and OPG estimators

In terms of the Hessian and OPG estimators, the full eigendecomposition of the covariance matrix can be partitioned into three subspaces as shown in Fig. 4

$$\Sigma = \Sigma_{\mathrm{L}} + \Sigma_{\mathrm{G}} + \Sigma_{\mathrm{R}} = Q_{\mathrm{L}}\Lambda_{\mathrm{L}}^{-1}Q_{\mathrm{L}}^T + Q_{\mathrm{G}}\Lambda_{\mathrm{G}}^{-1}Q_{\mathrm{G}}^T + Q_{\mathrm{R}}\Lambda_{\mathrm{R}}^{-1}Q_{\mathrm{R}}^T. \quad (15)$$

This decomposition applies to both Σ^{H} (11) and Σ^{G} (12), and thus we have omitted the superscripts in our notation. In practice, the two merely differs by which of the two matrices H and G the calculated eigenpairs come from. The subscript 'G' denotes the gap subspace which is based on eigenvectors with eigenvalues λ_{K+1} to λ_{P-K-1}. Subscript 'L' denotes the left tail subspace and is based on eigenvectors with eigenvalues λ_1 to λ_K. Finally, the subscript 'R' denotes the right tail subspace which is based on eigenvectors with eigenvalues λ_{P-K} to λ_P. Accordingly, we have that $Q_{\mathrm{L}} \in \mathbb{R}^{P\times K}$, $\Lambda_{\mathrm{L}} \in \mathbb{R}^{K\times K}$, $Q_{\mathrm{G}} \in \mathbb{R}^{P\times(P-2K)}$, $\Lambda_{\mathrm{G}} \in \mathbb{R}^{(P-2K)\times(P-2K)}$, $Q_{\mathrm{R}} \in \mathbb{R}^{P\times K}$ and $\Lambda_{\mathrm{R}} \in \mathbb{R}^{P\times K}$.

If $\lambda_K \approx \lambda$ we can safely assume that all the eigenvalues in the gap subspace must be close to λ. In line with (Granziol et al., 2019) and the empirical evidence presented in Fig. 3, we assume that all the eigenvalues in the right subspace are inherently noisy, and should not be used by the Hessian estimator. Consequently, we assume that also the eigenvalues in the right subspace are approximately equal to λ. Since the OPG estimator is always positive definite when $\lambda > 0$, the same assumption also holds true.

With reference to Fig. 4, there are now two possible extreme conditions: (a) when all the eigenvalues in the gap and right subspaces are set to λ_K (blue), or (b) when all the eigenvalues in the gap and right subspaces are set to λ (green). By defining $\widetilde{\lambda}$ (purple) as the harmonic mean of λ and λ_K, and ϵ_λ as the midpoint of their reciprocals,

$$\widetilde{\lambda} = \left(\frac{\lambda^{-1} + \lambda_K^{-1}}{2}\right)^{-1} \quad \text{and} \quad \epsilon_\lambda = \frac{\lambda^{-1} - \lambda_K^{-1}}{2}, \quad (16)$$

it follows that $\widetilde{\lambda}^{-1} \pm \epsilon_\lambda$ will enclose the interval $[\lambda_K^{-1}, \lambda^{-1}]$. The covariance matrix can now be approximated by

$$\widetilde{\Sigma} = \frac{1}{N}\left[Q_{\mathrm{L}}\Lambda_{\mathrm{L}}^{-1}Q_{\mathrm{L}}^T + \widetilde{\lambda}^{-1}(Q_{\mathrm{G}}Q_{\mathrm{G}}^T + Q_{\mathrm{R}}Q_{\mathrm{R}}^T)\right], \quad (17)$$

with a worst-case approximation error Δ given by

$$\Delta = \frac{\epsilon_\lambda}{N}\left[Q_{\mathrm{G}}Q_{\mathrm{G}}^T + Q_{\mathrm{R}}Q_{\mathrm{R}}^T\right], \quad (18)$$

such that Σ is bounded by $\widetilde{\Sigma} \pm \Delta$. Since Q is an orthonormal basis, we see that it is possible to express (17) and (18) without an explicit need to compute any of the eigenvectors relative to the gap nor right tail subspaces because

$$Q_{\mathrm{G}}Q_{\mathrm{G}}^T + Q_{\mathrm{R}}Q_{\mathrm{R}}^T = I - Q_{\mathrm{L}}Q_{\mathrm{L}}^T. \quad (19)$$

Inserting (17) into (10) with use of (19), yields the final form of the approximation to the uncertainty associated with prediction of x_0

$$\widetilde{\sigma}^2(x_0) = \frac{1}{N}\mathrm{diag}\left\{F\left[Q_{\mathrm{L}}\Lambda_{\mathrm{L}}^{-1}Q_{\mathrm{L}}^T + \widetilde{\lambda}^{-1}(I - Q_{\mathrm{L}}Q_{\mathrm{L}}^T)\right]F^T\right\} \in \mathbb{R}^{T_L}, \quad (20)$$

with a worst-case approximation error δ given by

$$\delta = \frac{\epsilon_\lambda}{N}\mathrm{diag}\left\{F\left(I - Q_{\mathrm{L}}Q_{\mathrm{L}}^T\right)F^T\right\} \in \mathbb{R}^{T_L}, \quad (21)$$

such that $\sigma^2(x_0)$ is bounded by $\widetilde{\sigma}^2(x_0) \pm \delta$.

In terms of standard deviations, the worst-case approximation error ϵ of $\widetilde{\sigma}(x_0)$ is given by

$$\epsilon = \frac{1}{2}\left(\sqrt{\widetilde{\sigma}^2(x_0) + \delta} - \sqrt{\widetilde{\sigma}^2(x_0) - \delta}\right) \in \mathbb{R}^{T_L}, \quad (22)$$

such that $\sigma(x_0)$ is bounded by $\widetilde{\sigma}(x_0) \pm \epsilon$. Lastly, we define an 'uncertainty score' (which we will use later to rank images) by summing the variances per class output (class variance), and then take the square root to get the total uncertainty in standard deviations

$$\widetilde{\sigma}_{\mathrm{score}}(x_0) = \sqrt{\sum_{m=1}^{T_L}\widetilde{\sigma}_m^2(x_0)} \in \mathbb{R}, \quad (23)$$

with the corresponding worst-case approximation error $\epsilon_{\mathrm{score}}$ given by,

$$\epsilon_{\mathrm{score}} = \frac{1}{2}\left(\sqrt{\sum_{m=1}^{T_L}\widetilde{\sigma}_m^2(x_0) + \delta_m} - \sqrt{\sum_{m=1}^{T_L}\widetilde{\sigma}_m^2(x_0) - \delta_m}\right) \in \mathbb{R}, \quad (24)$$

such that the true quantity is bounded by $\widetilde{\sigma}_{\text{score}}(x_0) \pm \epsilon_{\text{score}}$. We note that the worst-case approximation errors (21), (22) and (24) are functions of x_0 but we have notationally dropped this from the equations to avoid cluttering. The approximation errors should be thought of as an uncertainty of the predictive uncertainty which accounts for the worst-case loss of not computing the gap subspace explicitly. Since the right tail subspace can be extrapolated when H is not positive definite, the concept of an approximation error for the Hessian estimator must be used carefully.

At this point we make a few comments regarding Eq. (20). The first term on the right hand side, $Q_L \Lambda_L^{-1} Q_L^T$, corresponds to a low-rank approximation of the covariance matrix based on K explicitly computed principal eigenpairs. However, when the second term, $\widetilde{\lambda}^{-1}(I - Q_L Q_L^T)$, is added — the approximation becomes full-rank. When accounting for the left and right multiplication of the sensitivity matrix F, the per-class predictive uncertainties of x_0 can be interpreted as weighted sums of the squared sensitivities in the directions expressed by the eigenbasis Q using the inverse eigenvalues as weights. Hence, for the low-rank approximation – regardless of the sensitivity – the contribution to the predictive uncertainty will be zero in directions $k > K$, whereas for the full-rank approximation — the contribution can still be high. We will come back to this when we discuss out-of-distribution examples in Section 5.

4.3.2. The sandwich estimator

The approximation of the Sandwich estimator is defined by

$$\widetilde{\Sigma} = \frac{1}{N} \widetilde{H}^{-1} \widetilde{G} \widetilde{H}^{-1}. \tag{25}$$

We introduce two separate linearization constants for the approximation of the gap (and right tail) subspace of G and H^{-1} using the harmonic means

$$\widetilde{\lambda}^{\text{H}} = \left(\frac{\lambda^{-1} + \lambda_K^{\text{H}^{-1}}}{2} \right)^{-1}, \tag{26}$$

$$\widetilde{\lambda}^{\text{G}} = \left(\frac{\lambda^{-1} + \lambda_K^{\text{G}^{-1}}}{2} \right)^{-1}. \tag{27}$$

The approximation of H^{-1} is thus given by

$$\widetilde{H}^{-1} = Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} + \widetilde{\lambda}^{\text{H}^{-1}} (I - Q_L^{\text{H}} Q_L^{\text{H}^T}), \tag{28}$$

and the approximation of G given by

$$\widetilde{G} = Q_L^{\text{G}} \Lambda_L^{\text{G}} Q_L^{\text{G}^T} + \widetilde{\lambda}^{\text{G}} (I - Q_L^{\text{G}} Q_L^{\text{G}^T}). \tag{29}$$

The superscripts 'H' and 'G' are used to distinguish the eigenvectors and eigenvalues of H and G respectively. By inserting (28) and (29) into (25) and working out the product, we define the following eight matrices

$$\mathcal{S} = Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} Q_L^{\text{G}} \Lambda_L^{\text{G}} Q_L^{\text{G}^T} Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} \tag{30}$$

$$\mathcal{A} = Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} (I - Q_L^{\text{G}} Q_L^{\text{G}^T}) Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} \tag{31}$$

$$\mathcal{N} = (I - Q_L^{\text{H}} Q_L^{\text{H}^T}) Q_L^{\text{G}} \Lambda_L^{\text{G}} Q_L^{\text{G}^T} Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} \tag{32}$$

$$\mathcal{D} = (I - Q_L^{\text{H}} Q_L^{\text{H}^T})(I - Q_L^{\text{G}} Q_L^{\text{G}^T}) Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} \tag{33}$$

$$\mathcal{W} = Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} Q_L^{\text{G}} \Lambda_L^{\text{G}} Q_L^{\text{G}^T} (I - Q_L^{\text{H}} Q_L^{\text{H}^T}) = \mathcal{N}^T \tag{34}$$

$$\mathcal{I} = Q_L^{\text{H}} \Lambda_L^{\text{H}^{-1}} Q_L^{\text{H}^T} (I - Q_L^{\text{H}} Q_L^{\text{H}^T})(I - Q_L^{\text{G}} Q_L^{\text{G}^T}) = \mathcal{D}^T \tag{35}$$

$$\mathcal{C} = (I - Q_L^{\text{H}} Q_L^{\text{H}^T}) Q_L^{\text{G}} \Lambda_L^{\text{G}} Q_L^{\text{G}^T} (I - Q_L^{\text{H}} Q_L^{\text{H}^T}) \tag{36}$$

$$\mathcal{H} = (I - Q_L^{\text{H}} Q_L^{\text{H}^T})(I - Q_L^{\text{G}} Q_L^{\text{G}^T})(I - Q_L^{\text{H}} Q_L^{\text{H}^T}). \tag{37}$$

The uncertainty associated with prediction of x_0 can now be written

$$\begin{aligned} \widetilde{\sigma}^2(x_0) = \frac{1}{N} \text{diag}\big\{ F\big[& \mathcal{S} + \widetilde{\lambda}^{\text{G}} \mathcal{A} \\ & + \widetilde{\lambda}^{\text{H}^{-1}} (\mathcal{N} + \mathcal{N}^T) \\ & + \widetilde{\lambda}^{\text{G}} \widetilde{\lambda}^{\text{H}^{-1}} (\mathcal{D} + \mathcal{D}^T) \\ & + \widetilde{\lambda}^{\text{H}^{-2}} \mathcal{C} \\ & + \widetilde{\lambda}^{\text{G}} \widetilde{\lambda}^{\text{H}^{-2}} \mathcal{H} \big] F^T \big\} \in \mathbb{R}^{T_L}, \end{aligned} \tag{38}$$

with the worst-case approximation error given by

$$\begin{aligned} \delta = \frac{1}{2N} \text{diag}\big\{ F\big[& (\lambda_K^{\text{G}} - \lambda) \mathcal{A} \\ & + (\lambda^{-1} - \lambda_K^{\text{H}^{-1}})(\mathcal{N} + \mathcal{N}^T) \\ & + (\lambda_K^{\text{G}} \lambda^{-1} - \lambda_K^{\text{H}^{-1}} \lambda)(\mathcal{D} + \mathcal{D}^T) \\ & + (\lambda^{-2} - \lambda_K^{\text{H}^{-2}}) \mathcal{C} \\ & + (\lambda^{-2} \lambda_K^{\text{G}} - \lambda_K^{\text{H}^{-2}} \lambda) \mathcal{H} \big] F^T \big\} \in \mathbb{R}^{T_L}, \end{aligned} \tag{39}$$

such that $\sigma^2(x_0)$ is bounded by $\widetilde{\sigma}^2(x_0) \pm \delta$. In terms of standard deviations, the approximation error is readily found by inserting (38) and (39) into (22).

4.4. On the relation between the effective number of parameters and K

In MacKay (1992), the so-called effective number of parameters is defined in terms of the eigenvalues of the Hessian matrix. It is noted that directions in parameter space for which the eigenvalues are close to λ do not contribute to the number of good parameter measurements. Therefore, the effective number of parameters is a measure of the number of parameters which are well determined by the training data. In other words, when we select K so that $\lambda_K \approx \lambda$, we loosely cover the data dependent part of the Hessian matrix (first term of right hand side of (5)) and can therefore expect that K will be a crude estimate of the number of effective parameters.

As seen by Eqs. (21) and (39), the approximation error will be zero when the smallest eigenvalue λ_K in the left tail subspace (of H and G) is exactly equal to the L_2-regularization rate λ.

5. Demonstration and proof of concept

In the following Section we explore and demonstrate the approximate predictive epistemic uncertainty estimate governed by (10) for the two LeNet-based neural network classifiers that were introduced in Section 4.2.1. We establish by the use of regressions that the three estimators (11)–(13) yield close to perfectly correlated predictive epistemic uncertainty estimates for both of the classifiers.

5.1. The distribution of approximate predictive epistemic uncertainty

Fig. 5 shows nonparametrically smoothed versions of the predictive epistemic uncertainty for the three proposed estimators against class probability for all the images in the MNIST and CIFAR-10 test sets. Clearly, the three estimators yield close to identical results. Further, we observe that the average predictive epistemic uncertainty associated with false positives (yellow line) is higher than for true positives (blue line). The banana-shaped appearance of these plots suggests that there is a negative quadratic relationship between probability and uncertainty. The explanation for this is attributed to the softmax activation function whose gradient (i.e. sensitivity F) will always be weighted by a quantity which is negative quadratic in probability (i.e. $\hat{y}(1-\hat{y})$).

The evolution of the nonparametrically smoothed uncertainty levels and approximation errors for the OPG estimator as functions of the number of computed eigenpairs K and class probability is displayed in Fig. 6. As expected, for a growing K, the

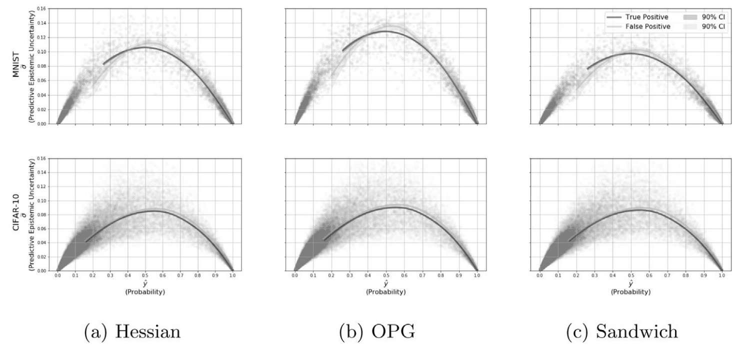


Fig. 5. Nonparametrically smoothed versions of the predictive epistemic uncertainty (10) for the true positives (blue) and false positives (orange) in the MNIST (upper row, $K = 1500$) and CIFAR-10 (lower row, $K = 2500$) test sets as functions of class probability for each of the three estimators. The shaded gray bullets ($N \times T_L$ such bullets) represent the raw predictive uncertainty for all T_L classes against probability. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

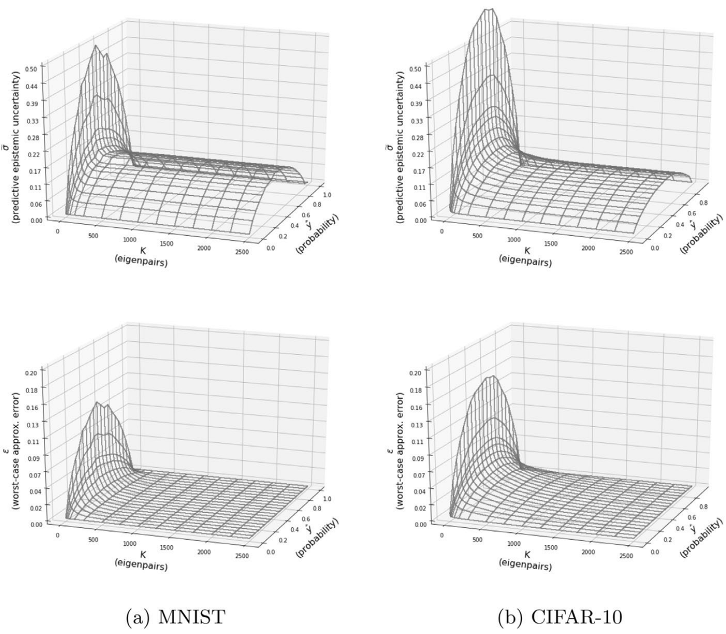


Fig. 6. Nonparametrically smoothed versions of the predictive epistemic uncertainty (upper row) and the approximation error (lower row) in the MNIST and CIFAR-10 test sets as functions of the number of computed eigenpairs K and class probability using the OPG estimator.

approximation errors diminish and the uncertainty stabilizes. Although we do not display similar plots for the other two estimators, we note that for MNIST, the approximation errors are smallest for the OPG estimator, followed by the Hessian estimator and the Sandwich estimator. The larger the difference between λ and the smallest eigenvalue λ_K, the higher the average approximation error. As seen by the eigenvalue spectra in Fig. 3, the drop-off rate toward λ is faster for G, thus explaining why the OPG estimator leads to the lowest approximation errors on MNIST. We note that since the Sandwich estimator is dependent on both the approximation of H and G, its approximation errors are not unexpectedly the highest. Furthermore, the fall-off rate toward λ in the eigenvalue spectrum for CIFAR-10 is slightly lower than for MNIST. This means that the CIFAR-10 classifier has a greater number of effective parameters — and thus requires a higher K to achieve acceptable approximation error levels. This fact is evident by Fig. 6, where we see that the OPG approximation errors for CIFAR-10 are dropping off to zero slower than for MNIST.

For all three estimators, it is evident by Fig. 6 that most of the contribution to the predictive epistemic uncertainty comes from the left subspace corresponding to the largest eigenvalues of H and G. This observation can be counter-intuitive since it is the directions with the smallest eigenvalues that will be the largest contributors to the variance when accounting for the inversions in (11), (12) or (13).

The explanation for this phenomenon is attributed to the sensitivity F (9). We observe that the training and test set sensitivity

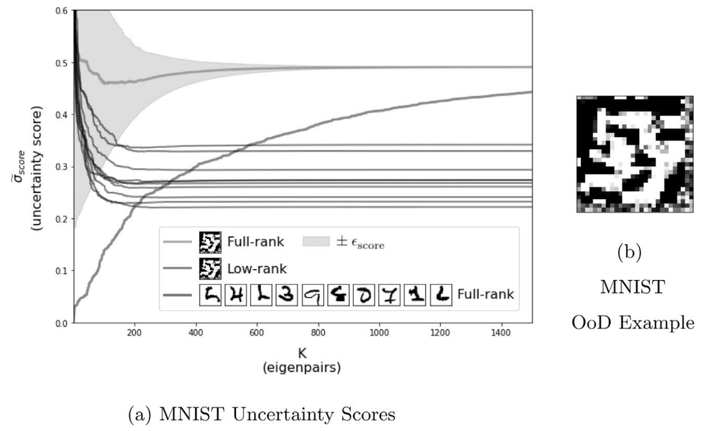


Fig. 7. The uncertainty score (a) as a function of K for the MNIST OoD example in (b) using the full-rank OPG approximation (green curve) vs. its low-rank counterpart (blue curve) from Eqs. (20) and (23). The green interval corresponds to the approximation error. The reference images (black curves) are computing using the full-rank approximation, and corresponds to the ten images in the training set with the highest uncertainty scores sorted in descending order. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
Regression comparison of $\widetilde{\sigma}^{\mathrm{H}}$, $\widetilde{\sigma}^{\mathrm{G}}$ and $\widetilde{\sigma}^{\mathrm{S}}$ across all the images in the MNIST and CIFAR-10 training and test sets. The respective superscripts H, G and S denote Hessian, OPG and Sandwich. The regression intercept, slope and squared correlation coefficient are denoted by α, β and R^2, respectively.

		Hessian vs. OPG $\widetilde{\sigma}^{\mathrm{G}}(x_n) = \alpha + \beta\widetilde{\sigma}^{\mathrm{H}}(x_n)$			Hessian vs. Sandwich $\widetilde{\sigma}^{\mathrm{S}}(x_n) = \alpha + \beta\widetilde{\sigma}^{\mathrm{H}}(x_n)$			OPG vs. Sandwich $\widetilde{\sigma}^{\mathrm{S}}(x_n) = \alpha + \beta\widetilde{\sigma}^{\mathrm{G}}(x_n)$		
		R^2	α	β	R^2	α	β	R^2	α	β
MNIST	Training set	0.997	0.000	1.206	0.998	0.000	0.923	0.990	0.000	0.761
	Test set	0.998	0.000	1.219	0.999	0.000	0.915	0.995	0.000	0.748
CIFAR-10	Training set	0.999	0.000	1.062	0.999	0.000	1.017	0.997	0.000	0.956
	Test set	1.000	0.000	1.066	1.000	0.000	1.014	0.998	0.000	0.950

drops to zero in directions k for which $\lambda_k \approx \lambda$ and is thus canceling with the reciprocals of the smallest eigenvalues in the linear combinations formed by (20) or (38). Nevertheless, as the sensitivity for data not belonging to the same distribution as the training can still be high in these directions, the corresponding predictive epistemic uncertainty can still receive significant contributions from directions $k > K$. This emphasizes the importance of making the estimators full-rank using the orthonormal basis technique presented in Section 4.3. We add that due to the full-rank property, the number K should be thought of as the number of explicitly computed eigenpairs rather than the number of utilized eigenpairs — as the latter will effectively be equal to P.

To illustrate the concept of a low vs. full-rank approximation, Fig. 7a displays the uncertainty scores as functions of K for the low and full-rank version of the OPG estimator applied to the out-of-distribution (OoD) example shown in Fig. 7b. For reference, we also plot the uncertainty scores for the ten images in the training set with the highest uncertainty scores sorted in descending order. Comparing the green curve with the blue curve shows that the OoD example has a sensitivity spectrum stretching out far beyond $K = 1500$ because the low-rank version (blue) has not yet reached the stable level achieved by the full-rank approximation (green) at this K. That the full-rank approximation quickly stabilizes already at around $K = 600$, can be explained by that it receives contribution from the full spectrum even though only K principal eigenpairs are computed explicitly at each stage. The reference images (black curves) are computing using the full-rank approximation, and are all lower ranked than the OoD example.

A detailed comparison of the three estimators is shown in Table 2. By regressing their outcomes against each other, we clearly see that the relative estimated uncertainty levels are near to perfectly correlated since the squared correlations coefficients are close to 1. As seen by the slopes β, only the absolute levels of the estimated uncertainty differ, and since the intercepts α are zero, there are no offsets.

5.2. Ranking images based on the ‘uncertainty score’

We propose to validate our results by studying the MNIST and CIFAR-10 images associated with the maximum and minimum amount of total predictive epistemic uncertainty as defined in (23) using the Hessian estimator. Unsurprisingly, since the squared correlation coefficients in Table 2 are close to 1, the OPG and Sandwich estimators yield almost identical results and are not shown.

The idea is based on the following reasoning: if a neural network classifies an image with low predictive epistemic ‘uncertainty score’, the image should be easy to classify also for a human. Conversely, if the neural network classifies an image with a high predictive epistemic ‘uncertainty score’, the image should be hard to classify for a human. Effectively, the predictive epistemic ‘uncertainty score’ ranks images according to the degree of ‘doubt’ expressed by the neural network — and by the figures we find striking evidence that this corresponds well with human judgment.

6. Computational considerations and larger architectures

Despite the fact that we have reduced the naïve Delta method’s computational complexity to be linear in P, the presented methodology still requires considerable amount of computing power when P grows very large. For reference, the Hessian estimator’s initial phase on the MNIST LeNet with P in the order of

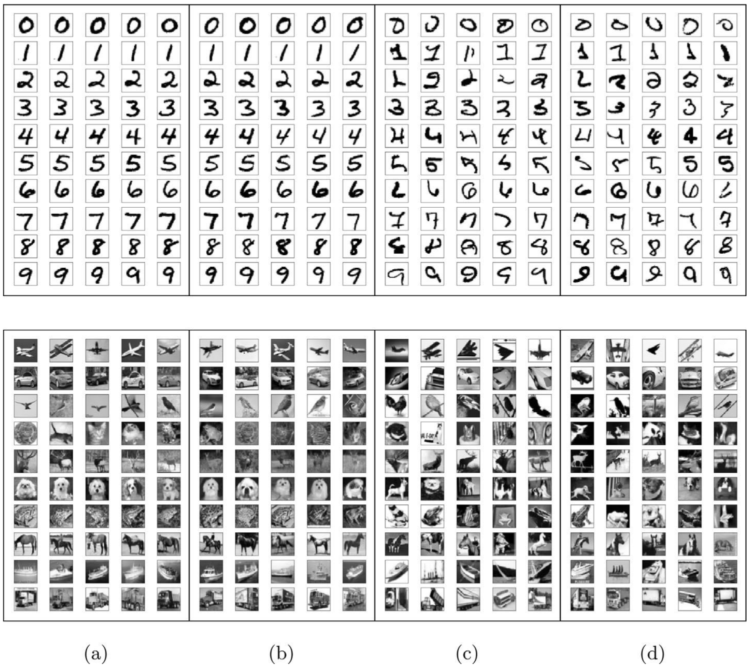


Fig. 8. The MNIST and CIFAR-10 images ranked by the predictive epistemic 'uncertainty score' per class: (a) lowest 5 in the training set, (b) lowest 5 in the test set, (c) highest 5 in the training set and (d) highest 5 in the test set.

10^5 using $K = 600$ (as seen by Fig. 6, $K = 600$ leads to acceptable approximations errors), amounts to an order of Tflops as $N = 60,000$ and as the final number of Lanczos steps turned out to be $S = 2330$ in this case. This corresponds to a computational time of about one hour using an AMD Ryzen 5 2600 CPU @ 3.4 GHz with eight cores and 32 GB memory along with an NVIDIA RTX 2080 Ti based GPU with 11 GB memory. The Hessian estimator's memory requirement amounts to about 500 MBs assuming single precision. For comparison, the naïve Delta method would clock in at the order of Pflops with a theoretical memory requirement of 35 GBs. Since in practice one would need to store both H and its inverse, as well as temporary variables depending on the type of inversion algorithm, the effective memory consumption can be as much as 320 GBs.[4] On top of this, handling the possibility of an indefinite H would require an additional explicit eigenvalue decomposition and several large matrix multiplications. In this regime, the use of direct linear algebra methods is infeasible.

With larger architectures such as ResNets (He, Zhang, Ren, & Sun, 2016), P is several orders of magnitude larger than for the LeNets discussed so far. In particular, ResNet-18 has a P in the order of 10^7. To further investigate the practicality of our methodology in this context, we present supplementing experiments for the Hessian estimator with MNIST and CIFAR-10 using the pre-activation ResNet-18 architecture.

6.1. ResNet-18

Adapting pre-activation ResNet-18 (He et al., 2016) to MNIST and CIFAR-10 leads to a total number of parameters $P = 11,175,818$ and $P = 11,176,970$, respectively. A vital building block of the ResNet architecture family is the batch-normalization (BN) layer (Ioffe & Szegedy, 2015). The β and γ parameters of the BN layers are in the following experiments treated as trainable parameters, and are thus included in both P and all relevant computations (e.g. Hessians). Furthermore, since the operation of BN layers depends on the mode to which they are configured (i.e. training mode or inference mode), we use the following rule: all quantities involving the training set as input data are computed in training mode (e.g. training cost, training accuracy, gradients, Hessians), while quantities involving the test set as input data are computed in inference mode (e.g. test predictions/test accuracy and sensitivity matrices (9)).

The training details are as follows: we apply uniform He (He, Zhang, Ren, & Sun, 2015) weight initialization and zero bias initialization. We use (5) as the cost function with a L_2-regularization rate $\lambda = 0.01$. We utilize the Adam optimizer (Bottou et al., 2018; Kingma & Ba, 2014) with a batch size of 100, and apply no form of randomized data shuffling. To ensure convergence (i.e. $\|\nabla C(\hat{\omega})\|_2 \approx 0$) we apply the following learning rate schedules given by the following (step, rate) pairs: MNIST = $\{(0, 10^{-3}), (60\text{k}, 10^{-4}), (70\text{k}, 10^{-5}), (80\text{k}, 10^{-6})\}$ and CIFAR-10 = $\{(0, 10^{-3}), (55\text{k}, 10^{-4}), (85\text{k}, 10^{-5}), (125\text{k}, 10^{-6}), (155\text{k}, 10^{-7}), (205\text{k}, 10^{-8}), (255\text{k}, 10^{-9})\}$. For MNIST, we stop the training after 90,000 steps — corresponding to a training accuracy of 0.999, test accuracy 0.995, training cost $C(\hat{\omega}) = 0.281$ and a gradient norm $\|\nabla C(\hat{\omega})\|_2 = 0.018$. For CIFAR-10, we stop the training after 285,000 steps – corresponding to a training accuracy of 0.994, test accuracy 0.773, training cost $C(\hat{\omega}) = 0.520$ and a gradient norm $\|\nabla C(\hat{\omega})\|_2 = 0.076$.

To stay within the 32 GB memory bound of the aforementioned computer specification, we managed to compute up to $K = 200$ eigenpairs for the Hessian estimator. For both MNIST and CIFAR-10, the Lanczos algorithm converged at exactly

[4] Try running numpy.linalg.inv(np.diag(10**5).astype('float32')) and watch the memory consumption throughout the whole process.

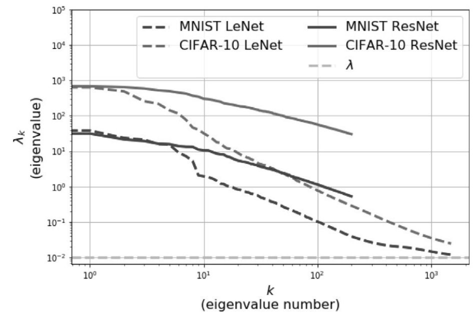


Fig. 9. Comparison of the Hessian eigenvalue spectra of LeNet and ResNet-18 for MNIST and CIFAR-10 data.

$S = 502$ iterations with total computation times of about 14 and 15 h, respectively.

In agreement with the findings in Yao, Gholami, Keutzer, and Mahoney (2020), Fig. 9 shows that the curvature (i.e. Hessian eigenvalue spectrum) of the ResNets has a slower decay to λ. In other words, for a given K, we can expect that a (large) ResNet will yield larger approximation errors (18) than for a (small) LeNet.

Fig. 10 shows the nonparametrically smoothed versions of the predictive epistemic uncertainty for the ResNets against the class probability for all the images in the MNIST and CIFAR-10 test sets. For reference, the corresponding plots for the LeNets were shown in Fig. 5. The absolute level of the predictive uncertainty for the ResNets is larger than for the LeNets, and exceeds the theoretical maximum standard deviation of 0.5 for softmax-based neural networks. A simple inspection of the computed approximation errors (22) for the ResNets rules out a too low $K = 200$ as the only culprit, because the MNIST test set image with the largest approximation error corresponds to $\widetilde{\sigma} \pm \epsilon \approx 19 \pm 4$ and the CIFAR-10 equivalent to $\widetilde{\sigma} \pm \epsilon \approx 50 \pm 13$ (i.e. lower bounds still greater than the theoretical bound 0.5). We leave to investigate the root cause of this anomaly, but speculate that the number of training examples N might simply be too small now that $N \ll P$. Nevertheless, the relative uncertainty levels are still reasonable in terms of the raised level for false positives (as seen by Fig. 10), and in terms of that meaningful rankings similar to those shown in Fig. 8 for the LeNets still can be obtained for the ResNets.

7. Summary, concluding remarks and further work

We have presented a computationally tractable framework for traditional Fisher information based (epistemic) uncertainty quantification in deep learning classification. To this end, we have introduced full-rank, positive definite covariance estimators using approximate eigendecompositions in terms of the Hessian, the OPG approximation and the so-called Sandwich estimator. We have recognized the Delta method as a measure of epistemic as opposed to aleatoric uncertainty and break it into two components: the eigenvalue spectrum of the Fisher information (i.e. Hessian) of the cost function and the per-example sensitivities (i.e. gradients) of the model function. Further, we have proposed to utilize the Lanczos algorithm in combination with Pearlmutter's technique to compute the needed eigenpairs of the Hessian, and to compute mini-batches of the Jacobian matrix using efficient per-example gradients in combination with incremental singular value decompositions for the OPG approximation. As the computational complexity of these methods scale linearly with the number of model parameters, they are therefore suited for deep learning. However, since the computational complexity also scales (linearly) with the number of eigenpairs K, it seems that with today's computing power, the bottleneck of our methodology is reached when the number of parameters is in the order of 10^7.

We have shown that the three estimators yield almost identical prediction uncertainty estimates when applied on two different LeNet-based neural network classifiers. We have seen that only the top $K \ll P$ Fisher information matrix eigenpairs contribute significantly to the predictive uncertainty for data in the same distribution as the training set. As this does not necessarily hold true for OoD examples, we have shown that thanks to the full-rank property of the proposed estimators, these too will converge quickly under the same framework.

We have also seen that when images are ranked according to their relative level of predictive epistemic uncertainty, the ordering corresponds well with human judgment: ambiguous images tend to be highly ranked, and we clearly see why data augmentation is beneficial — since the top ranked images often are prone to unusual perspectives and/or rare colors. Generally, we conjecture that deep learning classifiers can benefit from incorporating also the uncertainty measure in the classification rule. As a corroborative example we have empirically shown that false positives appears to have an average higher prediction uncertainty than true positives.

Looking forward, we point at several specific areas of research which could be investigated. The first candidate is to establish

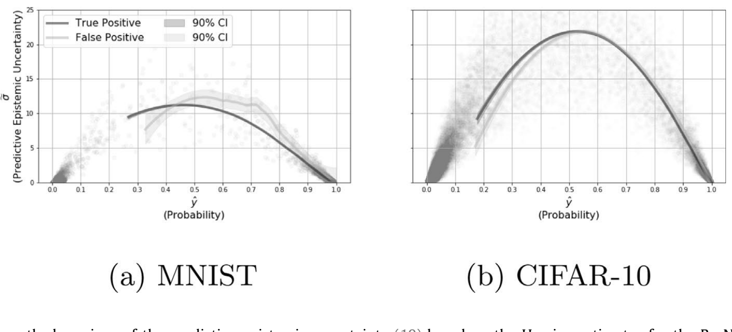


Fig. 10. Nonparametrically smoothed versions of the predictive epistemic uncertainty (10) based on the Hessian estimator for the ResNets using $K = 200$. The true positives (blue) and false positives (orange) are shown for MNIST in (a) and for CIFAR-10 in (b) using the test sets as input data and are plotted as functions of the class probability. The shaded gray bullets ($N \times T_L$ such bullets) represent the raw predictive uncertainty for all T_L classes against probability. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

how the Fisher information eigenspectrum of very large networks and datasets behave. If the contraction of the eigenspectrum toward λ continues to be fast with growing network and dataset sizes, the methodology presented can be tractable even for the most complex models. However, if the largest affordable K yields a λ_K far from λ, it can render the methodology intractable as the approximation errors can be too large. This points to understanding what causes the contraction phase in the first place, and hence uncovering the factors that drive it. Promising research in this direction is the Stochastic Lanczos algorithm (Lin, Saad, & Yang, 2016; Yao et al., 2020) as well as the investigation of pathological spectra of the Fisher information (Karakida, Akaho, & Amari, 2019). However, inconveniently for our methodology, we find evidence in agreement with recent literature (Yao et al., 2020) showing that ResNets have a higher number of effective parameters than LeNets. In other words, for a given K, we can expect that a (large) ResNet will yield larger approximation errors than for a (small) LeNet. Secondly, we leave the discussion regarding which of the three estimators (or other combinations) one should use – and when – as an opportunity for future research. Thirdly, as this work has been focused on the classification task, a natural extension is to see how the framework behaves under deep learning regression (Khosravi & Creighton, 2011). Fourthly, we point at a fundamental issue with the Delta method itself. The Delta method is inevitably based on the local curvature around the parameter estimate $\hat{\omega}$, hence incorporating no means about the uncertainty of the parameter estimate outside this local region. What is lost, and how much, by disregarding the broader perspective of the solution space – a space potentially within reach for sampling methods.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Parts of this work have been done in the context of CEDAS (Center for Data Science, University of Bergen, Norway). The lead author would also like to thank Dr. Berent Å. S. Lunde, Dr. Kristian Gundersen, Øyvind Lunde Rørtveit and Egil Martinsen Aspevik for their helpful advice on various technical issues examined in this paper.

Appendix

The cost function $C(\omega)$ can be interpreted as the negative log posterior,

$$C(\omega) = -\log p(\mathcal{D}|\omega)p(\omega), \tag{40}$$

for the parameter ω and some training data $\mathcal{D}$, where $p(\mathcal{D}|\omega)$ is the likelihood and $p(\omega)$ the prior. Under L_2-regularization with rate $\lambda/2$, the prior takes the form of a multivariate normal distribution with zero mean and covariance $(\lambda/2)^{-1}I$

$$\omega \sim \mathcal{N}\big(0, (\lambda/2)^{-1}I\big). \tag{41}$$

It follows that

$$H_{C(\omega)} = -H_{\log p(\mathcal{D}|\omega)p(\omega)} = -H_{\log p(\mathcal{D}|\omega)} + \lambda I, \tag{42}$$

where we have used that $H_{\log p(\omega)} = -\lambda I$. Taking expectation with respect to $p(\mathcal{D}|\omega)$, and drawing on the well known result for the expected Fisher information matrix (Lehmann & Casella, 1998):

$$\mathop{\mathbb{E}}_{p(\mathcal{D}|\omega)}\left[H_{\log p(\mathcal{D}|\omega)}\right] = -\mathop{\mathbb{E}}_{p(\mathcal{D}|\omega)}\left[\nabla \log p(\mathcal{D}|\omega)\nabla \log p(\mathcal{D}|\omega)^T\right], \tag{43}$$

it follows that

$$\mathop{\mathbb{E}}_{p(\mathcal{D}|\omega)}\left[H_{C(\omega)}\right] = \mathop{\mathbb{E}}_{p(\mathcal{D}|\omega)}[G] + \lambda I \quad \square \tag{44}$$

References

Alain, G., Roux, N. L., & Manzagol, P.-A. (2019). Negative eigenvalues of the hessian in deep neural networks. https://arxiv.org/abs/1902.02366, arXiv:1902.02366v1 [cs.LG].

Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Review*, *60*(2), 223–311.

Cardot, H., & Degras, D. (2015). Online principal component analysis in high dimension: Which algorithm to choose?. https://arxiv.org/abs/1511.03688 arXiv:1511.03688 [stat.ML].

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *Annals of Statistics*, *7*(1), 1–26.

Freedman, D. A. (2006). On the so-called "Huber Sandwich Estimator" and 'Robust Standard Errors". *The American Statistician*, *60*(4), 299–302, https://www.jstor.org/stable/27643806.

Gal, Y., & Ghahramani, Z. (2016). Dropout as a approximation: Representing model uncertainty in deep learning. https://arxiv.org/pdf/1506.02142, arXiv:1506.02142v6 [stat.ML].

Ghorbani, B., Krishnan, S., & Xiao, Y. (2019). An investigation into neural net optimization via hessian eigenvalue density. https://arxiv.org/abs/1901.10159, arXiv:1901.10159v1 [cs.LG].

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press, http://www.deeplearningbook.org.

Granziol, D., Garipov, T., Zohren, S., Vetrov, D., Roberts, S., & Wilson, A. G. (2019). The Deep Learning Limit: are negative neural network eigenvalues just noise? In *Presented at the ICML 2019 workshop on theoretical physics for deep learning*.

Grosse, R. (2020). Lecture 2: Taylor approximations. https://www.cs.toronto.edu/~rgrosse/courses/csc2541_2021/readings/L02_Taylor_approximations.pdf.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026–1034).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

Hoef, J. M. V. (2012). Who invented the delta method? *The American Statistician*, *66*(2), 124–127, https://www.researchgate.net/publication/254329376_Who_Invented_the_Delta_Method.

Hüllermeier, E., & Waegeman, W. (2020). Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. https://arxiv.org/abs/1910.09457, arXiv:1910.09457v2 [cs.LG].

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448–456). PMLR.

Karakida, R., Akaho, S., & Amari, S.-i. (2019). Pathological spectra of the Fisher information metric and its variants in deep neural networks. arXiv preprint arXiv:1910.05992.

Kendall, A., & Gal, Y. (2017). What uncertainties do we need in Bayesian deep learning for computer vision? https://arxiv.org/pdf/1703.04977, arXiv:1703.04977v2 [cs.CV].

Khosravi, A., & Creighton, D. (2011). A comprehensive review of neural network-based prediction intervals and new advances. *IEEE Transactions on Neural Networks*, *22*(9), https://www.researchgate.net/publication/51534965_Comprehensive_Review_of_Neural_Network-Based_Prediction_Intervals_and_New_Advances.

Kingma, D. P., & Ba, J. L. (2014). Adam: A method for stochastic optimization. In *Proc. 3rd int. conf. learn. representations*.

LeCun, Y., Simard, P. Y., & Pearlmutter, B. (1993). Automatic learning rate maximization by on-line estimation of the Hessian's eigenvectors. In *Advances in neural information processing systems (NIPS 1992)*. http://yann.lecun.com/exdb/publis/pdf/lecun-simard-pearlmutter-93.pdf.

Lehmann, E. L., & Casella, G. (1998). *Theory of point estimation* (2nd ed.). (p. 125). Springer Science & Business Media.

Levy, A., & Lindenbaum, M. (2000). Sequential karhunen–loeve basis extraction and its application to images. *IEEE Transactions on Image Processing*, *9*(8), http://www.cs.technion.ac.il/~mic/doc/skl-ip.pdf.

Lin, L., Saad, Y., & Yang, C. (2016). Approximating spectral densities of large matrices. *SIAM Review*, *58*(1), 34–65.

Litjens, G., Kooi, T., Bejnordi, B. E., Setio, A. A. A., Ciompi, F., Ghafoorian, M., et al. (2017). A survey on deep learning in medical image analysis. *Medical Image Analysis*, *42*, 60–88, http://www.sciencedirect.com/science/article/pii/S1361841517301135.

Loquercio, A., Segu, M., & Scaramuzza, D. (2020). A general framework for uncertainty estimation in deep learning. https://arxiv.org/pdf/1907.06890, arXiv:1907.06890v4 [cs.CV].

MacKay, D. (1992). A practical Bayesian framework for backpropagation networks. *Neural Computation*, *4*(3), 448–472, http://www.inference.org.uk/mackay/PhD.html#PhD.

Martens, J. (2020). New insights and perspectives on the natural gradient method. https://arxiv.org/abs/1412.1193, arXiv:1412.1193 [cs.LG].

Murfet, D., Wei, S., Gong, M., Li, H., Gell-Redman, J., & Quella, T. (2020). Deep learning is singular, and that's good. https://arxiv.org/abs/2010.11560, arXiv:2010.11560 [cs.LG].

Nagarajan, P., & Warnell, G. (2019). Deterministic implementations for reproducibility in deep reinforcement learning. https://arxiv.org/abs/1809.05676, arXiv:1809.05676 [cs.AI].

Newey, W. K., & McFadden, D. (1994). *Handbook of econometrics*. https://www.sciencedirect.com/science/article/pii/S1573441205800054.

Nilsen, G. K., Munthe-Kaas, A. Z., Skaug, H. J., & Brun, M. (2019). Efficient computation of Hessian matrices in TensorFlow. https://arxiv.org/abs/1905.05559, arXiv:1905.05559v1 [cs.LG].

Nilsen, G. K., Munthe-Kaas, A. Z., Skaug, H. J., & Brun, M. (2021). A comparison of the delta method and the bootstrap in deep learning classification. http://arxiv.org/abs/2107.01606, arXiv:2107.01606 [cs.LG].

Osband, I. (2016). Risk versus uncertainty in deep learning: Bayes, bootstrap and the dangers of dropout. In *NIPS workshop on Bayesian deep learning*. http://bayesiandeeplearning.org/2016/papers/BDL_4.pdf.

Osband, I., Blundell, C., Pritzel, A., & Roy, B. V. (2016). Deep exploration via bootstrapped DQN. In *Conference on neural information processing systems (NIPS)*. https://papers.nips.cc/paper/6501-deep-exploration-via-bootstrapped-dqn.pdf.

Pearlmutter, B. A. (1994). Fast exact multiplication by the hessian. *Neural Computation*, *6*(1), 147–160, http://www.bcl.hamilton.ie/~barak/papers/nc-hessian.pdf.

pyDeepDelta: A TensorFlow Module Implementing the Delta Method in Deep Learning Classification, https://github.com/gknilsen/pydeepdelta.git.

Sagun, L., Bottou, L., & LeCun, Y. (2017). Eigenvalues of the hessian in deep learning: Singularity and beyond. https://arxiv.org/abs/1611.07476, arXiv:1611.07476v2 [cs.LG].

Sagun, L., Evci, U., Guney, V. U., Dauphin, Y., & Bottou, L. (2018). Empirical analysis of the hessian of over-parametrized neural networks. https://arxiv.org/abs/1706.04454, arXiv:1706.04454v3 [cs.LG].

Schulam, P., & Saria, S. (2019). Can you trust this prediction? Auditing pointwise reliability after learning. https://arxiv.org/abs/1901.00403, arXiv:1901.00403 [stat.ML].

scikit-learn, https://scikit-learn.org/.

SciPy, http://www.scipy.org.

Song, H., Kim, M., Park, D., & Lee, J.-G. (2020). Learning from noisy labels with deep neural networks: A survey. https://arxiv.org/pdf/2007.08199, arXiv:2007.08199v2 [cs.LG].

Trefethen, L. N., & III, D. B. (1997). *Numerical linear algebra* (pp. 243–284). Siam.

Watanabe, S. (2007). Almost all learning machines are singular. In *Proceedings of the 2007 IEEE symposium on foundations of computational intelligence*. http://watanabe-www.math.dis.titech.ac.jp/users/swatanab/foci2007.pdf.

Yan, C., Gong, B., Wei, Y., & Gao, Y. (2020). Deep multi-view enhancement hashing for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *43*(4), 1445–1451.

Yao, Z., Gholami, A., Keutzer, K., & Mahoney, M. W. (2020). Pyhessian: Neural networks through the lens of the hessian. In *2020 IEEE international conference on big data (Big data)* (pp. 581–590). IEEE.

Zhu, L., & Laptev, N. (2017). Deep and confident prediction for time series at uber. In *2017 IEEE international conference on data mining workshops (ICDMW)*.

Paper 3 [Nilsen et al., 2021b]

5.3 A Comparison of the Delta Method and the Bootstrap in Deep Learning Classification

Geir K. Nilsen, Antonella Z. Munthe-Kaas, Hans J. Skaug, Morten Brun, arXiv preprint: 1912.00832, http://arxiv.org/abs/2107.01606, 2021.

A Comparison of the Delta Method and the Bootstrap in Deep Learning Classification

Geir K. Nilsen[1,2], Antonella Z. Munthe-Kaas[1], Hans J. Skaug[1], and Morten Brun[1]

[1] *Department of Mathematics, University of Bergen*
[2] *geir.kjetil.nilsen@gmail.com*

Abstract

We validate the deep learning classification adapted Delta method introduced in [11] by a comparison with the classical Bootstrap. We show that there is a strong linear relationship between the quantified predictive epistemic uncertainty levels obtained from the two methods when applied on two LeNet-based neural network classifiers using the MNIST and CIFAR-10 datasets. Furthermore, we demonstrate that the Delta method offers a five times computation time reduction compared to the Bootstrap.

1 Introduction

It can be beneficial to distinguish between epistemic and aleatoric uncertainty in machine learning models [5]. Bayesian statistics provides a coherent framework for representing epistemic uncertainty in neural networks [9], but has not so far gained widespread use in deep learning [3] – presumably due to the high computational cost that traditionally comes with Fisher information based methods. In particular, the Delta method [4, 6] depends on the empirical Fisher information matrix which grows quadratically with the number of neural network parameters P – and its direct application in modern deep learning is therefore prohibitively expensive. To mitigate this, [11] proposed a low cost variant of the Delta method applicable to L_2-regularized deep neural networks based on the top K eigenpairs of the Fisher information matrix.

In this paper, we validate the methodology introduced in [11] by a comparison with the classical Bootstrap [2, 6, 8, 12, 13]. We show that there is a strong linear relationship between the quantified epistemic uncertainty levels obtained from the two methods when applied on two LeNet-based neural network classifiers using the MNIST and CIFAR-10 datasets.

The paper is organized as follows: in Section 2 we review the Bootstrap and the Delta method in a deep learning classification context. In Section 3 we introduce two LeNet-based classifiers which will be used in the comparison in Section 4, and finally, in Section 5 we summarize the paper and give some concluding remarks.

2 Introduction to the Methodologies

In the following, we denote the training set by $\{x_n \in \mathbb{R}^{T_1}, y_n \in \mathbb{R}^{T_L}\}_{n=1}^{N}$, the test set by $\{x_n \in \mathbb{R}^{T_1}, y_n \in \mathbb{R}^{T_L}\}_{n=1}^{N_{\text{test}}}$ and an arbitrary input example by x_0. The parameter space is denoted by the vector $\omega \in \mathbb{R}^P$, where P is the number of parameters (weights and biases) in the model. The parameter values after training is denoted by the vector $\hat{\omega} \in \mathbb{R}^P$. Furthermore, a prediction for x_0 is denoted by $\hat{y}_0 = f(x_0, \hat{\omega}) \in \mathbb{R}^{T_L}$ where $f : \mathbb{R}^{T_1 \times P} \to \mathbb{R}^{T_L}$ is a deep neural network model function [3] and where T_L denotes the number of classes. Furthermore, it is assumed that the cost function denoted by C is L_2-regularized with a regularization-rate factor $\lambda/2$.

2.1 The Bootstrap in Deep Learning Classification

In the context of deep learning classification, the classical Bootstrap method starts by creating B datasets from the original dataset by sampling with replacement. Subsequently, B networks are trained separately on each of the bootstrapped datasets. The epistemic uncertainty for each of the T_L class predictions (in standard deviations) associated with prediction of x_0 is obtained by the sample standard deviation over the ensemble of B predictions,

$$\widetilde{\sigma}_{\text{boot}}(x_0) = \sqrt{\frac{1}{B-1}\sum_{b=1}^{B}(\hat{y}_0^{(b)} - \overline{\hat{y}_0})^2} \in \mathbb{R}^{T_L}, \tag{1}$$

where the vector $\hat{y}_0^{(b)}$ represents the T_L predictions for x_0 (one probability per class) obtained from the bth bootstrapped network, and where $\overline{\hat{y}_0}$ is the sample mean,

$$\overline{\hat{y}_0} = \frac{1}{B}\sum_{b=1}^{B}\hat{y}_0^{(b)} \in \mathbb{R}^{T_L}. \tag{2}$$

The method is easy to implement efficiently in practice. Training B networks is an 'embarrassingly' parallel problem, and the space complexity for the bootstrapped datasets is just $O(BN)$ when an indexing scheme is used for the sampling with replacement. The experiments conducted in this paper is based on the example `pydeepboot.py` provided in the `pydeepdelta` provision [14].

2.2 The Delta Method in Deep Learning Classification

The Delta method was adapted to the deep learning classification context by [11]. The adaption addresses several fundamental difficulties that arise when the method is applied in deep learning. In essence, it is shown that an approximation of the eigendecomposition of the Fisher information matrix utilizing only K eigenpairs allows for an efficient implementation with bounded worst-case approximation errors. We briefly review the standard method here for convenience.

An approximation of the epistemic component of the uncertainty associated with the prediction of x_0 can be found by the formula

$$\widetilde{\sigma}_{\text{delta}}(x_0) = \sqrt{\operatorname{diag}\big(F\Sigma F^T\big)} \in \mathbb{R}^{T_L}, \tag{3}$$

where the sensitivity matrix F in (3) is defined

$$F = \left[F_{ij}\right] \in \mathbb{R}^{T_L \times P}, \ F_{ij} = \frac{\partial}{\partial \omega_j} f_i(x_0, \omega)\bigg|_{\omega=\hat{\omega}}. \tag{4}$$

The covariance matrix Σ in (3) can be estimated by several alternative estimators. In [11] it was demonstrated that the Hessian estimator, the Outer-Products of Gradients (OPG) estimator and the Sandwich estimator lead to nearly perfect correlated results for two different deep learning models. Since the models discussed in this paper are identical to those in [11], we thus focus only on one of the estimators, namely the OPG estimator defined by

$$\Sigma = \frac{1}{N} G^{-1} = \frac{1}{N}\left[\frac{1}{N}\sum_{n=1}^{N} \frac{\partial C_n}{\partial \omega}\frac{\partial C_n}{\partial \omega}^T\bigg|_{\omega=\hat{\omega}} + \lambda I\right]^{-1} \in \mathbb{R}^{P \times P}, \tag{5}$$

where the summation part of G corresponds to the empirical covariance of the gradients of the cost function evaluated at $\hat{\omega}$. As discussed in [11], the term λI is explicitly added in order to make the OPG estimator asymptotically equal to the Hessian estimator, as is the primary motivation for the former as a plug-in replacement of the latter in the first place.

When the Delta method is implemented under the framework of [11], it has several desirable properties: a) requires only $O(PK)$ space and $O(KPN)$ time, b) fits well with deep learning software frameworks based on automatic differentiation, c) works with any L_2-regularized neural network architecture, and d) does not interfere with the training process as long as the norm of the gradient of the cost function is approximately equal to zero after training.

3 The Neural Network Classifiers

We deploy two LeNet-based neural network architectures which differs only by the number of neurons in two of the layers in order to individually match the formats of the MNIST and CIFAR-10 datasets. Our TensorFlow code for the Delta method is based on the `pydeepdelta` Python module [14], and is fully deterministic [10]. The corresponding Bootstrap implementation can be found in the same repository.

3.1 MNIST

There are $L = 6$ layers, layer $l = 1$ is the input layer represented by the input vector. Layer $l = 2$ is a $3 \times 3 \times 1 \times 32$ convolutional layer followed by max pooling with stride equal to 2 and with a ReLU activation function. Layer $l = 3$ is a $3 \times 3 \times 32 \times 64$ convolutional layer followed by max pooling with a stride equal to 2, and with ReLU activation function. Layer $l = 4$ is a $3 \times 3 \times 64 \times 64$ convolutional layer with ReLU activation function. Layer $l = 5$ is a 576×64 dense layer with ReLU activation function, and the output layer $l = 6$ is a $64 \times T_L$ dense layer with softmax activation function, where the number of classes (outputs) is $T_L = 10$. The total number of parameters is $P = 93322$.

3.2 CIFAR-10

There are $L = 6$ layers, layer $l = 1$ is the input layer represented by the input vector. Layer $l = 2$ is a $3 \times 3 \times 3 \times 32$ convolutional layer followed by max pooling with stride equal to 2 and with a ReLU activation function. Layer $l = 3$ is a $3 \times 3 \times 32 \times 64$ convolutional layer followed by max pooling with a stride equal to 2, and with ReLU activation function. Layer $l = 4$ is a $3 \times 3 \times 64 \times 64$ convolutional layer with ReLU activation function. Layer $l = 5$ is a 1024×64 dense layer with ReLU activation function, and the output layer $l = 6$ is a 64×10 dense layer with softmax activation function, where the number of classes (outputs) is $T_L = 10$. The total number of parameters is $P = 122570$.

3.3 Training Details

For the Bootstrap networks, we test two different weight initialization variants: dynamic random normal weight initialization (DRWI) and static random normal weight initialization (SRWI). The former uses a different (e.g. dynamic) seed across the replicates, meaning that each network in the DRWI Bootstrap ensemble will start out with different random weight values. The latter case uses the same (e.g. static) seed across the replicates, and hence all the networks in the SRWI Bootstrap ensemble receives the same random initial weight values. For all networks, we use zero bias initialization. Futhermore, to investigate the impact of random weight initialization on the Delta method, we apply the Delta method 16 times on a set of 16 networks distinguished only by DRWI.

We use the cross-entropy cost function with a L_2-regularization rate $\lambda = 0.01$, and utilize the Adam [7, 1] optimizer with a batch size of 100, and no form of randomized data shuffling. To ensure convergence (e.g. $||\nabla C(\hat{\omega})||_2 \approx 0$), we apply two slightly different learning rate schedules given by the following (step, rate) pairs: MNIST $= \{(0, 10^{-3}), (60\text{k}, 10^{-4}), (70\text{k}, 10^{-5}), (80\text{k}, 10^{-6})\}$ and CIFAR-10 $= \{(0, 10^{-3}), (55\text{k}, 10^{-4}), (85\text{k}, 10^{-5}), (95\text{k}, 10^{-6}, (105\text{k}, 10^{-7})\}$. For MNIST, we stop the trainings after $90,000$ steps, while for CIFAR-10, after $115,000$ steps – corresponding to the overall training statistics shown in Table 1.

Networks	**Dataset**	**Training Set Accuracy**	**Test Set Accuracy**	$C(\hat{\omega})$	$\vert\vert\nabla C(\hat{\omega})\vert\vert_2$
DRWI Bootstrap B=100	MNIST	0.979 ± 0.000	0.981 ± 0.001	0.253 ± 0.006	0.016 ± 0.013
	CIFAR-10	0.705 ± 0.025	0.684 ± 0.020	1.248 ± 0.042	0.035 ± 0.020
SRWI Bootstrap B=100	MNIST	0.979 ± 0.000	0.981 ± 0.001	0.254 ± 0.002	0.017 ± 0.013
	CIFAR-10	0.715 ± 0.010	0.693 ± 0.009	1.235 ± 0.018	0.031 ± 0.014
Delta 16 reps (DRWI)	MNIST	0.979 ± 0.000	0.981 ± 0.001	0.257 ± 0.002	0.016 ± 0.005
	CIFAR-10	0.701 ± 0.032	0.687 ± 0.029	1.284 ± 0.053	0.030 ± 0.012

Table 1: Training statistics for the Delta and Bootstrap networks. The DRWI and SRWI Bootstrap ensembles each consists of $B = 100$ bootstrapped networks, while the Delta method is applied repeatedly on 16 networks distinguished only by DRWI. Averages $\pm$ two standard deviations are calculated across the $B = 100$ networks for the Bootstrap, and across the 16 repetitions for the Delta method.

4 Comparison

The basic comparison design entails a set of 16 linear regressions on the predictive uncertainty estimates obtained from the two methods using test sets as input data

$$\widetilde{\sigma}_{\text{boot}}(x_n)_m = \alpha_d + \beta_d \widetilde{\sigma}_{\text{delta}}(x_n)_{m,d} + e_{n,m,d}, \quad \begin{aligned} n &= 1, 2, \ldots, N_{\text{test}} \\ m &= 1, 2, \ldots, T_L \\ d &= 1, 2, \ldots, 16. \end{aligned} \tag{6}$$

Accounting for the two variants of the Bootstrap (SRWI/DRWI), this leads to two sets of squared correlation coefficients, intercepts, slopes and Delta method approximation errors, respectively denoted by $\{R^2_d, \alpha_d, \beta_d, \epsilon_d\}_{d=1}^{16}$. Furthermore, as we wish to analyze the impact of the number of Bootstrap replicates and the number of Delta method eigenpairs, we generate these sets for various B and K. An outline of the setup is shown in Figure 1.

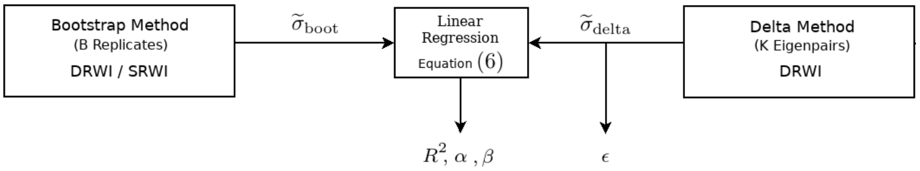
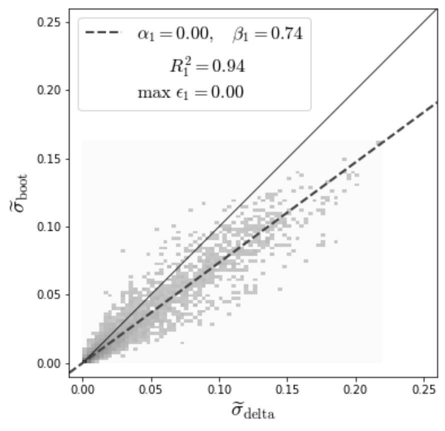


Figure 1: Regression (6) of $\widetilde{\sigma}_{\text{boot}}$ onto $\widetilde{\sigma}_{\text{delta}}$.

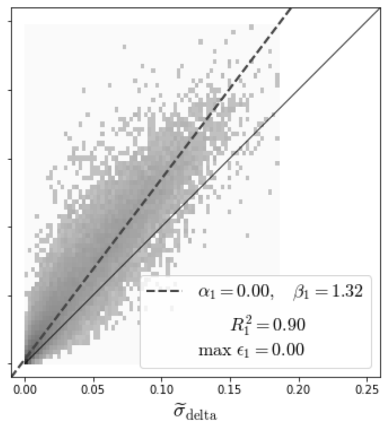
Figure 2 shows scatter plots of the regression results for the first repetition ($d = 1$) of the Delta method against the DRWI Bootstrap ensemble. These plots are based on $B = 100$ bootstrap replicates, and we have selected $K = 1500$ eigenpairs for MNIST and $K = 2500$ eigenpairs for CIFAR-10. Clearly, there is a strong linear relationship between the two methods: the squared correlation coefficients are $R^2_1 = 0.94$ for MNIST and $R^2_1 = 0.90$ for CIFAR-10. On the other hand, the absolute uncertainty level differs between the methods and datasets. This can be seen by the slope coefficients, where the Delta method is overestimating ($\beta_1 < 1$) on MNIST, and underestimating ($\beta_1 > 1$) on CIFAR-10. Further, since the estimated intercepts (α_1) are zero, there are no offsets between the methods. Finally, we see that the maximum across examples and class outputs of the Delta method approximation errors (ϵ_1) are zero, so there is nothing to be achieved by increasing K. As we will see later, K has here been selected unnecessarily high and can be significantly reduced with no loss of accuracy.

4.1 Discussion of the Regression Results as a Function of B and K

The results from the full set of regressions ($d = 1, 2, \ldots, 16$) holding a fixed $B = 100$ are shown in Figure 3. The primary observations are as follows: The mean squared correlation coefficients R^2 are generally high for MNIST and CIFAR-10, meaning that there is a strong linear relationship between the uncertainty levels obtained by the Bootstrap and the Delta method. For the lowest K, the R^2 starts out at 90% for MNIST, and at 81% for CIFAR-10. As K grows, an increase by only 4% is observed for MNIST, while 8% for CIFAR-10. The major difference observed as K increases lies in the absolute uncertainty levels expressed by the slope β: for MNIST, the slope stabilizes at around $K = 600$ while at about $K = 1000$ for CIFAR-10. The same trend is reflected in the maximum approximation errors ϵ, where we respectively see



(a) MNIST



(b) CIFAR-10

Figure 2: Predictive uncertainty estimates obtained from the Delta method (first repetition, $d = 1$) against the DRWI Bootstrap for (a) MNIST using $B = 100$ replicates and $K = 1500$ eigenpairs, and (b) CIFAR-10 using $B = 100$ replicates and $K = 2500$ eigenpairs.

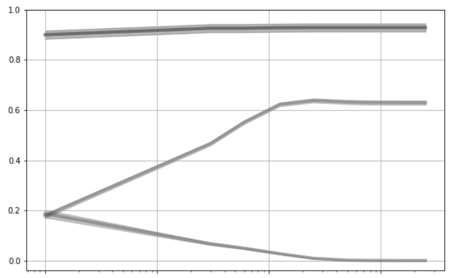
them approach zero at the same values for K. Although not shown in the plots, the regression intercepts α are always zero, meaning that there is no offset in the uncertainty estimates by the two methods.

The main difference found from applying DRWI opposed to SRWI for the Bootstrap ensembles, is that the absolute level of uncertainty increases with DRWI. This is expected, since the DRWI version of the Bootstrap will be more prone to reaching different local minima, and therefore also captures this additional variance. Supporting evidence for this hypothesis is evident by CIFAR-10's wider confidence intervals. A more pronounced geometry difference across various local minima will ultimately lead to higher variability in the R^2 and β. A slightly higher mean R^2 (+1-2%) is also observed for the DRWI version of the Bootstrap. This is reasonable given the fact that also the Delta method networks are more prone to reaching different local minima across the 16 repetitions because of DRWI.

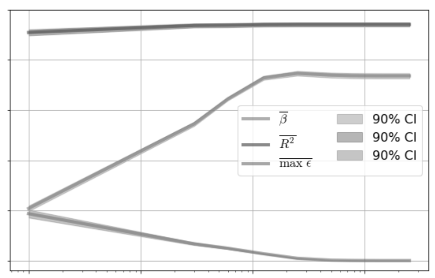
Figure 4 shows the same type of comparison when the number of Bootstrap replicates B varies, and the number of eigenpairs are fixed ($K = 1500$ for MNIST and $K = 2500$ for CIFAR-10). The main observation from this experiment is that there is very little to achieve by selecting a larger ensemble size B than about 50, as this is the point where the mean slope and squared correlation coefficient stabilizes.

4.2 Computation Time

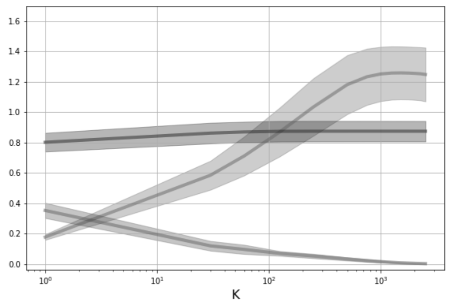
Table 2 shows the computation time for the two methods when executed on a Nvidia RTX 2080 Ti based GPU. For MNIST, the smallest K leading to acceptable approximation errors and stable absolute uncertainty levels for the Delta method is at $K = 600$, while for CIFAR-10 the same applies at $K = 1000$. Furthermore, the smallest acceptable B leading to stable correlation and absolute uncertainty levels for the Bootstrap is at $B = 50$. We conclude that



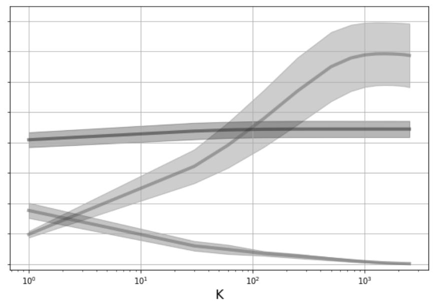
(a) MNIST
Delta vs. SRWI Bootstrap



(b) MNIST
Delta vs. DRWI Bootstrap

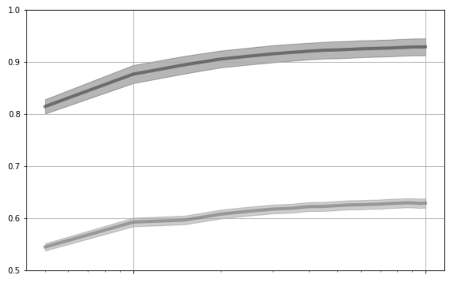


(c) CIFAR-10
Delta vs. SRWI Bootstrap

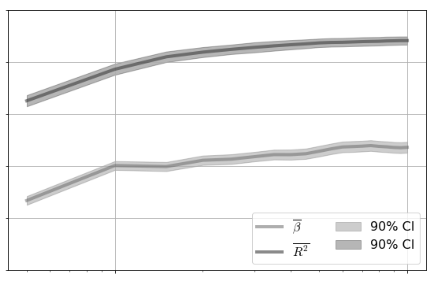


(d) CIFAR-10
Delta vs. DRWI Bootstrap

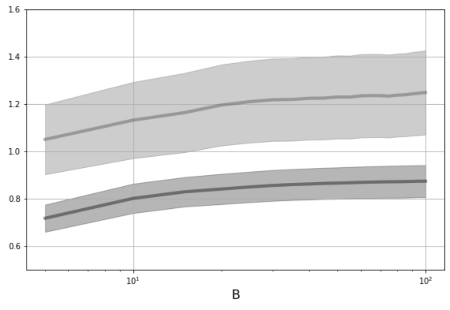
Figure 3: Summaries of the regressions of $\widetilde{\sigma}_{\text{boot}}$ onto $\widetilde{\sigma}_{\text{delta}}$ as given by (6), for different values of K and a fixed $B = 100$. The solid lines and the associated confidence intervals represent the mean and the variation of the regression results across the 16 repetitions of the Delta method.



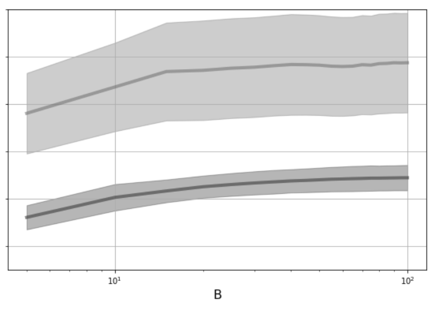
(a) MNIST
Delta vs. SRWI Bootstrap



(b) MNIST
Delta vs. DRWI Bootstrap



(c) CIFAR-10
Delta vs. SRWI Bootstrap



(d) CIFAR-10
Delta vs. DRWI Bootstrap

Figure 4: Summaries of the regressions of $\widetilde{\sigma}_{\text{boot}}$ onto $\widetilde{\sigma}_{\text{delta}}$ as given by (6), for different values of B and a fixed number of eigenpairs K. The solid lines and the belonging confidence intervals represent the mean and the variation of the regression results across the 16 repetitions of the Delta method.

in these experiments the Delta method outperforms the Bootstrap in terms of computation time by a factor 4.6 on MNIST, and a factor 5.9 for CIFAR-10.

Method	Classifier	B	K	Initial Phase [h:mm:ss]	Prediction Phase [mm:ss]		Total [h:mm:ss]
					Training Set	Test Set	
Bootstrap	MNIST	50	N/A	4:08:28	00:19	00:03	4:08:50
	CIFAR-10			7:37:16	00:40	00:07	7:38:04
Delta	MNIST	N/A	600	0:42:33	9:52	1:37	0:54:02
	CIFAR-10		1000	1:00:54	14:44	02:56	1:18:35

Table 2: Computation time for the Bootstrap and Delta method. For the Bootstrap, the 'initial phase' accounts for the parallelized training of B networks, while the 'prediction phase' accounts for the predictive epistemic uncertainty estimation (1), which is further divided into the training and test sets. For the Delta method, the 'initial phase' accounts for the approximate eigendecomposition of the covariance matrix (5), while the 'prediction phase' accounts for the predictive epistemic uncertainty estimation (3), further divided into the training set and test sets.

5 Concluding Remarks

We have shown that there is a strong linear relationship between the predictive epistemic uncertainty estimates obtained by the Bootstrap and the Delta method when applied on two different deep learning classification models. Firstly, we find that the number of eigenpairs K in the Delta method can be selected order of magnitudes lower than P with no loss of correspondence between the methods. This coincides with the fact that when the Delta method approximation errors are sufficiently close to zero, there is no nothing to achieve by a further increase in K, and therefore the correspondence will stabilize at this point.

Secondly, we find that the DRWI version of the Bootstrap yields the best correspondence, and that there is little to achieve by using more than $B = 50$ replicates. Thirdly, we observe that the most complex model (CIFAR-10) yields a high variability in the correspondence across multiple DRWI Delta method runs. We interpret this effect as caused by cost functional multi-modality, and that the Delta method fails to capture the additional variance tied to reaching local minima of different geometric characteristics. Finally, in our experiments we have seen that the Delta method outperforms the Bootstrap in terms of computation time by a factor 4.6 on MNIST and by a factor 5.9 for CIFAR-10.

References

[1] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. SIAM Rev., vol. 60, no. 2, pp. 223-311, 2018.

[2] B. Efron. Bootstrap methods: Another look at the jackknife. Ann. Stat., vol. 7, no. 1, pp. 1–26., 1979.

[3] I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. `http://www.deeplearningbook.org`, MIT Press, 2016.

[4] J. M. V. Hoef. Who Invented the Delta Method? https://www.researchgate.net/publication/254329376_Who_Invented_the_Delta_Method, The American Statistician, 66:2, 124-127, 2012.

[5] E. Hüllermeier and W. Waegeman. Aleatoric and Epistemic Uncertainty in Machine Learning: An Introduction to Concepts and Methods. https://arxiv.org/abs/1910.09457, arXiv:1910.09457v2 [cs.LG], 2020.

[6] A. Khosravi and D. Creighton. A Comprehensive Review of Neural Network-based Prediction Intervals and New Advances. https://www.researchgate.net/publication/51534965_Comprehensive_Review_of_Neural_Network-Based_Prediction_Intervals_and_New_Advances, IEEE Transactions On Neural Networks, Vol. 22, No. 9, 2011.

[7] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In Proc. 3rd Int. Conf. Learn. Representations, 2014.

[8] B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. https://arxiv.org/pdf/1612.01474, arXiv:1612.01474v3 [stat.ML], 2017.

[9] D. MacKay. A practical Bayesian framework for backpropagation networks. http://www.inference.org.uk/mackay/PhD.html#PhD, Neural Computation, 4(3):448–472, 1992., 1992.

[10] P. Nagarajan and G. Warnell. Deterministic Implementations for Reproducibility in Deep Reinforcement Learning. https://arxiv.org/abs/1809.05676, arXiv:1809.05676 [cs.AI], 2019.

[11] G. K. Nilsen, A. Z. Munthe-Kaas, H. J. Skaug, and M. Brun. Epistemic Uncertainty Quantification in Deep Learning Classification by the Delta Method. https://arxiv.org/abs/1912.00832, arXiv:arXiv:1912.00832 [cs.LG], 2021.

[12] I. Osband. Risk versus Uncertainty in Deep Learning: Bayes, Bootstrap and the Dangers of Dropout. http://bayesiandeeplearning.org/2016/papers/BDL_4.pdf, NIPS Workshop on Bayesian Deep Learning, 2016.

[13] I. Osband, C. Blundell, A. Pritzel, and B. V. Roy. Deep Exploration via Bootstrapped DQN. https://papers.nips.cc/paper/6501-deep-exploration-via-bootstrapped-dqn.pdf, Conference on Neural Information Processing Systems (NIPS), 2016.

[14] pyDeepDelta: A TensorFlow Module Implementing the Delta Method in Deep Learning Classification. https://github.com/gknilsen/pydeepdelta.git.

Graphic design: Communication Division, UIB / Print: Skipnes Kommunikasjon AS

uib.no

ISBN: 9788230863992 (print)
9788230861981 (PDF)