On the Delta Method for Uncertainty Approximation in Deep Learning

Geir K. Nilsen∗
geir.kjetil.nilsen@gmail.com

Antonella Z. Munthe-Kaas∗ Hans J. Skaug∗ Morten Brun∗

Abstract

The Delta method is a well known procedure used to quantify uncertainty in statistical models. The method has previously been applied in the context of neural networks, but has not reached much popularity in deep learning because of the sheer size of the Hessian matrix. In this paper, we propose a low cost variant of the method based on an approximate eigendecomposition of the positive curvature subspace of the Hessian matrix. The method has a computational complexity of $O(KPN)$ time and $O(KP)$ space, where $K$ is the number of utilized Hessian eigenpairs, $P$ is the number of model parameters and $N$ is the number of training examples. Given that the model is $L_2$-regularized with rate $\lambda$, we provide a bound on the uncertainty approximation error given $K$. We show that when the smallest Hessian eigenvalue in the ‘positive’ $\frac{K}{2}$-tail of the full spectrum, and the largest Hessian eigenvalue in the ‘negative’ $\frac{K}{2}$-tail of the full spectrum are both approximately equal to $\lambda$, the error will be close to zero even when $K \ll P$. We demonstrate the method by a TensorFlow implementation, and show that meaningful rankings of images based on prediction uncertainty can be obtained for a convolutional neural network based MNIST classifier. We also observe that false positives have higher prediction uncertainty than true positives. This suggests that there is supplementing information in the uncertainty measure not captured by the probability alone.
1 Introduction

The method known as the Delta method is a variance estimation method used to quantify uncertainty in statistical models [1]. The Delta method has previously been applied in the context of neural networks [5], but has not reached much popularity in deep learning because of the sheer size of the Hessian matrix.

In this paper, we propose a low cost approximation to the Delta method based on an approximate eigendecomposition of the positive curvature subspace of the Hessian matrix. By Hessian vector products [9] and the Lanczos iteration [15], we bring down the computational complexity to $O(KP)$ time and $O(KP^2)$ space, where $K$ is the number of utilized Hessian eigenpairs, $P$ is the number of model parameters and $N$ is the number of training examples. Given that the model is $L_2$-regularized with rate $\frac{\lambda}{2}$, we provide a bound on the uncertainty approximation error given $K$. We show that when the smallest Hessian eigenvalue in the ‘positive’ $\frac{K}{2}$-tail of the full spectrum, and the largest Hessian eigenvalue in the ‘negative’ $\frac{K}{2}$-tail of the full spectrum are both approximately equal to $\lambda$, the error will be close to zero even when $K \ll P$.

For the classical Delta method to render absolute levels of uncertainty, the Hessian matrix must be positive definite. However, recent research [10] consistent with our own observations, show that this is very difficult to achieve in deep learning. To cope with this, we propose to ignore the contribution from the negative curvature subspace of the Hessian matrix. And we make two important observations: 1) meaningful rankings of images based on prediction uncertainty can still be obtained, and 2) the prediction uncertainty associated with false positives seem to be higher than for true positives. This suggests that even when the quantified levels of uncertainty cannot be regarded as absolute, there seems to be important supplementing information in the uncertainty measure which is not captured by the probability alone.

This work is a continuation of the work of [8], and we here introduce the open sourced Python module pydeepdelta [13], and demonstrate the method by applying it on a convolutional neural network based MNIST classifier.

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 classical Delta method in a deep learning classification context, and in Section 4 we outline the details of the proposed method. In Section 5 we demonstrate the method, and finally, in Section 6 we summarize the paper and give some concluding remarks.

2 Definitions

Although we here use a feed-forward neural network architecture with dense layers to introduce terminology, the theory and implementation presented is still directly applicable on other neural network architectures using convolutional layers, pooling and other forms of regularization.

2.1 The Architecture

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

2.2 The Parameter Vectors

The total number of parameters in the model can be written,

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

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

$$\omega^{(l)} = \left[\begin{array}{c} \text{flattened}(W^{(l)}) \\ b^{(l)} \end{array}\right] = \left[\begin{array}{c} \omega^{(l)}_1 \end{array}\right] \in \mathbb{R}^{P^{(l)}},$$  \hspace{1cm} (2)

for $l = 2, 3, \ldots, L$ and $i = P^{(l-1)} + 1, P^{(l-1)} + 2, \ldots, P^{(l)}$. The notation flattened($W$) denotes a row-wise vectorization of the matrix $W^{A \times B}$ into a column vector of dimension $\mathbb{R}^{AB}$. The notation used allows for expressing parameter vectors for subsets
of layers. For example, for layers 2, 4 and 6 we can write,

$$
\omega^{(2,4,6)} = \begin{bmatrix}
\omega^{(2)} \\
\omega^{(4)} \\
\omega^{(6)}
\end{bmatrix} \in \mathbb{R}^{P^{(2)} + P^{(4)} + P^{(6)}}.
$$

For the rest of this section, we consider the full model. The corresponding parameter vector is written,

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

2.3 The Model Function

The architecture shown in Figure 1 has the corresponding model function \( f(x_n, \omega) : \mathbb{R}^{T_1} \times P \to \mathbb{R}^{T_L} \) defined by,

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

2.4 The Cost Function

Let the cost function \( C(\omega) : \mathbb{R}^{P} \to \mathbb{R} \) coincide with TensorFlow’s built-in softmax cross-entropy function with \( L_2 \)-regularization,

$$
C(\omega) = \frac{1}{N} \sum_{n=1}^{N} C_n(y_n, \hat{y}_n) + \frac{\lambda}{2} \sum_{p=1}^{P} \omega_p^2
$$

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

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 \( n \)th example, and where \( \hat{y}_n \) represents the corresponding prediction vector obtained by evaluating the model function \( f(x_n, \omega) \) using the input vector \( x_n \) and the parameter vector \( \omega \). As usual, the parameter \( \lambda \) is used to control the impact of the \( L_2 \)-regularization.

Per definition, the activation function in the output layer is now the vectorized softmax function \( \sigma^{(L)}(z) : \mathbb{R}^{T_L} \to \mathbb{R}^{T_L} \) defined by,

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

where \( \exp(\cdot) \) denotes the vectorized exponential function.

---

1. TensorFlow API r1.13: tf.losses.softmax_cross_entropy()
2.5 Training

Training of the neural network can be defined as finding an ‘optimal’ parameter vector $\hat{\omega}$ by minimizing the cost function \(6\),

$$\hat{\omega} = \arg \min_{\omega \in \mathbb{R}^P} C(\omega). \quad (9)$$

3 The Delta Method

The Delta method \[4\] views a neural network as non-linear regression. By a second-order Taylor expansion, it can be shown that the covariance of the model function \[5\] can be approximated by,

$$\text{Cov}(f(x_0, \hat{\omega})) \approx F\Sigma F^T \in \mathbb{R}^{T_x \times T_x}, \quad (10)$$

where

$$F = [F_{ij}] \in \mathbb{R}^{T_x \times P}, \quad F_{ij} = \left. \frac{\partial}{\partial \omega_j} f_i(x_0, \omega) \right|_{\omega = \hat{\omega}} \quad (11)$$

is the Jacobian matrix of the model function, and \(\Sigma\) is the covariance matrix of the model parameters. An approximation of the variance associated with the prediction of \(x_0\) can thus be found by the formula,

$$\sigma^2(x_0, \hat{\omega}) \approx \text{diag}(F\Sigma F^T) \in \mathbb{R}^{T_x}. \quad (12)$$

Equation \(12\) means that when the neural network predicts for an input \(x_0\), the associated prediction variance per class output is determined by summing products of parameter sensitivity (e.g. \(F\)) and parameter variance (e.g. \(\Sigma\)). Parameter sensitivity prescribes the amount of change in the neural network output for an infinitesimal change in the parameter estimates, whereas the parameter variance prescribes the amount of variance associated with the parameter estimates themselves.

Furthermore, we define the approximate prediction uncertainty (standard deviation) associated with \(x_0\) as,

$$\sigma(x_0, \hat{\omega}) = \sqrt{\sigma^2(x_0, \hat{\omega})} \in \mathbb{R}^{T_x}. \quad (13)$$

If (and only if) the parameter vector \(\hat{\omega}\) is a local minimum of the cost function \(6\), the covariance matrix of the model parameters can be estimated by the Hessian estimator \[7\],

$$\Sigma = \frac{1}{N} H(\hat{\omega})^{-1} \in \mathbb{R}^{P \times P}, \quad (14)$$

where \(H(\hat{\omega}) \in \mathbb{R}^{P \times P}\) is the Hessian matrix defined by

$$H(\hat{\omega}) \equiv \left. \frac{\partial^2 C(\omega)}{\partial \omega \partial \omega^T} \right|_{\omega = \hat{\omega}} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial^2 C_n}{\partial \omega \partial \omega^T} \left|_{\omega = \hat{\omega}} \right. + \lambda I. \quad (15)$$

At this point, we can see that two fundamental difficulties arise when applying the Delta method in deep learning: 1) the sheer size of the Hessian matrix grows quadratically with \(P\), and 2) for the inverse of the Hessian matrix to be a valid covariance estimator, the Hessian matrix must be positive definite. In other words, we are dependent on that the optimizer can find a true local minimum of the cost function.

In the next section we present a method that attempts address both these aspects. We present an indirect approximation of the positive curvature subspace of \(14\) that has a 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 separate process carried out in two phases after the neural network has been trained. See Figure \[2\].

The first phase – coined the ‘initial phase’ – is carried out only once, and is done to compute an indirect approximation of the covariance matrix \(14\) based on an approximate eigendecomposition of the positive curvature subspace of the Hessian matrix. The second phase – termed the ‘prediction phase’ – is carried out hand in hand with the regular neural network prediction process \[5\], and is used to approximate the associated prediction uncertainty governed by \[13\]. The ‘prediction phase’ is based on the indirect covariance matrix approximation found in the ‘initial phase’.

In the next sections, we will address the following aspects of the proposed method: 1) how to efficiently compute eigenvalues and eigenvectors of the Hessian matrix via the Lanczos iteration and exact Hessian vector products (Section \[11\], and 2) how to approximate the positive curvature subspace of the covariance matrix \(14\), and apply it to efficiently compute an approximation of \(13\).
Figure 2: The Delta method for quantifying the uncertainty of $\hat{y} = f(x_0, \hat{w})$ in deep learning (solid line).

4.1 Computing Hessian Eigenvalues and Eigenvectors

The full eigendecomposition of the Hessian matrix is defined by

$$H(\hat{\omega}) = QAQ^T \in \mathbb{R}^{P \times P},$$

where $Q \in \mathbb{R}^{P \times P}$ is the matrix whose $k$th column is the eigenvector $q_k$ of $H(\hat{\omega})$, and $\Lambda \in \mathbb{R}^{P \times P}$ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, $\Lambda_{kk} = \lambda_k$. We assume that the eigenvalues are sorted so that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_P$. Note that 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 [6]. See Figure 3.

The Lanczos iteration [15] can be applied to find $K < P$ Hessian eigenvalues (and corresponding eigenvectors) in $O(KNP)$ time and $O(KP)$ space. The algorithm does not require the formation of the full Hessian matrix, but proceeds based solely on Hessian vector products, which in our deep learning context can be computed exactly and efficiently in $O(NP)$ time [9]. Our Python module pydeepdelta [13], achieves this by setting up a Hessian vector product graph in TensorFlow [1] as shown by [8], and evaluates this in a LinearOperator-callback passed on to SciPy’s [14] ARPACK [12] wrapper eigsh(). This seems to be roughly the same approach as taken by [2] although they have not released any code.

For the classifier introduced in Section 4.2.1 we observe that the convergence of the Lanczos algorithm is quite fast. However, as the Hessian vector products in this application must be evaluated over the entire training set per iteration, the overall processing time can still be relatively long. For $K = 5000$, the processing time is about 4 hours when using a Nvidia RTX 2080 Ti based GPU. We emphasize, however, that it is only required to run the Lanczos iteration once time in the ‘initial phase’ of the Delta method as described in the beginning of Section 1.

Figure 3: A saddle (non-convex, hyperbolic paraboloid) with one positive ($\lambda_1$) and one negative ($\lambda_2$) Hessian eigenvalue. The principal direction $q_2$ has negative curvature, therefore $\lambda_2 < 0$. Furthermore, since $|\lambda_1| > |\lambda_2|$, the curvature in direction $q_1$ is seen sharper than in direction $q_2$.

4.2 The Covariance Matrix Approximation

To better understand the proposed covariance approximation, we first need to explore the typical deep learning Hessian eigenvalue spectrum. To this end, we introduce a convolutional neural network based MNIST classifier, and draw parallels to the findings of [10, 11, 2, 3, 16]. For reproducibility, we give explicit references denoted by (#x.y.) to our Jupyter notebook pydeepdelta_demo.ipynb included in the pydeepdelta [13] Github repository.
4.2.1 Classifier Architecture, Parameters and Setup

The neural network architecture (#1.1) can be described as follows: 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 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 \( 576 \times 64 \) dense layer with softplus activation function, and layer \( l = 6 \) is a \( 64 \times 10 \) dense layer with softmax activation function. The total number of parameters is \( P = 93322 \).

The cost function is according to (6) with \( \lambda = 0.01 \). For the training (#1.2), we utilize the Adam optimizer with a batch-size of 100 with no form of data shuffling, an initial learning rate of 0.001, random weight initialization and biases initialized to zero. We stop the training after 55000 steps, when the minimization seems to stall on a training cost of about 0.3 (#1.5), and with training and validation accuracies of about 97% (#1.4). The norm of the cost gradient is strictly non-zero, but has the value of about 1 (#1.5).

4.2.2 The Hessian Eigenvalue Spectrum

Under the assumption that deep learning Hessian matrices after training likely are not positive definite [10, 11, 2, 3, 16], it is to expect that the typical Hessian eigenvalue spectrum consists of a relatively small number of positive eigenvalues, a large number of zero eigenvalues, and a relatively small number of negative eigenvalues.

To test this hypothesis for our classifier, we select \( K = 5000 \) and utilize the Lanczos iteration (Section 4.1) to find \( \frac{K}{2} \) Hessian eigenvalues (and corresponding eigenvectors) at each end of the full spectrum (#2.1, #2.2, #2.3). A log-scale Hessian eigenvalue magnitude spectrum is shown in Figure 4 (#2.4). As indicated by the red vertical dotted line representing the zero crossing, there are both positive and negative eigenvalues, and therefore the Hessian matrix is not positive definite. Furthermore, we let \( P^+ \) denote the number of positive eigenvalues. In this example, \( P^+ = 92376 \), so the fraction of positive eigenvalues is 99%.

The sharp dip in the spectrum located at the zero-crossing is clearly not an emerging singularity, because selecting a larger \( K \) will only populate more eigenvalues within the midpoint gap of the spectrum. The gap in the spectrum is located between \( k = \frac{K}{2} \) and \( k = P - \frac{K}{2} \), and is indicated by a green vertical dotted line. In other words, there are \( P - K = 88322 \) ‘missing’ positive eigenvalues in the gap which we have not computed.

As shown in [15], \( L_2 \)-regularization with rate \( \frac{\lambda}{2} \) has the effect of shifting the diagonal values of the Hessian matrix by \( \lambda \). This means that the eigenvalues can be seen as shifted from \( \lambda_k \) to \( \lambda_k + \lambda \) when compared to a Hessian matrix computed with \( \lambda \) equal to zero. Therefore, as indicated by the blue horizontal dotted line in Figure 4, we confirm the above hypothesis by the fact that each half portion of the spectrum converge to the \( L_2 \)-regularization rate \( \lambda \) (rather than zero). In this particular case, we have \( \lambda_{K/2} = 0.011137 \) and \( \lambda_{P-K/2} = 0.009579 \).

![Figure 4: A log-scale eigenvalue magnitude spectrum showing the first \( \frac{K}{2} \) Hessian eigenvalues from each end of the full spectrum.](image)

4.2.3 Closing the Gap

Based on the observations in the previous section, we now propose a partitioning of the Hessian eigen-decomposition which reveals that an approximation of the positive curvature subspace of the covariance matrix can be obtained without explicitly requiring to compute any of the \( P - K \) Hessian eigenvectors and eigenvalues in the gap.
The eigendecomposition of the Hessian matrix can be partitioned as follows,

\[
H(\hat{\omega}) = H_{\text{pos}} + H_{\text{gap}} + H_{\text{neg}} \in \mathbb{R}^{P \times P}
\]

\[
= Q_{\text{pos}}\Lambda_{\text{pos}}Q_{\text{pos}}^T + Q_{\text{gap}}\Lambda_{\text{gap}}Q_{\text{gap}}^T + Q_{\text{neg}}\Lambda_{\text{neg}}Q_{\text{neg}}^T,
\]

where the subscripts ‘pos’, ‘gap’ and ‘neg’ refer to Figure 5 and denote eigenvectors and corresponding eigenvalues according to whether they belong to the left of the zero crossing, in the gap, or to the right of the zero crossing. Accordingly, we have that \(Q_{\text{pos}} \in \mathbb{R}^{p \times (P^++P)}\), \(\Lambda_{\text{pos}} \in \mathbb{R}^{(P^++P) \times (P^++P)}\), \(Q_{\text{gap}} \in \mathbb{R}^{P \times (P-K)}\), \(\Lambda_{\text{gap}} \in \mathbb{R}^{(P-K) \times (P-K)}\), \(Q_{\text{neg}} \in \mathbb{R}^{P \times (P-K)}\) and \(\Lambda_{\text{neg}} \in \mathbb{R}^{(P-K) \times (P-K)}\). Furthermore, we ignore the principal directions in the cost landscape of negative curvature (if any) by dropping the last term in the right hand side of (17), and so the covariance matrix \(\Sigma\) can be approximated by,

\[
\hat{\Sigma} = \frac{1}{N} \left[ Q_{\text{pos}}\Lambda_{\text{pos}}^{-1}Q_{\text{pos}}^T + Q_{\text{gap}}\Lambda_{\text{gap}}^{-1}Q_{\text{gap}}^T + Q_{\text{neg}}\Lambda_{\text{neg}}^{-1}Q_{\text{neg}}^T \right] \in \mathbb{R}^{P \times P}.
\]

If \(\lambda_{\frac{K}{2}} \approx \lambda_{P-\frac{K}{2}} \approx \lambda\), we can approximate (18) by

\[
\begin{array}{c|c|c|c}
\text{Positive } \frac{K}{2}-\text{tail} & \text{Gap} & \text{Negative } \frac{K}{2}-\text{tail} \\
\hline
\lambda_k > 0 & \lambda_k \approx \lambda & \lambda_k > 0 & \lambda_k < 0 \\
\hline
\lambda_{\text{gap}} & - & - & - \\
\hline
\lambda_{\text{gap}} - \lambda_{\epsilon} & - & - & - \\
\end{array}
\]

Figure 5: Partitioning of the Hessian eigendecomposition.

replacing \(\Lambda_{\text{gap}}\) with a scalar term \(\lambda_{\text{gap}} \approx \lambda\). With reference to Figure 5 and since the eigenvalues in the gap are sorted in decreasing order (Section 4.1), there are now two possible extreme conditions: 1) when all the eigenvalues in the gap are equal to \(\lambda_{\frac{K}{2}}\) (blue), or 2) when all the eigenvalues in the gap are equal to \(\lambda_{P-\frac{K}{2}}\) (green). Hence, we can get a bound on the error as follows. Let \(\lambda_{\text{gap}} \pm \lambda_{\epsilon}\) be defined by,

\[
\lambda_{\text{gap}} \pm \lambda_{\epsilon} = \frac{\lambda_{\frac{K}{2}} + \lambda_{P-\frac{K}{2}}}{2} \pm \frac{\lambda_{\frac{K}{2}} - \lambda_{P-\frac{K}{2}}}{2} \in \mathbb{R},
\]

then,

\[
\tilde{\Sigma} \pm \Delta^2 = \frac{1}{N} \left[ Q_{\text{pos}}\Lambda_{\text{pos}}^{-1}Q_{\text{pos}}^T + (\lambda_{\text{gap}} \pm \lambda_{\epsilon})^{-1}Q_{\text{gap}}Q_{\text{gap}}^T \right].
\]

Since we are dealing with an orthonormal basis, we can use the identity,

\[
Q_{\text{pos}}Q_{\text{pos}}^T + Q_{\text{gap}}Q_{\text{gap}}^T + Q_{\text{neg}}Q_{\text{neg}}^T = I \in \mathbb{R}^{P \times P},
\]

and we see that it is possible to express (20) without having to compute any of the eigenvectors in the gap because,

\[
Q_{\text{gap}}Q_{\text{gap}}^T = I - Q_{\text{pos}}Q_{\text{pos}} - Q_{\text{neg}}Q_{\text{neg}} \in \mathbb{R}^{P \times P}.
\]

Furthermore, inserting (20) into (12) yields the final form of the prediction variance approximation,

\[
\tilde{\sigma}(x_0, \hat{\omega}) \pm \delta^2 = \text{diag} \left\{ F \left[ \Sigma \pm \Delta^2 \right] F^T \right\} \in \mathbb{R}^{T_L},
\]

where the associated error \(\delta^2\) is given by,

\[
\delta^2 = \frac{\lambda_{P-\frac{K}{2}} - \lambda_{\frac{K}{2}}}{2N} \text{diag} \left\{ FQ_{\text{gap}}Q_{\text{gap}}^TF^T \right\} \in \mathbb{R}^{T_L},
\]

In practice, by distributing the \(F\) matrices in \(20\) and \(24\), these quantities can be computed without forming any \(\mathbb{R}^{P \times P}\) matrices. Furthermore, the approximate prediction uncertainty is given by,

\[
\tilde{\sigma}(x_0, \hat{\omega}) \pm \epsilon^2 = \sqrt{\tilde{\sigma}^2(x_0, \hat{\omega}) \pm \delta^2} \in \mathbb{R}^{T_L},
\]

with the corresponding error \(\epsilon^2\) given by,

\[
\epsilon^2 = \frac{1}{2} \left( \sqrt{\tilde{\sigma}^2(x_0, \hat{\omega}) + \delta^2} - \sqrt{\tilde{\sigma}^2(x_0, \hat{\omega}) - \delta^2} \right).
\]

Lastly, we define an ‘uncertainty score’ by summing the approximate prediction variance per class output (total variance), and then take the square root to get the (total) prediction uncertainty,

\[
\tilde{\sigma}_{\text{score}}(x_0, \hat{\omega}) \pm \epsilon^2_{\text{score}} = \sqrt{\sum_{m=1}^{T_L} \tilde{\sigma}^2_m(x_0, \hat{\omega}) \pm \epsilon^2_m} \in \mathbb{R},
\]
with the corresponding error \( \epsilon_{\text{score}}^2 \) given by,

\[
\epsilon_{\text{score}}^2 = \frac{1}{2} \left( \sqrt{\frac{1}{L} \sum_{m=1}^{L} \tilde{\sigma}_m^2(x_0, \hat{\omega}) + \delta_m^2} - \sqrt{\frac{1}{L} \sum_{m=1}^{L} \tilde{\sigma}_m^2(x_0, \hat{\omega}) - \delta_m^2} \right) \in \mathbb{R}.
\]  \tag{28}

5 Demonstration and Proof of Concept

In the following sections we explore and demonstrate the approximate prediction uncertainty governed by (25) for the convolutional neural network based MNIST classifier that was introduced in section 4.2.1. For relevant code sections or results, we give explicit references denoted by (#x.y.) where x.y. corresponds to the markdown heading number in the Jupyter notebook \texttt{pydeepdelta \_demo.ipynb} included in the \texttt{pydeepdelta} Github repository.

5.1 The Classifier’s Distribution of Approximate Prediction Uncertainty

We compute (25) and (26) for \( K = \{2500, 3000, 4000, 5000\} \) for all the 60,000 plus 10,000 images in the official MNIST training and test sets, respectively (#2.5, #2.6, #2.9, #2.10). The error is shown in Figures 6a and 6b. We observe that the mean error diminish as \( K \) increases, and is for \( K = 5000 \) in the order of \( 10^{-6} \). For the rest of the discussion, we therefore focus on the \( K = 5000 \) approximation.

The distribution of approximate prediction uncertainty for all the images in the training set is shown in Figure 7a (#2.6). The corresponding plot is shown for the test set in Figure 7b (#2.10). We observe that there is a strong correlation between uncertainty and probability: the uncertainty is close to zero for probabilities close to zero and one, and peaks when the probability is around 0.5.

In Figures 8a (#2.6) and 8b (#2.10), we zoom in and apply local smoothing on the true positives and false positives. We observe that the prediction uncertainty seems to be higher for false positives than for true positives. This suggests that there is supplementing information in the uncertainty measure not captured by the probability alone.

5.2 Ranking Images Based On the ‘Uncertainty Score’

We propose to validate our results by studying the MNIST images with the maximum and minimum amount of total prediction uncertainty as defined in (27). The idea is based on the following reasoning: if a neural network classifies an image with low ‘uncertainty score’, the image should be easy to classify also for a human. Conversely, if the neural
Figure 7: The distribution of approximate prediction uncertainty in the MNIST training set (a) and test set (b). There are 10 bullet points per image \(x_n\), because there are \(T_L = 10\) classes (’0’-’9’). For a correctly classified image, there will be one true positive and nine true negatives. For a misclassified image, there will be one false positive, one false negative and eight true negatives.

network classifies an image with a high ‘uncertainty score’, the image should be hard to classify. Effectively, the ‘uncertainty score’ ranks images according to the degree of ‘doubt’ expressed by the neural network.

The 15 images in the training set with highest ‘uncertainty scores’ are shown in ascending order in Figure 9a (#2.7), while the 15 images in the training set with lowest ‘uncertainty scores’ are shown in descending order in Figure 9b (#2.8). The corresponding plots are shown for the test set in Figures 9c (#2.11) and 9d (#2.12). It is evident that the ‘uncertainty score’ provides a good quantification of the degree of doubt most humans would experience if they were asked to classify the images shown. We note that the true-positive rate within the top 15 group of images in the training set is at 50%, while in the bottom 15 group, it is at 100%. For the test set, the true-positive rate within the top 15 group of images is at 66%, while in the bottom 15 group, 100%.
5.3 Layer-wise Approximations

The layer-wise topology of neural networks naturally offers for uncertainty approximation in subsets of layers. If the parameter vector is stacked according to Section 2.2, the block diagonal matrices of the full Hessian matrix will correspond to layer-wise Hessian matrices.

It turns out that the classifier’s output layer Hessian matrix is positive definite. This allows us to prove the concept also in the positive definite case. Furthermore, we compare the output layer approximation to the full model approximation discussed in the previous section. We note that (13) has the flexibility to focus on any subsets of layers (#3.1), and that all governing equations still hold, except from that the parameter vector \( \hat{\omega}^{(6)} \) (Section 2.2) is used in place of \( \hat{\omega}^{(6)} \).

The output layer’s Hessian eigenvalue spectrum is shown in Figure 11 (#3.1–#3.4). Here \( P^{(6)} = P^{+^{(6)}} = 650 \), but we select \( K = 648 \) because it is the closest even alternative given that the Lanczos iteration requires that \( K < P \). Consequently, there are only \( P^{(6)} - K = 2 \) ‘missing’ positive eigenvalues in the gap. Furthermore, we note that \( \lambda^{\frac{K}{2}} = 0.0100000007 \) and \( \lambda^{P-K} = 0.0000000007 \), so we can expect the errors (24) to be zero up to machine precision. In Figures 10a (#3.5, #3.6) and 10b (#3.7, #3.8) we compare the output layer approach to the full model. We note that there is a surprisingly good correlation between the two, and we summarize the correlation coefficients in Table 1. The output layer approximation shows a lower overall level of uncertainty, but this is expected since the uncertainty must increase or stay at the same level when the effective \( P \) grows. However, since the Hessian matrix of the full model is not positive definite, the uncertainty approximations for the full model cannot be regarded as absolute in the first place, therefore the importance of this offset diminish.
Figure 10: Comparison of uncertainty approximations for the output layer (vertical axes) versus full model (horizontal axes).

Figure 11: A log-scale eigenvalue magnitude spectrum showing the first $K$ Hessian eigenvalues from each end of the full spectrum of the output layer.

6 Summary and Concluding Remarks

We have explored the classical Delta method in a deep learning classification context. By an approximation of the positive curvature subspace of the Hessian matrix, we have shown that reasonable uncertainty approximations can be obtained even when the number of utilized Hessian eigenpairs is much lower than the number of model parameters.

This result is promising since the sheer size of the Hessian matrix seems to be the main reason why the Delta method has not received much attention in deep learning.

We have observed that prediction uncertainty in the classification context provides supplementing information to the traditional measure of probability. In particular, we have observed that false positives seem to have a higher prediction uncertainty than true positives. Interestingly, this suggests that uncertainty quantification can be used to distinguish true positives from false positives.

Finally, we observe that uncertainty approximation based only the output layer is surprisingly well correlated with full model approximations. This result is important because the computational burden to compute the Hessian eigendecomposition of a model’s output layer will generally be orders of magnitude lower compared to a full model. However, more research is required to conclude that this result holds for all models and all datasets.
References

[1] 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. Software available from https://www.tensorflow.org/, 2015.

[2] G. Alain, N. L. Roux, and P.-A. Manzagol. Negative eigenvalues of the hessian in deep neural networks. https://arxiv.org/abs/1902.02366, arXiv:1902.02366v1 [cs.LG] 6 Feb 2019., 2019.

[3] B. Ghorbani, S. Krishnan, and Y. Xiao. An investigation into neural net optimization via hessian eigenvalue density. https://arxiv.org/abs/1901.10159, arXiv:1901.10159v1 [cs.LG] 29 Jan 2019., 2019.

[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] 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.

[6] Y. LeCun, P. Y. Simard, and B. Pearlmutter. Automatic learning rate maximization by on-line estimation of the hessian's eigenvectors. http://yann.lecun.com/exdb/publis/pdf/lecun-simard-pearlmutter-93.pdf, Advances in Neural Information Processing Systems (NIPS 1992), 1993.

[7] W. K. Newey and D. McFadden. Handbook of econometrics. https://www.sciencedirect.com/science/article/pii/S1573444120580054, 1994.

[8] 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] 14 May 2019., 2019.

[9] 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.

[10] L. Sagun, L. Bottou, and Y. LeCun. Eigenvalues of the hessian in deep learning: Singularity and beyond. https://arxiv.org/abs/1611.07476, arXiv:1611.07476v2 [cs.LG] 5 Oct 2017., 2017.

[11] L. Sagun, U. Evci, V. U. Guney, Y. Dauphin, and L. Bottou. Empirical analysis of the hessian of over-parametrized neural networks. https://arxiv.org/abs/1706.04454, arXiv:1706.04454v3 [cs.LG] 7 May 2018., 2018.

[12] Arpack. http://li.mit.edu/Archive/Activities/Archive/CourseWork/Ju_Li/MITCourses/18.335/Doc/ARPACK/Lehoucq97.pdf.

[13] pydeepdelta. https://github.com/gknilsen/pydeepdelta.git.

[14] Scipy. http://www.scipy.org.

[15] L. N. Trefethen and D. B. III. Numerical linear algebra. Siam, 1997.

[16] S. Watanabe. Almost all learning machines are singular. http://watanabe-www.math.dis.titech.ac.jp/users/swatanab/foci2007.pdf, Proceedings of the 2007 IEEE Symposium on Foundations of Computational Intelligence, 2007.