Bachelorarbeit

Where is the Uncertainty in Bayesian Neural Networks (Laplace Approximation)

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Methods of Machine Learning
Moritz Kniebel, moritz.kniebel@student.uni-tuebingen.de, 2020

Bearbeitungszeitraum: von-bis

Gutachter: Prof. Dr. Philipp Hennig, Universität Tübingen
Betreuer: Marius Hobbhahn, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

My Name (Matrikelnummer 123456), November 5, 2020

Abstract

Template

Acknowledgments

If you have someone to Acknowledge ;)

Contents

1. Introduction

What is this all about?

Cite like this: [AFS+11]

TODO: start catchy, context of study, to whom important/beneficial?, personal motivation, aims and objective, structure of thesis (later) The field of Artificial Intelligence and Machine Learning is one of the fastest-growing, not only in Computer Science but also compared to everything else in the world economy. The Growth is due to the shift from mere academic to real-world applications. Some examples of real-world applications are optimization algorithms to solve complex problems, advanced screening and diagnosis of malicious diseases, and virtual assistants, which are constantly improved to yield better results. The said shift demands some new quantities to AI, such as better accuracy and a bound of confidence to rely on the used algorithms.
The field of Machine Learning, i.e. Deep Learning, has made huge steps in this direction, by yielding results that previously were not considered possible. This progress was mainly achieved through the usage of Deep Neural Networks. Although the results are good, it's hard to gain intuition about what happens between the input and the output. With its complex structure and a huge number of parameters the DNN can be compared to a black box. Besides that, the Predictions of Deep Neural Networks are usually point estimates. This creates problems, as the uncertainties of the associated weights and outputs are unknown. Especially in safety-critical applications, such as autonomous driving, having a posterior distribution can improve decision-making. Bayesian Neural Networks (BNNs) describe methods that apply uncertainty to Neural Networks. These methods provide a principled framework for several applications, including uncertainty estimation. With this information, every output and weight can be quantified with a bound of confidence. This gives the algorithm an introspective quality, knowing what it doesn't know and what it knows and how certain it is. A system, such as a self-driving car, can benefit from uncertainty estimates in decision-making. For example to break, or not to break. Therefore, it's important to know, how much the system can rely on the model's prediction. Consequently, it is desirable, that a trained model has high confidence in safety-critical regions.

2. Background

There are certain concepts, which are essential to follow the contents of this thesis. Therefore, in this chapter, the ones, which are used extensively will be explained sufficiently. The chapter will start with an overview of Neural Networks (NNs). Due to the extensive use of Convolutional Neural Networks in this thesis, an introduction to this network class will extend this overview. Next, an overview of uncertainty in Deep Learning is given, which includes an introduction to Bayesian Neural Networks. The chapter will be completed by a brief presentation of the BackPACK package for PyTorch since all uncertainty estimates are received by making use of it.

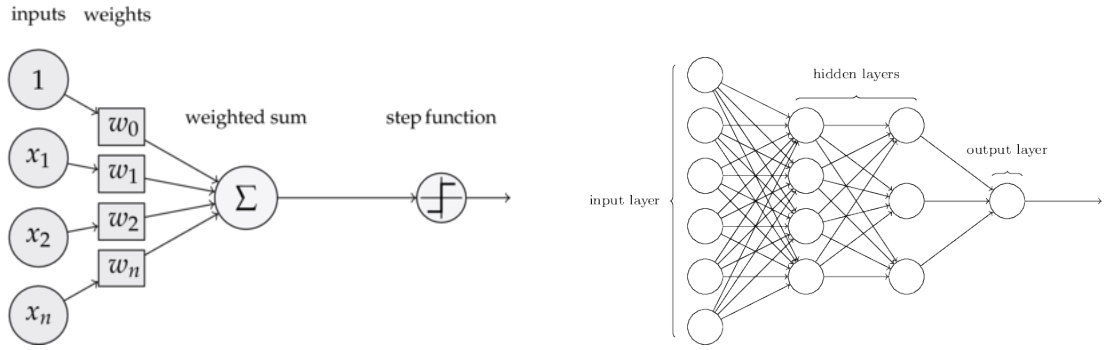
2.1. From Perceptron to Deep Neural Network

Neural Networks can be seen as non-linear, parameterized functions. Nevertheless, because of their complex structure, they are often referred to as black boxes. Given an arbitrary input, a trained network yields a point estimate for a possible target. But with thousands, if not millions, of parameters it is hard to get an intuition of what happens between input and output. In this section, a short overview of Neural Networks will be presented. Starting from the smallest component, the Perceptron, through the multilayer perceptron (MLP) to Deep Neural Networks (DNNs).

2.1.1. The Perceptron

The Perceptron is a classifier for binary tasks in the setting of supervised learning. Given an input-vector of size N, it returns one of two possible outputs. Every function that can be represented by the Perceptron can be learned in a finite time. For the single Perceptron, only linear separable functions can be represented. Inspired by a neuron in the human brain, it takes inputs and fires an output. This behavior can be modeled mathematically. All inputs x_i are weighted and then summed up. One of the inputs, the bias b, is a constant with a value of 1. The bias helps the Perceptron to control the activation function, which provides more flexibility and helps to generalize on the input data. The aggregated input v is given into an activation function ϕ, such as the rectified linear unit (ReLU), which returns the output o (Figure 2.1a).

$$v = \sum_i^N w_i x_i$$
$$o = \phi(v)$$



(a) (b)

Figure 2.1.: (a): Schematic representation of the Perceptron [1]. (b) Structure of MLP with two hidden layers [2].

By arranging them layer-wise, Perceptrons can be combined into a feed-forward-network, the MLP (Figure 2.1b). It at least consists of three layers, which are an input layer followed by a hidden layer followed by the output layer. MLPs can classify more complex tasks as the number of layers increases. MLPs with one hidden layer can classify convex polygons. By adding a second hidden layer, the MLP can classify sets of arbitrary form.
The MLP is the quintessential example of a Deep Neural Network (DNN). Nevertheless, in today's variety of different deep learning models, the MLP is only a subset of DNNs.

2.1.2. Convolutional Neural Networks

The Convolutional Neural Network is a popular class of DNNs for image-related tasks, such as image classification. The approach is based on applying filters on the input image. Since every pixel in an image is of different importance for a specific task, filtering is a suitable concept. Looking upon a handwritten digit, several features are decisive to classify it. For example, digit one consists of a somewhat vertical line and maybe an added dash on the top. The filters also referred to as kernels, are realized by a m x n matrix, with feature specific values in every field. A simple filter for a horizontal line could have value 1 throughout every entry of a single row and value 0 for every other entry. The filters are applied traversing over every possible m x n segment of the image. Every segment is multiplied element-wise with the filter, and the results are summed up, as described in Figure 2.2. Once the process is fished a filtered image, the convoluted feature is returned. There are also specific filters for average pooling and max pooling. For those, the mean value or the largest value of

[2]Figure adopted from https://blog.dbrgn.ch/2013/3/26/perceptrons-in-python/
[2]Figure adopted from https://github.com/rcassani/mlp-example

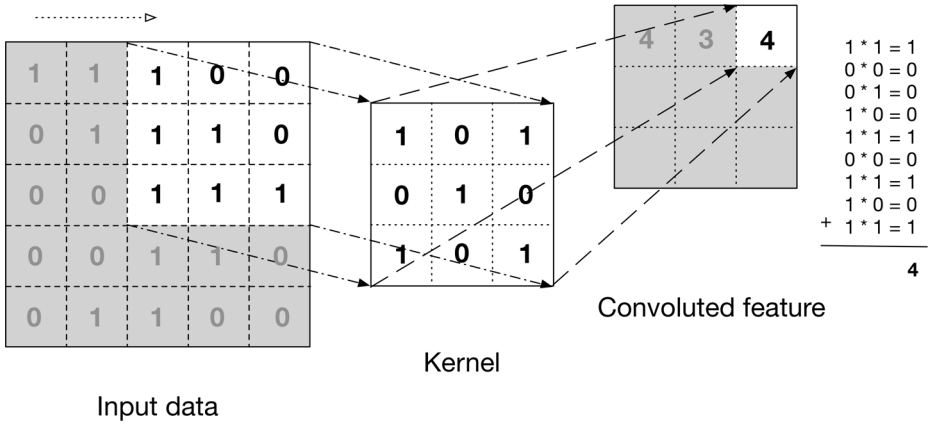


Figure 2.2.: Applying a 3x3 kernel on a 5x5 input. This results in an 3x3 convoluted feature [3].

a segment is returned.
CNNs consist of several layers in sequential order. Each of these includes a defined number of filters. From this, it follows that the input image gets filtered as it travels through the network, whereby the input of every layer is the convoluted feature of its previous layer. Now one can ask how the CNN learns from the input data. The filters are adjusted to learn specific features during training. As the number of iterations increases, the filters get more complex. So it happens that filters that were simpler in the beginning, like the horizontal filter mentioned earlier, evolve to detect complex structures, such as geometric shapes. In the example of handwritten digits, late filters learned to detect whole digits, as visualized in Figure 2.3. The architecture of CNNs always combines three types of layers, the convolutional layers (ConvLayers), the pooling layers (e.g., maxpool, avgpool), and the dense or fully connected layers. These layers are put together in sequential order to form a CNN architecture (Figure 2.4). The weights are trained by backpropagation. In the course of this, every filter is trained to focus on specific features to determine the best accuracy. The trained network transforms an input image layer by layer, which can lead to down-sampling, from original to final classification (Figure 2.3).

2.2. Uncertainty in Deep Neural Networks

NNs can learn observed data to perform different kinds of tasks, such as regression. A trained NN performs well for data inside or similar to the training data. The predictions for such data are accurate and assigned with high confidence. In contrast, NNs should predict with low accuracy and confidence for data far away from the

[3] Figure adopted from [PG17]
[4] Figure adopted from https://towardsdatascience.com/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9

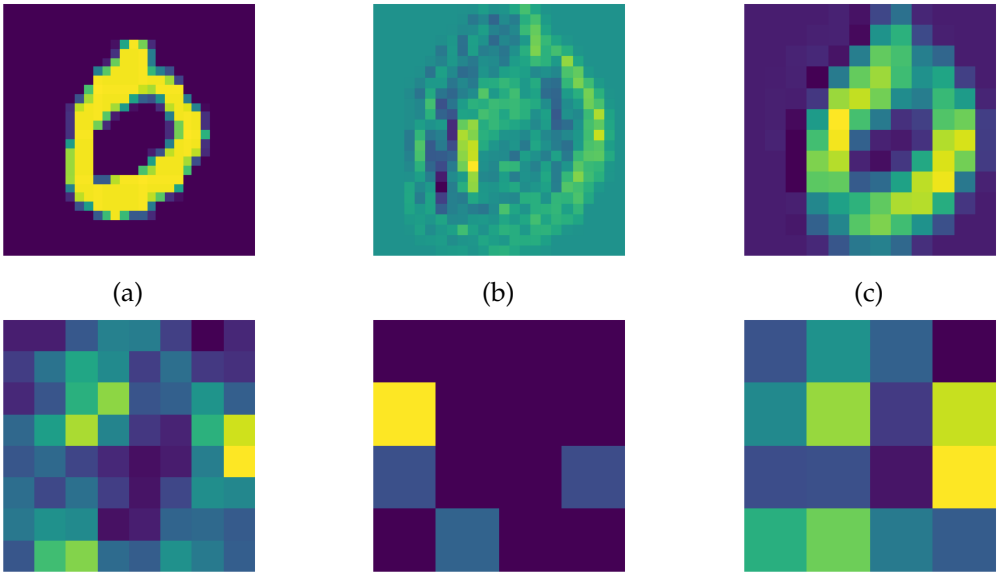


Figure 2.3.: A CNN was trained on the MNIST dataset. The intermediate activations for a specific filter were visualized for every layer. (a) The input image. (b) The feature map returned by the first ConvLayer. (c) The feature map returned by the second ConvLayer. (d) The feature map returned by the third ConvLayer. (e) The feature map returned by the fourth ConvLayer. (f) The feature map returned by the linear layer.

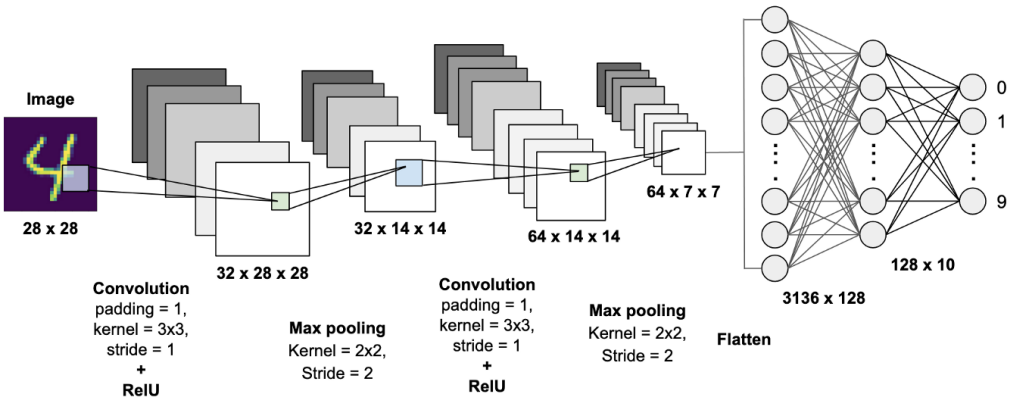


Figure 2.4.: CNN architecture with two ConvLayers, Maxpooling and two linear layers [4].

training data. But in fact, NNs tend to be overconfident for out-of-distribution data. At the same time, NNs can easily be fooled by an input. An added perturbation leads to misclassification of an image. That misconception entails severe problems in safety-critical applications, such as autonomous driving or medical use-cases. Real-world data inherits variation, whereby events far away from the training data naturally occur. An example of such an event could be a self-driving car entering a new environment. To use NNs for real-world applications, they need the ability to tell how certain or rather uncertain they are with a prediction. There are two types of uncertainty, which are present in deep learning.

- *epistemic uncertainty* is knowledge uncertainty. I.e., the uncertainty in the parameter-space and structure of a model, observing dataset D.

- *aleatoric uncertainty* is the inherent variation in the observations, the intrinsic uncertainty of the world they take place in. (stochastic uncertainty). One example is noisy data.

A probabilistic view on machine learning integrates both types of uncertainty. Supplementing machine learning, i.e., deep learning with Bayesian statistics integrates the uncertainty mentioned above into a DNN.

2.3. Bayesian Modeling

2.3.1. Bayesian Statisitcs

Bayesian statistics describe the application of probability to statistical problems. Encountering new evidence modifies our beliefs about the occurrence of an event. For example, our belief about the upcoming weather updates when we see clouds in the far. Bayesian statistics provide mathematical tools to update the possibility of event A in light of seeing new data D or a particular event B. In contrast to frequentist statistics, Bayesian statistics do not eliminate uncertainty by providing estimates. Instead, uncertainty is an essential aspect of Bayesian statistics as it is preserved and refined. Bayesian statistics' mathematical basis is Bayes' theorem. It derives from the probability of two events A and B happening, $P(A \cap B)$:

$$
\begin{aligned}
&P(A \cap B) = P(B|A)P(A) \\
&P(B \cap A) = P(A|B)P(B) \\
&P(A \cap B) = P(B \cap A) \\
&\Rightarrow P(A|B)P(B) = P(B|A)P(A)
\end{aligned}
$$

By dividing both sides by $P(B)$ we derived Bayes' theorem :

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)},$$

where $P(A|B)$ is the probability of interest, called posterior probability. It is the product of the likelihood $P(B|A)$ and the prior probability $P(A)$ divided by the evidence $P(B)$. This theorem is of fundamental importance not only for Bayesian Deep Learning but also in data science and statistics.

2.3.2. Bayesian Inference

In probability theory inference describes the process of reasoning a probability by analyzing new data. Therefore, Bayesian inference leverages Bayes' theorem to deduce probabilities from given data. Recall Bayes' theorem:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)},$$

where events A and B have prior probabilities $p(A)$ and $p(B)$. For now, the probabilities of A and B were exact values. But for several applications, such as deep learning models, probability distributions are better representations of A and B. For deep learning models, the notation of Bayes' theorem changes:

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)},$$

where θ denotes the parameter space of the used model, and D is the observed data. Bayesian inference adjusts the prior distribution $p(\theta)$ after observing data D. This adjustment forms the posterior distribution $p(\theta|D)$. Comparing the posterior distribution on the prior distribution shows the effect of the new data D. The most probable value in the posterior distribution is called the maximum a posteriori (*MAP*). However, the MAP will not have a probability of 100%. Thus, the posterior distribution holds information on how uncertain the model is about the outcome. The posterior distribution also gives the possibility to perform the further inference. Given a new data point x^* the output is predicted by integrating:

$$p(y^*|x^*, D) = \int p(y^*|x^*, \theta)p(\theta|D)d\theta$$

Having a posterior distributions brings great possibilities to deep learning. Analytically the posterior distribution has its downsides, though.

2.3.3. The Intractable Posterior

The posterior distribution for deep learning models is obtained by invoking Bayes' theorem. Practically the posterior distribution is analytically intractable in most cases. I.e., it is only tractable for simple models, such as Bayesian linear regression. The intractability of complex models is due to marginalization, which is the key

to a meaningful posterior distribution. The model evidence $p(D)$ is responsible for marginalization by integrating:

$$p(D) = \int p(D|\theta)p(\theta)d\theta$$

This integral gets analytically intractable, as it includes all possible parameter values for θ. The number of all possible values for θ grows exponentially with the model complexity. Since we are still interested in the properties of the posterior distribution, an approximation is needed.

2.3.4. Approximate Inference

2.4. Bayesian Neural Networks

A Bayesian Neural Network is not specifically another class of DNN but integrates a given DNN into a bayesian Framework (MacKay). This framework comprises a set of tools to transform a DNN into a probabilistic model. Thus, a CNN, as well as an MLP, could be turned into a BNN. The difference is made in how the weights w of a network are treated. When integrated into the Bayesian framework, a prior distribution is placed over the network's space of weights. By observing a dataset D, the weights turn into a distribution of possible values during training. Thus, the prediction of a BNN generates a posterior distribution $p(w|D)$, which is obtained using Bayes theorem. Consequently, every prediction has a probabilistic guarantee.

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)}$$

To gain intuition, a BNN can be seen as an ensemble (2.2) of all possible models with given architecture after observing dataset D. All predictions of this ensemble form the posterior distribution, a measure of predictive uncertainty.

2.5. On BackPACK for PyTorch

BackPACK ([DKH20]) is a library for PyTorch, a well-known framework for machine learning. It extends the backwards pass of a neural network to compute additional information. Such information could be the individual gradient of a minibatch or in the context of this thesis curvature approximations of the Hessian [5]. To receive said information only a few lines of code are necessary. The package made it easy to integrate DNNs into the Bayesian framework and gather uncertainty information.

[5] BackPACK website: https://backpack.pt

3. Related Work

Since the early 1990s, Bayesian Neural Networks gained attention, and the Bayesian toolbox is used broadly in connection with deep learning. Many approaches for approximate inference in Bayesian Neural Networks were proposed. This chapter gives a high-level summary of some of the most popular methods to get an idea of approximate inference possibilities.

3.1. Variational Inference

The practical use of Bayesian inference bears some computational difficulties. Among them is the intractability of the posterior distribution, which section 4.2. will discuss. Variational inference (VI) describes a method to overcome those difficulties by approximating the posterior distribution $p(w|D)$. A family of distributions $q(w)$ parameterized with θ is defined to approximate the posterior distribution. The goal is to find the setting of parameters θ^* that minimizes the difference between q and p. The Kullback-Leibler divergence (KL) formalizes the difference between the two distributions as:

$$KL(q(w|\theta)||p(w|D)) = \int q(w|\theta)\ \log \frac{q(w|\theta)}{p(w|D)} dw$$

The parameters in q determine the difference to p. This formulates the objective minimization-problem:

$$\begin{aligned}\theta^* &= \arg\min_\theta \int_\theta q(w|\theta)\ \log \frac{q(w|\theta)}{p(w|D)} dw \\ &= \arg\min_\theta \int_\theta q(w|\theta)\ log \frac{q(w|\theta)}{p(D|w)p(w)} dw\end{aligned}$$

In this form, the minimization is not possible due to the direct dependence on the posterior, which is still intractable. But the KL has some properties that help to solve it:

- $\forall p, q : KL(q||p) > 0$
- $KL(q||p) = 0 \iff q = p$
- $\log p(D) = KL(q(w|\theta)||(p(w|D)) + \mathbb{E}_{q(w|\theta)}\left[\log p(D, w) - \log q(w|\theta)\right]$,

Decomposing the log evidence reveals the equivalence of minimizing the KL and maximizing the ELBO. As the ELBO and KL add up to the log evidence and the KL is non-negative, the KL is minimal when the ELBO is maximal. Maximizing the

ELBO is computationally possible, as it only depends on q. This reformulates the above minimization problem to:

$$\theta^* = \arg\max_\theta \mathbb{E}_{q(w|\theta)} [\log p(D, w) - \log q(w|\theta)]$$

The reparametrization trick makes it possible to optimize a models parameters since we cannot back-propagate through a stochastic node. By moving the parameters outside of the distribution function, a gradient can be computed for the parameters. Aforementioned is realized by defining a deterministic function $t(\mu, \sigma, \epsilon)$, which takes samples from a parameter-free distribution $\epsilon \sim \mathcal{N}(0, 1)$ and transforms them. This method makes it possible to calculate a gradient for $t(\mu, \sigma, \epsilon)$, as its independent from parameters. The transformation is defined as

$$t(\mu, \sigma, \epsilon) = \epsilon \odot \sigma + \mu \ ,$$

where $\odot$ is element-wise multiplication.

3.2. Bayes by Dropout

Dropout is a technique to improve the training of NNs, which was first introduced by JMLR:v15:srivastava14a. Excluding random neurons during each iteration in training reduces the network's generalization error. A dropout is implemented by setting the weights of a chosen neuron to zero, virtually eliminating it from the network. Training with dropout optimizes a variety of sparse networks with shared weights on the same task. The network generalizes much better as dropout forces it to learn redundant and independent features.
gal2016dropout proposed to use dropout for Bayesian approximation by integrating dropout during training and testing. Random neurons are eliminated from the network for each input during test time. The network returns different outputs when feeding an identical input several times. For example, given the input $X = [3, 3, 3, 3]$ a possible output could be $O = [4.2, 4.0, 3.8, 4.1]$. Such output holds information about the uncertainty and can be compared to a set of MC-samples. Dropout during testing gives the possibility to form a predictive distribution from the outputs. Uncertainty measurements, such as the variance, can be estimated from this predictive distribution.

3.3. Deep Ensembles

Deep ensembles describe a simple method to assess model uncertainty using DNNs. The idea behind deep ensembles is to have several networks trained on the same task. In such way, the strengths of one network complement the weaknesses of another network and vice versa. This synergy goes back to Nicolas de Condorcet's famous jury theorem: If each juror (in a jury) has a probability greater than 50% to

decide correctly on a problem, then the probability of a correct jury verdict goes against 100%, as the size of the jury increases. Likewise, numerous networks with a mediocre validation accuracy know more about the data than one network with high accuracy.
Diversification in weight initialization, data shuffling, and architectures induce the differences between ensembled networks. Taking every members output into account gives insights into uncertainty, such as the variance. Deep ensembles perform notably well for out of distribution inputs. With its large number of parameters, a NN can represent observed data with many different functions. The variance within all possible functions' outputs is significant for out of distribution inputs, assigning high uncertainty to such inputs.

4. Theory

This work contains several experiments on the subject of uncertainty. The essential theory will be presented in this chapter to ensure a good understanding.
In section 4.1, we will begin with a theoretical look into Bayesian Inference to provide a good intuition for uncertainty. As it is crucial to know why approximate inference is needed for uncertainty estimation, the true posterior's intractability will be explained in section 4.2. Next, the Laplace approximation of the weights in NNs and the Hessians role will be introduced in section 4.3 and 4.4. respectively. Lastly, the alternative way to receive information about the uncertainty will be explained in section 4.5, which is the Kronecker factored approximate curvature (KFAC).

4.1. Laplace Approximation of the Weights in Neural Networks

The Laplace approximation is a way to approximate a posterior distribution with a Gaussian. Using it for Neural Networks was pioneered by [Mac92]. The method locates the mode of the posterior distribution by using optimization and places a Gaussian around the mode (Figure 4.1). The curvature of this Gaussian is given by the inverse of the covariance matrix w.r.t. the Loss L. The single steps will be explained in more detail to get a more specific intuition of Laplace approximation for NNs. The first step would be to locate the mode W_{MAP} of the posterior distribution via optimization, denoted as $\hat{\theta}$. This is done by merely training the network, whereby the optimizer seeks to minimize the loss through backpropagation:

$$\hat{\theta} = \arg\min_{\theta} L(\theta)$$

Therefore training the network will give us the mode $\hat{\theta}$. The next step is to perform a second-order Taylor expansion around the mode $\hat{\theta}$. This is where the approximation takes place:

$$log\ p(\theta|D) = log\ p(\hat{\theta}|D) + (\theta - \hat{\theta})^T\ \mathbf{J}(\theta - \hat{\theta}) - \tfrac{1}{2}(\theta - \hat{\theta})^T\ \mathbf{H}(\theta - \hat{\theta})$$

where the first-order term can be dropped since the gradient at W_{MAP} is equal to zero:

$$log\ p(\theta|D) = log\ p(\hat{\theta}|D) - \tfrac{1}{2}(\theta - \hat{\theta})^T\ \mathbf{H}(\theta - \hat{\theta})$$

, where $\mathbf{H}$ is the Hessian w.r.t. the loss $\mathbf{H}_{ij} = \frac{\partial^2 L}{\partial W_i \partial W_j}$, evaluated at W_{MAP}. The obtained

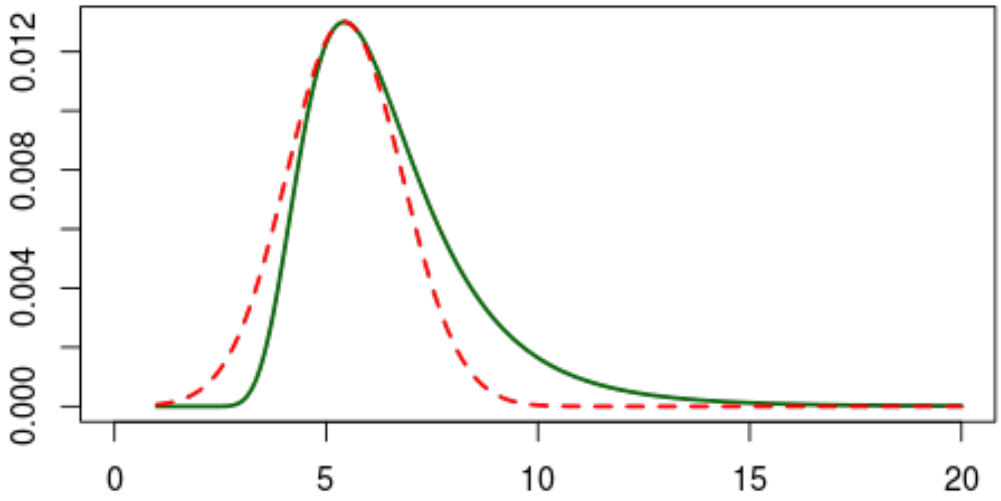


Figure 4.1.: Example of a Laplace approximation. The solid green line shows the posterior distribution that needs to be approximated. The red dotted line shows the approximated posterior, which is a Gaussian centered at the mode of the true posterior. [1].

approximation is of Gaussian functional form. This shows, that the true posterior distribution is approximated with a Gaussian centered at the mode, which curvature is given by the inverse of the Hessian **H**:

$$p(W|\mathcal{D}) \approx \mathcal{N}(\hat{\theta}, H^{-1}) ,$$

where $\mathcal{N}$ is the normal distribution. In this thesis we don't focus on the approximated posterior distribution itself, but on the curvature given by the inverse of the Hessian **H**. We use the curvature as a measure of uncertainty, since the magnitude of the values in **H** reflects the magnitude of the uncertainty.

[1] Figure adopted from https://www.r-bloggers.com/2013/11/easy-laplace-approximation-of-bayesian-models-in-r/

4.2. About the Hessian

The Hessian holds essential properties of the gradient. I.e., it holds information about the rate of change of the gradient. In a high-dimensional space, like the loss surface of a NN, the gradient has many directions. For each of which directions, the rate of change is different. For every direction, there is one row in the Hessian matrix (w.r.t. Loss L), which is defined as:

$$H_{ij} = \begin{bmatrix} \frac{\partial^2 L}{\partial W_1^2} & \frac{\partial^2 L}{\partial W_1 \partial W_2} & \cdots & \frac{\partial^2 L}{\partial W_1 \partial W_i} \\ \frac{\partial^2 L}{\partial W_2 \partial W_1} & \frac{\partial^2 L}{\partial W_2^2} & \cdots & \frac{\partial^2 L}{\partial W_2 \partial W_i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 L}{\partial W_j \partial W_1} & \frac{\partial^2 L}{\partial W_j \partial W_2} & \cdots & \frac{\partial^2 L}{\partial W_j^2} \end{bmatrix}$$

The entries of this matrix fall into two classes of second-order derivates. The diagonal entries are pure partial second-order derivatives, whereas the non-diagonal entries are mixed partial second-order derivatives. On account of a large number of parameters in DNNs computing, the full Hessian becomes infeasible due to the storage it would occupy. An approximation of the generalized Gauss-Newton (GNN) diagonal will be used in this thesis to avoid this problem. The GGN and the Hessian are equivalent if the associated NN uses piece-wise linear activation functions [DKH20]. As we will use ReLU networks, the equivalence holds in this work. The diagonal of the GNN gets approximated as [2]:

$$diag(G(\theta^{(i)})) \approx \frac{1}{N} \sum_{n=1}^{N} diag([(\mathbf{J}_{\theta^{(i)}} z_n^{(i)})^T \hat{S}(z_n^{(i)})]\,[(\mathbf{J}_{\theta^{(i)}} z_n^{(i)})^T \hat{S}(z_n^{(i)})]^T),$$

with

$$\Delta_f^2 \ell(f(x_n, \theta), y_n) = S(z_n^{(L)}) S(z_n^{(L)})^T,$$

where $z_n^{(L)}$ is the input of the last layer L.

4.3. Kronecker-factored Approximate Curvature (KFAC)

Looking at state of the art Neural Networks, with millions of weight parameters, the Laplace approximation proposed by [Mac92] reaches its limit. This is due to a huge amount of storage (several terabytes) the Hessian would occupy, as mentioned above. Another approximation besides the diagonal GNN was presented in [RBB18]. They approximate the Hessian by a block matrix of independent Kronecker factorized matrices. This brings some benefits to it, as Kronecker products reduce the computational complexity and can be inverted separately. The method's

[2] calculated with the backPACK package. More details in the appendix of [DKH20]

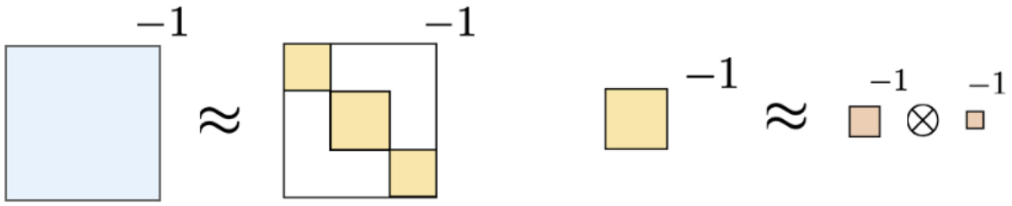


Figure 4.2.: visualization of KFAC. The Hessian (blue) gets approximated by block-diagonals. Each of the diagonal blocks is approximated by a krockecker product of two smaller matrices. Inverting the Hessian can be achieved by inverting the kronecker products [3].

idea is to block-diagonalize the Hessian, where every block corresponds to one layer of the associated NN. Each of those diagonal blocks is approximated by Kronecker-factorization (Figure 4.2). Thus, the sample Hessian H_λ of each layer λ is approximated with:

$$[H_\lambda]_{(a,b)(c,d)} \equiv \frac{\delta^2 L}{\delta W^\lambda_{a,b} \delta W^\lambda_{b,c}} = a^{\lambda-1}_b a^{\lambda-1}_d [\mathcal{H}_\lambda]_{a,c} \,,$$

where a_λ are the activation values of layer λ and $\mathcal{H}$ is the pre-activation Hessian of layer λ:

$$[\mathcal{H}_\lambda]_{a,b} = \frac{\delta^2 E}{\delta h^\lambda_a \delta h^\lambda_b} \,,$$

where h_λ is the pre-activation of layer λ. Additionally, we can denote the covariance of the incoming activations $a_{\lambda-1}$ as:

$$[\mathcal{Q}_\lambda]_{c,d} = (a^{\lambda-1}_c a^{T\lambda-1}_d)$$

With the approximated kronecker products $[\mathcal{Q}_\lambda]$ and $[\mathcal{H}_\lambda]$ the sample Hessian of each layer is:

$$H_\lambda = \frac{\delta^2 L}{\delta \mathrm{vec}(W_\lambda) \delta \mathrm{vec}(W_\lambda)} = (a_{\lambda-1} a^T_{\lambda-1}) \otimes \frac{\delta^2 L}{\delta h_\lambda \delta h_\lambda} = \mathcal{Q}_\lambda \otimes \mathcal{H}_\lambda,$$

where $\otimes$ denotes the Kronecker-product.
Since the layerwise Hessian is split into two Kronecker products, both of which should be described separately. $[\mathcal{Q}_\lambda]$ and $[\mathcal{Q}_\lambda]$ is the Hessian of the layerwise output w.r.t. the loss L. $[\mathcal{H}_\lambda]$ is the auto-correlation matrix of the layer-wise inputs [WZW+20]. With those Kronecker factors, the posterior of the weights in layer λ can be approximated as:

$$W_\lambda \sim \mathcal{MN}(W^*_\lambda, \mathcal{Q}^{-1}, \mathcal{H}^{-1}) \,,$$

where $\mathcal{MN}$ is the matrix Gaussian distribution.

[3] Figure adopted from https://towardsdatascience.com/introducing-k-fac-and-its-application-for-large-scale-deep-learning-4e3f9b443414

5. Experiments

5.1. Datasets

5.1.1. MNIST Dataset

The Modified National Institute of Standards and Technology database of handwritten digits or MNIST dataset for short is the primary benchmark for image classification. MNIST consists of 70000 black and white images split into 60000 training and 10000 testing images. Each image has a normalized size of 28×28 pixels and shows a handwritten digit between zero and nine at its center. Due to the varying styles of handwritten digits, the dataset gives robustness to a properly trained model. Therefore MNIST is gladly used for method comparison and entry-level machine learning examples. Figure x shows a set of images contained in the MNIST dataset.

Alternative MNIST

The MNIST dataset is extensively used for image-related tasks in machine learning. Complementary, there are some drop-in replacements for the MNIST dataset, namely: Extended MNIST (EMNIST), Fashion-MNIST (FMNIST), and Kuzushiji-MNIST (KMNIST). Those alternative datasets share specifications with MNIST but incorporate different content. EMNIST contains images of handwritten characters. FMNIST, developed by Zalando, picture various pieces of clothing. KMNIST images show phonetic letters of hiragana, a Japanese syllabary. A sample of each dataset is shown in Figure x.

5.1.2. SVHN

The Street View House Numbers (SVHN) dataset is a popular computer vision dataset containing real-world images (colored). The dataset is widely used to develop and benchmark machine learning and object detection algorithms, as it requires minimal data preprocessing and formatting. Taking pictures from a natural scene adds real-world complexity to the data. Therefore SVHN is the next step to simpler datasets, such as MNIST. The added difficulty is visible in lack of contrast normalization, overlapping digits, and distracting features. SVHN includes 73257 training and 26032 testing images (32×32 pixels) with an additional 531131 more simplistic images to extend the training set. The pictures in SVHN show real-world

photographs of house numbers, which differ in style, size, and perspective. Every house number contains numbers from zero to nine, which are labeled as ten distinct classes. An excerpt from SVHN is shown in Figure x.

5.1.3. CIFAR-10 and CIFAR-100

The Canadian Institute For Advanced Research (CIFAR) created two established datasets for object recognition and image classification tasks. Both CIFAR-10 and CIFAR-100 are subsets of the 80 Million Tiny Images dataset. The CIFAR datasets were developed to test and optimize machine learning, i.e., computer vision algorithms.

CIFAR-10 is more frequently used to benchmark image classification methods. The dataset consists of 60000 images, divided into 50000 training and 10000 testing images. A relatively low resolution of 32×32 pixels makes fast training and testing possible. The ten different classes in CIFAR-10 show vehicles like cars or airplanes and animals like cats and deers. Every class is independent, which means that there is no overlap existent in the images.
CIFAR-100 is similar to CIFAR-10, except the number of classes is raised to 100. Those classes consist of 20 superclasses, where each superclass contains five subclasses. An example of a superclass is reptiles containing the subclasses crocodile, dinosaur, lizard, snake, and turtle. To get an idea of both CIFAR datasets, Figure x and y show examples of CIFAR-10 and CIFAR-100, respectively.

5.1.4. Iris Dataset

The Iris Dataset is often used in classification and clustering tasks. It holds four attributes of three different iris-flowers: Iris setosa, Iris virginica, and Iris versicolor. The features that set the flowers apart are the length and width of petal and sepal. The usual classification task should identify the species from given attributes. Table x shows one set of features for each species.

5.2. Gaining Intuition

5.2.1. Setup

5.2.2. Ex 1

5.3. Moving to a Larger Network

5.4. The Perceptron and Multilayer Perceptron

6. Results

7. Conclusion

To conclude...

A. Blub

Bibliography

[AFS+11] Sameer Agarwal, Yasutaka Furukawa, Noah Snavely, Ian Simon, Brian Curless, Steven M. Seitz, and Richard Szeliski. Building rome in a day. *Commun. ACM*, 54(10):105–112, October 2011.

[DKH20] Felix Dangel, Frederik Kunstner, and Philipp Hennig. Backpack: Packing more into backprop. In *International Conference on Learning Representations*, 2020.

[Mac92] David J. C. MacKay. A practical bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472, 1992.

[PG17] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly Media, Inc., 1st edition, 2017.

[RBB18] Hippolyt Ritter, Aleksandar Botev, and David Barber. A scalable laplace approximation for neural networks. In *International Conference on Learning Representations*, 2018.

[WZW+20] Yikai Wu, Xingyu Zhu, Chenwei Wu, Annie Wang, and Rong Ge. Dissecting hessian: Understanding common structure of hessian in neural networks, 2020.