2. 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.

2.1. Variational Inference

Variational Inference approximates the posterior over the weights $p(W|\mathcal{D})$, where W denotes the weights, and D is the data. The method approximates the posterior using a variational distribution $q(W|\theta)$, parameterized with θ. The actual posterior distribution and the approximate distribution are of the same functional form. The approximating distribution $q(W|\theta)$ should get as close as possible to the true posterior p(w|D). This approximation is an optimization-problem since we seek the smallest difference between $p(W|\mathcal{D})$ and $q(W|\theta)$. The difference is reduced by minimizing the Kullback-Leibler divergence between p(W|D) and q(W|), while the parameters in $q(W|\theta)$ are estimated. In this way, Variational Inference approximates a good representation of the true posterior.

The Evidence Lower Bound and Reparametrization Trick

A problem comes up in minimizing the KL divergence, as it is directly dependent on the posterior, which is intractable. The reason for this intractability is discussed in section 4.2.. However, maximizing the evidence lower bound (ELBO) accomplishes the same as minimizing the KL. To clarify this equivalence the log evidence $\log p(D)$ can be decomposed as follows:

$$\log p(\mathcal{D}) = \mathcal{L}(\mathcal{D}, \theta) + KL(q(W|\theta)||p(W|\mathcal{D})) \,,$$

where $\mathcal{L}(\mathcal{D}, \theta)$ is 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. The key is that ELBO only depends on $q(W|\theta)$, making it possible to solve the difficulty of minimizing the difference between $p(W|\mathcal{D})$ and $q(W|\theta)$. Hence the new objective is to maximize the ELBO. 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.

2.2. Bayes by Dropout

There are many approaches to regularize the training process of NNs. Dropout-training can prevent the network from overfitting and hence helps to get better validation accuracy. Training with dropout regularizes the training process by deactivating or 'dropping out' a defined proportion of random neurons. The dropout happens in previously defined layers and during each training-iteration. A dropout is realized by setting the weights of selected neurons to zero, whereby their connections are practically eliminated from the network. In this way, the network changes in every iteration of training. Consequently, a set of similar but different networks is trained on the same task. Therefore a trained network is more robust and less overfitted w.r.t. the dataset.
[GG16] developed tools, which extract information from eliminated neurons for uncertainty estimation. All of which can yield uncertainty estimates without increasing computational complexity or decreasing validation accuracy (?). In their approach,[GG16] also utilized dropout during inference. Possible sets of weights are sampled from a Bernoulli distribution. For each forward pass, one of these samples is used in the network. By averaging over the results of performed stochastic forward passes, uncertainty estimates can be derived.

2.3. Deep Ensembles

A Bayesian formalism determines the majority of probabilistic methods to get uncertainty estimates from NNs. Not so a method developed by [LPB17]. They suggest using deep ensembles since they have shown good predictive performance. The idea behind deep ensembles goes back to Nicolas de Condorcet's famous jury theorem. The theorem says that if each juror (in a jury) has a probability greater than 50% to decide correctly on a binary problem, then the probability of a correct jury verdict goes against 100%, as the size of the jury increases. This idea is transferred to deep learning by training various models (e.g., different data, different architecture). The outputs of the trained models are averaged to form a single output. Following the jury theorem, this should boost the predictive performance of the ensemble.

With their work, [LPB17] first introduced utilizing deep ensembles for predictive uncertainty estimation. They encouraged their method by comparing it to state-of-the-art Bayesian approaches for uncertainty estimation. The crucial factors for good performance are a proper scoring rule and the use of adversarial training. Deep ensembles were compared to state-of-the-art (2017) Bayesian methods on a series of classification and regression benchmark datasets. With the right setup, deep ensembles matched or outperformed Bayesian methods, such as MC-dropout.

3. 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.

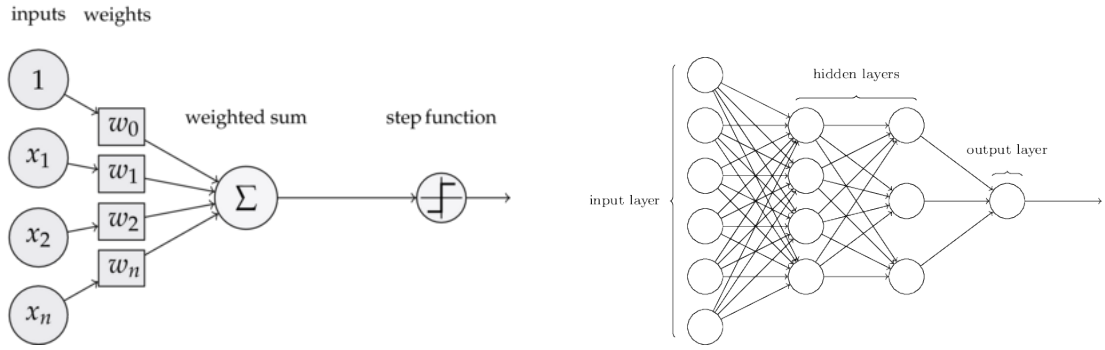
3.1. From Perceptron to Deep Neural Network

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. However, with hundreds and 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).

3.1.1. 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 3.1a).

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



(a) (b)

Figure 3.1.: (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 3.1b). 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.

3.1.2. 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 3.2. 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

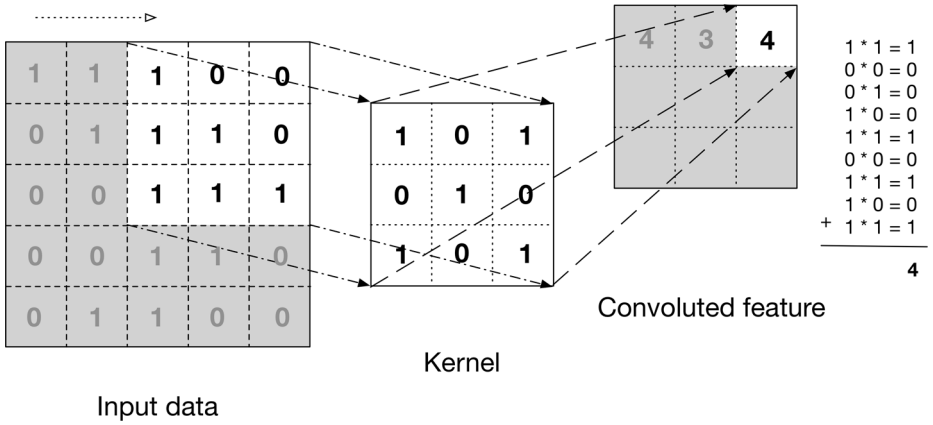


Figure 3.2.: Applying a 3x3 kernel on a 5x5 input. This results in an 3x3 convoluted feature [3].

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 3.3. 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 3.4). 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 3.3).

3.2. Uncertainty in Deep Neural Networks

Deep learning models can learn observed data to perform different kinds of tasks. They predict well and with high confidence given data inside or similar to the training data. However, given an input far away from the training data, the model will predict an output with relatively high confidence. However, the confidence

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

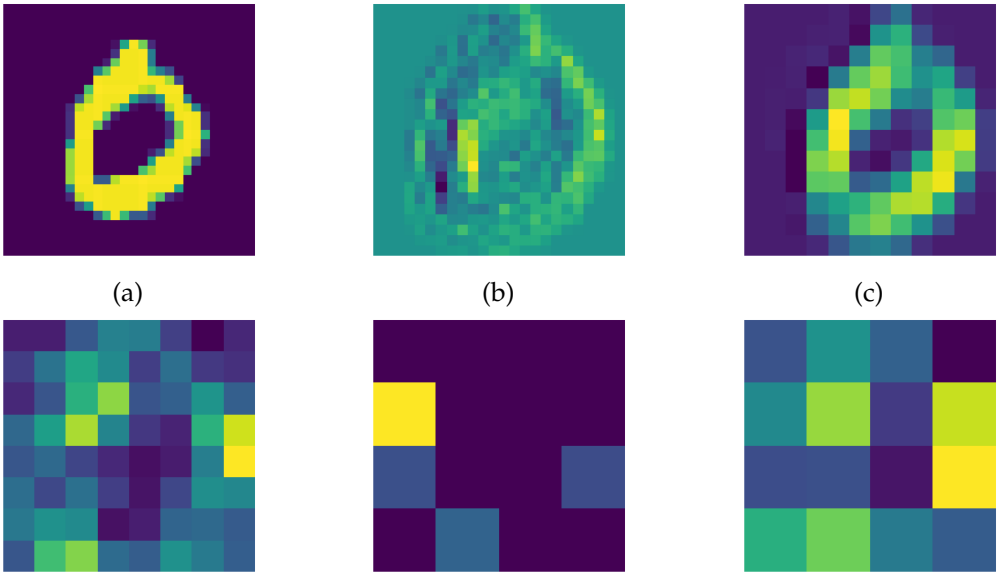


Figure 3.3.: A CNN was trained on the MNIST dataset. The intermediate activations for a specific filter were visualized for every layer. (a) The input image. (b) The featuremap returned by the first ConvLayer. (c) The featuremap returned by the second ConvLayer. (d) The featuremap retuned by the third ConvLayer. (e) The featuremap returned by the fourth ConvLayer. (f) The featuremap returned by the linear layer.

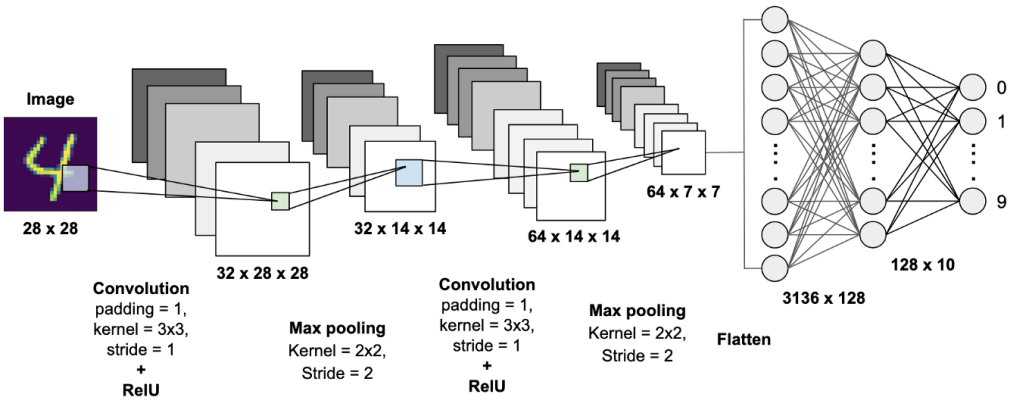


Figure 3.4.: CNN architecture with two ConvLayers, Maxpooling and two linear layers [4].

for out-of-distribution inputs should be small. This misconception entails severe problems in safety-critical applications, such as autonomous driving or cancer detection. For such applications, it is usual to get an out-of-distribution input. An example for such input could be when a self-driving car enters a new environment. Thus, it is desirable to receive a measure of uncertainty besides the prediction. Probabilistic models include properties to return the desired measure of uncertainty besides a prediction. To get an intuition on how uncertainty can be integrated into such models, a closer look at the background of uncertainty is required. We distinguish between two kinds of uncertainty, epistemic and aleatoric uncertainty. In the context of deep learning they can be described as follows:

- *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.

Both types of uncertainty are present when training a deep learning model. Combined they make it possible to form the predictive uncertainty. To see how uncertainty is integrated into a deep learning model we will take a closer look on the probabilistic equivalent for a Neural Network, the Bayesian Neural Network.

3.2.1. 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.

3.3. 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

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. Bayesian Inference

Bayesian inference is a method that uses the evidence to update the beliefs about a specific event. It is Bayes' theorem that determines how we should update the belief based on the evidence.

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

where events A and B have prior probabilities $p(A)$ and $p(B)$. Since exact probability values for A and B are not always appropriate, they are often better represented as probability distributions. So it is when dealing with (deep learning) models. Though, Bayes theorem has a different notation for models.

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

The information we are interested in is the models' parameter-space, which is denoted as θ. The evidence is represented by datapoints $D = \{x_1, x_2, ..., x_n\}$, which will be observed by the model. When performing inference, the prior distribution $p(\theta)$ is refined on the observed data D. This refined distribution is called posterior distribution $p(\theta|D)$. Comparing the posterior distribution to the prior belief $p(\theta)$ will tell us what the model learned from observing the data. The most probable value in the posterior is called the maximum a posteriori (*MAP*). However, this value will not have a probability of 100%. In other words, it tells us how certain or rather uncertain the model is about the outcome.

4.2. The Intractable Posterior

We learned that the posterior distribution gives us a measure of uncertainty. Since a (Bayesian) Neural Network is a parameterized function with parameters θ, a posterior distribution could be obtained with Bayes theorem. Nevertheless, in most cases, the posterior distribution becomes intractable. To compute the posterior $p(\theta|D)$ we need to know the likelihood $p(D|\theta)$, the prior $p(\theta)$, and the evidence $p(D)$. The former two terms are easy to compute since they are either dependent on theta or already known, respectively. It is the evidence $p(D)$ that makes the computation infeasible. The evidence normalizes the posterior to add up to 1. In other words, the evidence needs to be computed such that:

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

This integral is computable for low dimensions of θ. It is practically intractable for higher dimensions because the number of possible values for θ grows exponentially with its dimension. When working with (Bayesian) Neural Networks, the dimension of θ is usually very large; thus, the posterior needs to be approximated.
There are many methods to approximate the posterior distribution. Some of them were already mentioned in chapter 2. The approximating methods used in this thesis will be presented hereinafter.

4.3. 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:

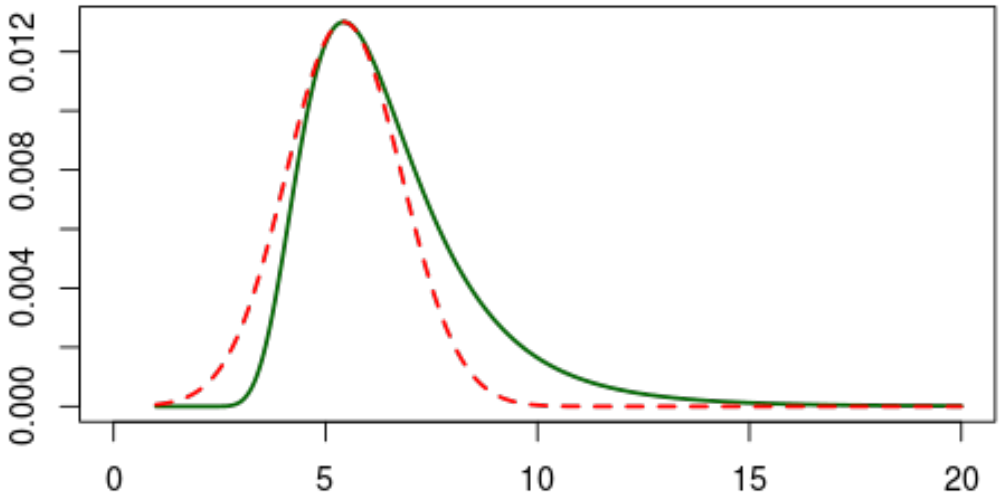


Figure 4.1.: Example of a Laplace approximation. The solid green line shows the posterior distribution that needs to be approximated. The red dotted line shows the approximated posterior, which is a Gaussian centered at the mode of the true posterior. [1].

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

, 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 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 $\mathbf{H}$:

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

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 $\mathbf{H}$. We use the curvature as a measure of uncertainty, since the magnitude of the values in $\mathbf{H}$ 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/

4.4. 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.

4.5. 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]

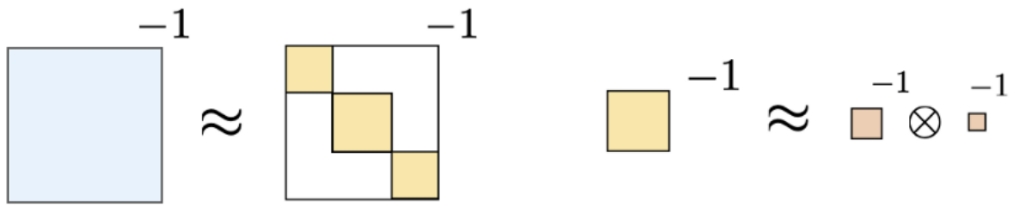


Figure 4.2.: Visualization of KFAC. The Hessian (blue) gets approximated by block-diagonals. Each of the diagonal blocks is approximated by a krockecker product of two smaller matrices. Inverting the Hessian can be achieved by inverting the kronecker products [3].

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:

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

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

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

where $\otimes$ denotes the Kronecker-product.
Since the layerwise Hessian is split into two Kronecker products, both of which should be described separately. $[Q_\lambda]$ and $[Q_\lambda]$ 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:

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

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

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.

[GG16] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2016.

[LPB17] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6402–6413. Curran Associates, Inc., 2017.

[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.