2. Background

This chapter sensibly introduces you to the background and underlying concepts of this work. Even though not all sections go into detail, they contain adequate information to understand the related contents.
The chapter will start with a survey of Neural Networks (NNs). The central network class in this work is the Convolutional Neural Network. Therefore, a general description of CNNs is part of this survey. After that, the need for uncertainty in deep learning is reasoned. To round off the concepts of uncertainty in deep learning, a general framework for uncertainty in NNs is introduced: the Bayesian Neural Network. It follows an introduction of Bayesian modeling to explain the mathematical basis and the need for approximating methods in Bayesian deep learning.Lastly, the Python package BackPACK for PyTorch is presented, as this work uses it to implement stated concepts and receive information about uncertainty.

2.1. From Perceptron to Deep Neural Network

Computers showed fascinating performance in the calculation with vast amounts of data. Nevertheless, they only follow the instructions of humans or a program written by humans. Inspired by the brain, Neural Networks give computers the ability to reason like humans. As of today, NNs outperform humans in specific tasks, like image recognition or video games. This was made possible by modeling neurons mathematically. The first Neural Network, as we know it today, was Rosenblatt's perceptron [Ros58]. From this basis, a variety of specialized NN classes originated, such as Convolutional Neural Networks and Recurrent Neural Networks. Given some data D = X, Y, NNs can learn to generate Y from inputs X. But it is hard to get an intuition of how the network generates the data. Depending on the complexity of the parameter space, there are plenty of possible functions to generate D. Due to this property, NNs are often referred to as black boxes.
This section will introduces Neural Networks, aimed to understand the fundamentals. We are beginning at the simplest NN, the perceptron.

2.1.1. The Perceptron

The perceptron was the first type of artificial neuron, and its basic design is still present in modern NNs. It is a classifier for linear separable tasks in the setting of

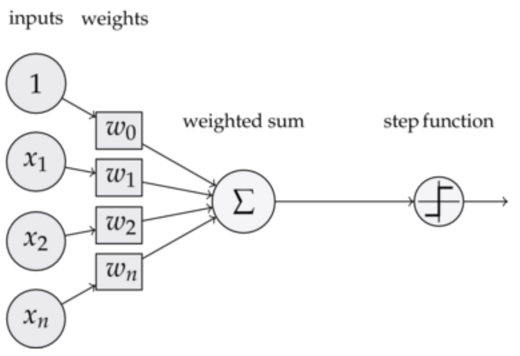
supervised learning. As the perceptron can learn every function it can represent, it was pioneering for deep learning.
Inspired by a neuron in the human brain, the perceptron takes inputs and fires an output. Simple mathematics can model this behavior. For that an input $X = \{x_1, ..., x_i\}$ is converted into a binary output y_i. Every input is weighted and therefore quantifies the importance of input x_i for output y_i, w.r.t. the data. The weighted sum of all inputs forms an intermediate output v_i. This aggregate v is given into a binary step function ϕ, which is called activation function and returns the final output o (Figure 2.1a)

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

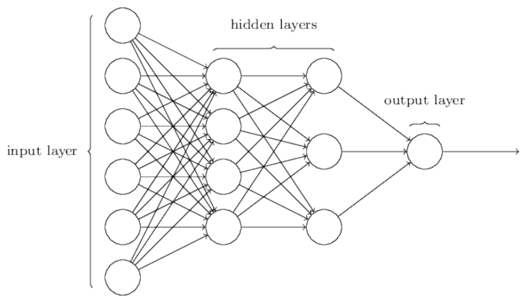
One of the inputs, the bias b, has a constant value of 1. The bias helps the perceptron to control the activation function, providing more flexibility, and generalizing the input data.
As mentioned, the single perceptron can classify linear separable datasets. Combining plural perceptrons extends the number of tasks it can perform. A multilayer perceptron (MLP) arranges several perceptrons layerwise to form a feed-forward-network (Figure 2.1b). It consists of three layers: the input layer, the hidden layer, and 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) as we know it today. Nevertheless, in today's various 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. Over the last decade, CNNs showed impressive performance in the realm of image-classification. The approach builds on the application of filters to the input image. Since every pixel in an image is of different importance for a specific task, filtering is a suitable concept. 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 the value 1 throughout every entry of a single row and the value 0 for every other entry. The filters are applied by traversing over every possible m x n pixels segment of a given image. Every color value in a segment is multiplied element-wise with the filter matrix. The element-wise products are summed up and return a single value for each segment. Figure 2.2 visualizes the application-process of a kernel. There are also two special filters commonly used in



(a)



(b)

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

CNNS, namely average pooling and max pooling. They return the mean value or the maximal value of a segment when applied. The product of a thoroughly filtered image is called a convoluted feature.
The architecture of CNNs always combines three types of layers: the characteristic ConvLayers, the pooling layers (e.g., maxpool, avgpool), and the dense or fully connected layers known from feed-forward-networks. A CNN starts with a sequence of ConvLayers and is followed by some fully connected layers. The pooling layers are located after specific ConvLayer to reduce the input's spatial size and the number of parameters. Each ConvLayer applies a defined number of filters. For example, a ConvLayer of size (1, 32) returns 32 convoluted features of one input image. The outputs produced by a ConvLayer feed into the following ConvLayer. Like so the ConvLayer-sequence filters down an input image to its relevant features for classification (Figure 2.3).
Like every NN, CNNs need training to generate a given dataset. The training procedure adjusts the filters and weights to minimize the expected loss via back-propagation. Thus, training refines the filters to determine the best accuracy. So it happens that the initial filters evolve to detect complicated structures or specialize in complementing another filter. In the example of handwritten digits, trained filters learned to detect whole digits, as visualized in Figure 2.3.

[2]Figure adopted from https://blog.dbrgn.ch/2013/3/26/perceptrons-in-python/
[2]Figure adopted from https://github.com/rcassani/mlp-example
[3]Figure adopted from [PG17]
[4]Figure adopted from https://towardsdatascience.com/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9

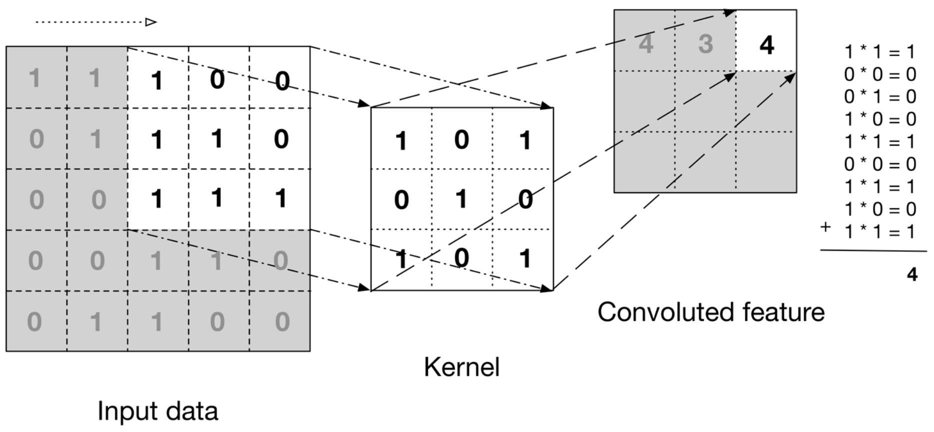


Figure 2.2.: Applying a 3x3 kernel on a 5x5 input. This results in an 3x3 convoluted feature [3].

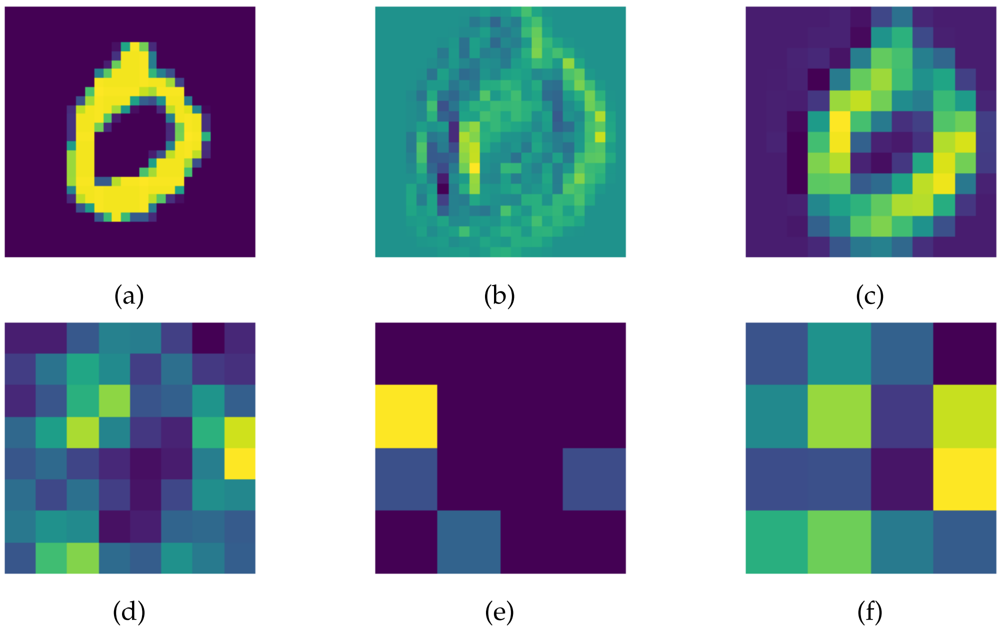


Figure 2.3.: A CNN was trained on the MNIST dataset. The intermediate activations for a specific filter were visualized for every layer. (a) The input image. (b) The feature map returned by the first ConvLayer. (c) The feature map returned by the second ConvLayer. (d) The feature map returned by the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

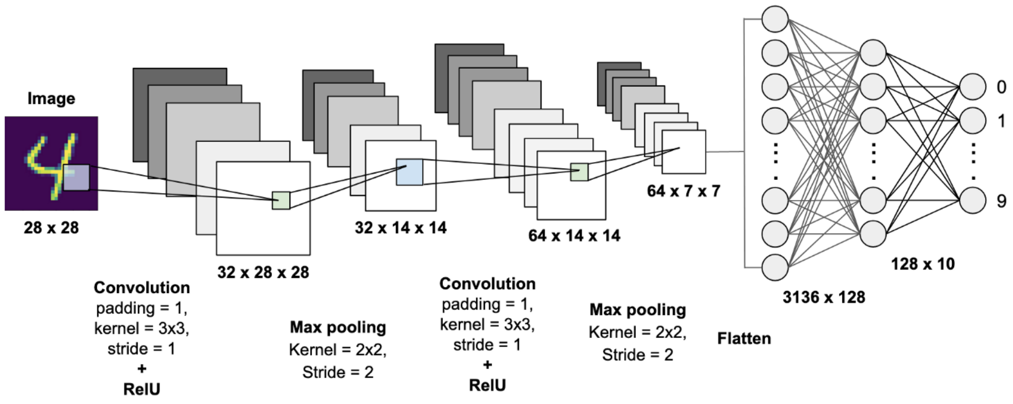


Figure 2.4.: CNN architecture with two ConvLayers, Maxpooling and two linear layers [4].

2.2. Uncertainty in Deep Neural Networks

NNs can learn observed data to perform different kinds of tasks, such as regression or classification. 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 inputs far away from the training data. But in fact, NNs tend to be overconfident for out-of-distribution inputs. At the same time, NNs can easily be fooled by an input. An added perturbation leads to misclassification of an image. Such misconceptions entail 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 makes NNs more reliable for real-world application. Supplementing machine learning, i.e., deep learning with Bayesian statistics incorporates the uncertainty mentioned above into a DNN.

2.3. Bayesian Neural Networks

DNNs lack the ability to form a posterior distribution. This is due to the use of exact values in the weight space of a network. The Bayesian Neural Network (BNN) tackles this problem. Though, BNNs are not specifically a new network class but describe a Bayesian framework for any kind of DNN [Mac92]. This framework comprises a set of tools to transform a DNN into a probabilistic model by inferring distributions over a deep learning model's weight space. Given some training data, the prior distributions over the weights are adjusted to attain the most probable set of weights. Further, a posterior distribution can be obtained to get uncertainty estimates from a prediction. To gain intuition, a BNN can be interpreted as an ensemble of all possible models with given architecture after observing dataset D with a joint output.

2.4. Bayesian Modeling

2.4.1. Bayesian Statisitcs

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 an 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 :

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

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.

2.4.2. 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.

2.4.3. 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.

Approximate Inference

Approximate inference is an essential field of today's machine learning. It makes reasoning about engaging and real-world data for many applications possible and, more importantly, assessable. From a high-level perspective, approximate inference solves Bayes' theorem for complex cases. Chapter 3 describes some approaches for approximate inference. Additionally, this work mainly focuses on one approach: The Laplace approximation, covered in chapter 4.

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. Though, the practical use of Bayesian inference bears some difficulties, most notably the intractability of the posterior distribution. 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

Variational inference (VI) describes a method for approximate inference. A family of distributions $q(w)$ parameterized with θ is defined to approximate the posterior distribution $p(w|D)$. 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:

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

As this objective is dependent on the intractable posterior, the minimization is not possible in this form. But the KL has some properties that help to approximate $p(w|D)$:

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

Decomposing the log evidence reveals the equivalence of minimizing the KL and maximizing the ELBO. The non-negative KL and the ELBO add up to the log evidence. Therefore, the KL is minimal when the ELBO is maximal. Maximizing the ELBO is computationally possible, as it only depends on q. Stated insights reformulate the above minimization problem to:

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

3.2. Bayes by Dropout

Dropout is a technique to improve the training of NNs, which was first introduced by [SHK+14]. Excluding random neurons during each iteration of 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. Theoretically, training with dropout optimizes a number 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.
[GG16] proposed to use dropout during test time for Bayesian approximation. Doing this, we get different outputs for a series of identical inputs. For example, given the input $X = [3, 3, 3, 3]$ a possible output could be $O = [4.2, 4.0, 3.8, 4.1]$. Dropout during testing gives the possibility to form a predictive distribution from the outputs, as we can compare them to a set of MC-samples. Such samples give the possibility to form a predictive distribution. 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 of this approach is to have several networks trained on the same task. In such a way, one network's strengths and weaknesses complement each other. 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 member's 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. Therefore, we get a precise measure of uncertainty from an ensemble of networks.

4. Theory

This work contains several experiments with focus on uncertainty measures. The essential theory to obtain those measures will be presented in this chapter.
In section 4.1, we will begin with the workhorse of this work, the Laplace approximation. After that is the Hessian will be introduced in section 4.2, as it is a crucial part of the Laplace approximation. Lastly, section 4.3. will explain the alternative way to receive information about the uncertainty: 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]. Laplace approximation places a Gaussian at the mode of the posterior distribution (Figure 4.1). The curvature of this Gaussian is found in 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 is 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 expected loss with backpropagation:

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

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

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

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

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

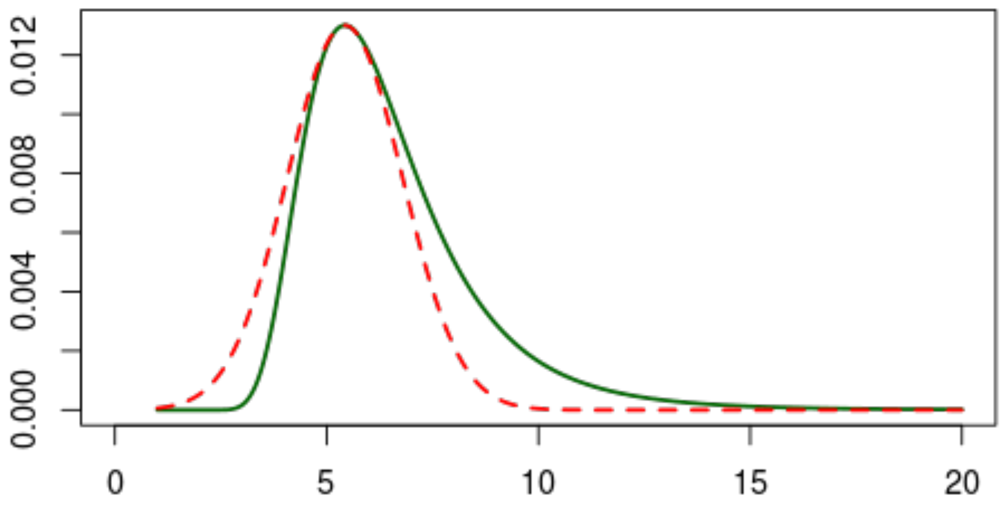


Figure 4.1.: Example of a Laplace approximation. The solid green line shows the posterior distribution that needs to be approximated. The red dotted line shows the approximated posterior, which is a Gaussian centered at the mode of the true posterior. [1].

, 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}. As the derived equation is of Gaussian functional forum, we approximated the posterior as:

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

where $\mathcal{N}$ is the normal distribution. In this thesis we don't focus on the approximated posterior distribution itself, but on the measure of uncertainty it provides. This measure of uncertainty is the curvature we approximated. Therefore we use the inverse of the Hessian as a proxy for uncertainty.

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

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

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 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:

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

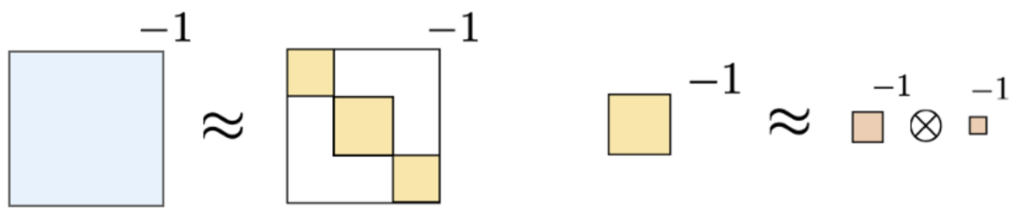


Figure 4.2.: visualization of KFAC. The Hessian (blue) gets approximated by block-diagonals. Each of the diagonal blocks is approximated by a krockecker product of two smaller matrices. Inverting the Hessian can be achieved by inverting the kronecker products [3].

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

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

$$[\mathcal{H}_\lambda]_{a,b} = \frac{\delta^2 E}{\delta h_a^\lambda \delta h_b^\lambda} \,,$$

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

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

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

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

where $\otimes$ denotes the Kronecker-product.
Since the layerwise Hessian is split into two Kronecker products, both of which should be described separately. $[\mathcal{Q}_\lambda]$ 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

– no figures yet –

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.

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.

[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.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

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