LU TP 20-07
January 2020

Training Bayesian Neural Networks

A study of improvements to training algorithms

Johan W. Book

Department of Astronomy and Theoretical Physics, Lund University
Desupervised ApS, Copenhagen

Master thesis supervised by Michael Green
Co-supervised by Mattias Ohlsson

LUND
UNIVERSITY

Abstract

Although deep learning has made advances in a plethora of fields, ranging from financial analysis to image classification, it has some shortcomings for cases of limited data and complex models. In these cases the networks tend to be overconfident in their prediction even when erroneous - something that exposes its applications to risk. One way to incorporate an uncertainty measure is to let the network weights be described by probability distributions rather than point estimates. These networks, known as Bayesian neural networks, can be trained using a method called variational inference, allowing one to utilize standard optimization tools, such as SGD, Adam and learning rate schedules. Although these tools were not developed with Bayesian neural networks in mind, we will show that they behave similarly. We will confirm some best practices for training these networks, such as how the loss should be scaled and evaluated. Moreover, we see that one should avoid using Adam in favor of SGD and AdaBound. Wee see that one should also group the learnable parameters in order to use custom learning rates for the different groups.

Populärvetenskapligt sammanfattning

Genom att använda maskininlärning går det att lösa problem som vi tidigare trodde var olösbara. Saker som ansiktsigenkänning, chatbottar och maskiner som lär sig gå självmant känns idag helt triviala men var en enorm utmaning bara några decennier sedan. Maskinlärning används i alla hörn av vårt samhälle, från medicin och finans till bilindustrin och telekommunikation. Gemensamt för alla dessa sektorer är att man behöver ha full koll på osäkerheten i alla beslut man tar. Ett exempel på detta är en doktor med en patient med cancer. Hur säker doktorn är på att patienten faktiskt har cancer kan avgöra om hen kommer behandlas med cellgifter eller ej.

Artificiella neurala nätverk, vilka är benstommen inom maskininlärning, har idag inget mått på hur säkra de är i de beslut de gör. Ändå används de för t.ex. diagnosticera cancer och i självstyrande bilar. Så vad kan vi göra åt nätverken för att göra dem mer osäkra? En möjlighet är något som heter Bayesianska neurala nätverk. Dessa nätverk är medvetna om hur osäkra de är när de tar beslut. Dessa Bayesianska neurala nätverk är dock svårare att lära upp (att lära någon som är kritisk till allt är tämligen krångligt). Detta arbete undersöker om det finns knep som går att använda for att snabba upp träningsprocessen.

Contents

Chapter 1

Introduction

"The beginning is the most important part of the work.", Plato.

Artificial neural networks [1] are applied in sectors ranging from medicine and finance to automotive and telecommunications. In all these sectors, decisions must have high confidence and well-assessed uncertainties. However, standard artificial neural networks provide no inherent concepts of neither uncertainty nor confidence [2]. When exposed to out-of-domain data they tend to make over-confident and even erroneous predictions (see the example in fig 1.1) [2]. The reason for this is that these networks rely on the principle of maximum likelihood, also known as frequentism. One flaw of frequentism is that it only seeks to maximize the likelihood of the observed data without accounting for its uncertainties. This exposes applications using said networks for the risk of making errors without any estimation of when it is doing so [2].

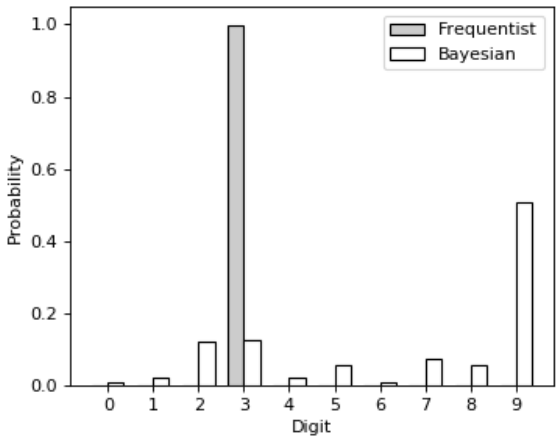

Figure 1.1: A hand-picked example comparison between the classifications of a frequentist and a Bayesian network on Gaussian noise. Both networks have been trained to above 98% accuracy on recognizing handwritten digits.

Another issue of frequentist networks is the proneness to over-training. This requires careful choice of network architecture, fine-tuned hyper-parameters and well selected regularization methods. Not only does this force the user to resort to heuristics, but can lead users to end up with non-optimal, even dubious, networks.

Interestingly enough, the lack of uncertainty is one of the reasons for the tendency of over-training. Hence, if one could implement an inherent uncertainty measure into a network, one would also likely mitigate the effects of over-training [3].

Uncertainties may arise from the data, so called aleatoric uncertainty, and it can rise from the model itself, resulting in epistemic uncertainty. As one often cannot alter the data, the aleatoric uncertainty can be considered fixed. However, models should be changed and updated. This renders epistemic uncertainty a valuable tool for evaluating models.

In order to implement uncertainty into a network, the first step is to look beyond traditional deep learning [4] and into probabilistic machine learning. To highlight the differences this implies, consider a labelled dataset consisting of the inputs x and the labels y. In light of new data points $(\hat{x}, \hat{y})$ the prediction $\mathrm{P}(\hat{y}|\hat{x})$ can be expanded as

$$\mathrm{P}(\hat{y}|\hat{x}) = \int \mathrm{P}(\hat{y}|w, \hat{x})\mathrm{P}(w|y, x)\,\mathrm{d}w \tag{1.0.1}$$

with an intermediate variable w which corresponds to a specific hypothesis. As one must integrate over the whole space of hypothesises, the integral quickly becomes intractable. As an answer to this inconvenient integral, various methods have been developed to approximate it. Until recently those methods were either unscalable or unfeasible computation-wise and one instead adopted a more simplistic approximation for artificial neural networks, namely one chooses a hypothesis as[1]

$$w = \arg\max_{w^*} \mathrm{P}(y|w^*, x) \tag{1.0.2}$$

which gives the following prediction,

$$\mathrm{P}(\hat{y}|\hat{x}) = \mathrm{P}(\hat{y}|w, \hat{x}). \tag{1.0.3}$$

This approximation is based in the belief that the likelihood will always take on the largest possible value, also known as maximum likelihood. Constructing networks based on this principle will produce well performing networks which, however, suffer from the drawback that one cannot estimate the epistemic uncertainty. One can of course build ensembles of frequentist networks and use them to construct epistemic measures [5], but doing so is time extensive - especially for deep networks.

Not being able to estimate the epistemic uncertainty exposes one to the vulnerabilities discussed earlier. The Bayesian approach is an alternative route to solving the integral in order to construct an epistemic measure. In contrast to maximum likelihood, it tries to model the distribution of $\mathrm{P}(w|y, x)$. If one refrains from using point estimates for w, the integral does not vanish. Instead one can utilize Monte Carlo sampling;

$$\mathrm{P}(\hat{y}|\hat{x}) = \frac{1}{M}\sum_{w} \mathrm{P}(\hat{y}|w, \hat{x}) \tag{1.0.4}$$

where w is sampled from the posterior distribution $\mathrm{P}(w|y, x)$. As in the frequentist case, $\mathrm{P}(\hat{y}|w, \hat{x})$ is modelled in the choice of the architecture. But now the weights w are sampled. However, the distribution for w is inherently high-dimensional and

[1] This is a special case of the integral where $\mathrm{P}(w|y, x) = \delta(w - w^*)$ with δ as the Dirac delta function. Then maximum likelihood is invoked to choose the values on the weights w^*. Further, the Dirac delta turns the integral into a single value.

complex which makes it hard to sample from. The two main approaches are Markov chain Monte Carlo (MCMC) and Variational Inference (VI). In MCMC one creates an approximate distribution of $P(w|y,x)$ and samples from it. The sample can be improved to arbitrary accuracy by repeatedly applying random transitions. Although providing a high accuracy, it is a lengthy process and not very suitable for large datasets [6].

Just as MCMC, VI creates an approximate distribution to $P(w|y,x)$ but it uses a loss function (distance between approximation and ground truth) which allows one to use traditional optimization tools. This approach is more suitable for large datasets in shorter computation times [6]. For these reasons, we will use VI to build our networks in this study.

In the variational inference framework, the number of hyper-parameters is greatly increased (typically doubled) while the gains in model complexity is minimal [3]. Further, using conventional training methods, the learning process is slower than its frequentist counterpart and the end result has often worse performance, as seen in fig. 1.2.

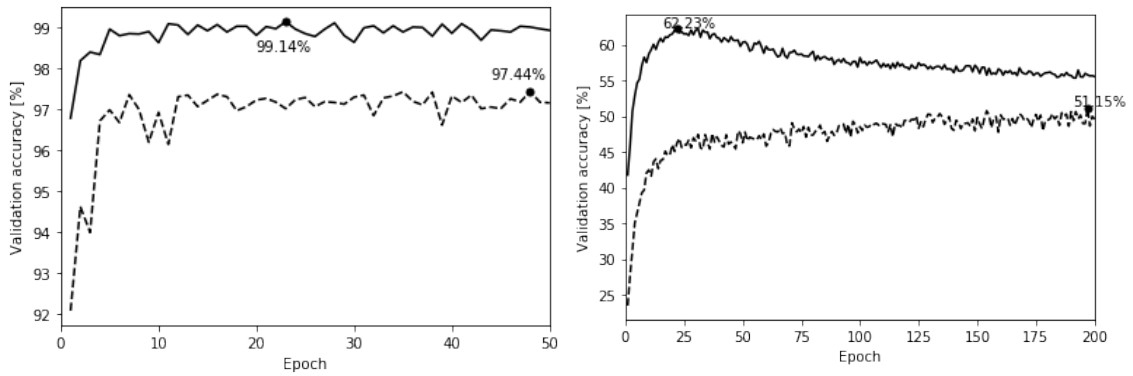

Figure 1.2: Comparison between accuracies of a frequentist (solid) and a Bayesian network (dashed) on MNIST (left) and CIFAR10 (right). Both models have been trained with Adam. Apart from that, no other optimization techniques have been used.

For the Bayesian framework to be an attractive alternative to the frequentist approach, the performance must be on par. When using VI, the optimization tools used are built and adapted for frequentist networks. There is no guarantee that same toolset applies equally well to both frameworks. We will in this study consider how suitable different optimizers are for this task, what learning schedules should be used and how the loss in VI should be treated. From there we derive a set of best practices. As seen in fig 1.2, this investigation is of relevance even for fairly small convolutional networks applied to image classifications, which will be the focus of this study.

Chapter 2

Bayesian neural networks

"The greatest obstacle to true learning is the inability to say "I don't know." , Marty Rubin.

Bayesian neural networks unify artificial neural networks with Bayesian statistics. This gives several advantages over regular frequentist networks: Bayesian neural networks are less confident when exposed to extraneous data, are less prone to overfitting and feature inherent measurements for uncertainty. There are two cornerstones of such networks:

Bayesianism Previous knowledge and beliefs are incorporated into predictions. Instead of starting each network from scratch, we encode available information into it.

Uncertainty Instead of learning point estimates for trainable parameters the networks learn the underlying distributions. This allows one to sample predictions and from there calculate uncertainty in the generated predictions.

This chapter will derive the basic formalism for Bayesian neural networks and what tools should be used for obtaining efficient models. The tools include variational inference, backpropagation and weight initialization. As an introduction to the Bayesian formalism the maximum a posteriori approach will be discussed. It is an alternative to maximum likelihood, which is summarized in appendix B, where one incorporates knowledge about the past and beliefs into the model.

The Maximum A Posteriori approach (MAP) relies on Bayes' theorem,

$$\underbrace{\mathrm{P}(A|X)}_{\text{Posterior}} = \underbrace{\frac{1}{\mathrm{P}(X)}}_{\text{Evidence}} \underbrace{\mathrm{P}(X|A)}_{\text{Likelihood}} \underbrace{\mathrm{P}(A)}_{\text{Prior}} \tag{2.0.1}$$

which can be derived by reformulating the chain rule of probability for two events X and A: $\mathrm{P}(A, X) = \mathrm{P}(A|X)\,\mathrm{P}(X) = \mathrm{P}(X|A)\,\mathrm{P}(A)$. The theorem relates the prior probability $\mathrm{P}(A)$ to the posterior probability $\mathrm{P}(A|X)$ in light of new evidence X. This relation requires knowledge of the evidence probability $\mathrm{P}(X)$ and the event likelihood $\mathrm{P}(X|A)$.

The Maximum a Posteriori approach (MAP) seeks to maximize the posterior $\mathrm{P}(A|X)$ rather than $\mathrm{P}(X|A)$ which is what one does using Maximum Likelihood Estimates (MLE). As an example, consider a network with input x, labels y and

weights w. If we regard the network as a model of the distribution $\mathrm{P}(y|x,w)$, the weights in the two approaches become

$$w_{\mathrm{MLE}} = \arg\max_{w^*} \log \mathrm{P}(y|w^*,x) \quad \text{and} \quad w_{\mathrm{MAP}} = \arg\max_{w^*} \log \mathrm{P}(w^*|y,x). \tag{2.0.2}$$

The weights obtained using MAP can be reformulated using Bayes' theorem. Since $\mathrm{P}(y|x)$ is independent of w^*, the MAP weights become

$$w_{\mathrm{MAP}} = \arg\max_{w^*} \left[\log \mathrm{P}(y|w^*,x) + \log \mathrm{P}(w^*)\right]. \tag{2.0.3}$$

The result in the maximum a posteriori framework is equivalent to the result using maximum likelihood but with an added regularization term. This regularization term is nothing else but the prior of the weights.

So far we have implicitly used point estimates for the weights. But in order to develop a proper epistemic uncertainty measure, we should consider the weights w to be a random variable from a distribution $\mathrm{P}(w|y,x)$. Then we can use the Monte Carlo approximation in eq. (1.0.4). Exactly how we should treat $\mathrm{P}(w|y,x)$ is discussed in the next section. There is an example below to further clarify MAP.

An example Consider an event with a probability p. Using a Bernoulli distribution, p is related to the observations y via $\mathrm{P}(y) = \mathrm{Bernoulli}(y|p)$. Given the observations $y = (0,1)$, the likelihood function becomes

$$\mathrm{P}(y = 0,1|p) = p(1-p). \tag{2.0.4}$$

Using MLE, one obtains a probability of $p = 0.5$. Additionally, consider a related event with a probability that is distributed as $\mathrm{Beta}(3,2)$. Including this in a MAP estimate, gives[1]

$$p = \arg\max_{p^*} p^*(1-p^*) \underbrace{{p^*}^{3-1}(1-p^*)^{2-1}}_{\text{Prior}} \tag{2.0.5}$$

which corresponds to $p = 0.6$ and differs from what was obtained using MLE.

[1] The parameters in $\mathrm{Beta}(a,b)$ can be considered as a successes and b failures. This gives a mean of $a/(a+b)$.

2.1 Variational inference

As discussed in the introduction, the posterior $\mathrm{P}(w|y,x)$ can often not be evaluated exactly. The two main approaches treating it is Markov Chain Monte Carlo (MCMC) and Variational Inference (VI). As mentioned in chapter 1, MCMC is not suitable for complex models with a significant number of parameters and large datasets. For this reason, we will discuss how to build Bayesian neural networks in a VI framework.

The concept of VI is to approximate the posterior $\mathrm{P}(w|y,x)$ with a distribution that is easier to evaluate. This approximation, which is called the variational posterior, is found by minimizing the distance between the true posterior and said approximation. The most common metric for this distance is the Kullback-Leibler divergence, which is defined as

$$\mathrm{KL}[\, q \,|\, p \,] := \int_{-\infty}^{\infty} q \log \frac{q}{p} \mathrm{d}w. \tag{2.1.1}$$

The smaller the KL-divergence, the more similar the two distributions are.

Denote the approximate distribution as $q(w|\theta)$ where θ are the latent variables of the distribution q. Said latent variables should then be chosen as[2]

$$\theta = \arg\min_{\theta} \mathrm{KL}[\, q(w|\theta) \,|\, \mathrm{P}(w|y,x) \,]. \tag{2.1.2}$$

One can here identify the argument to the minimization as the loss of the model. Using Bayes' theorem, the loss can be reformulated as[3]

$$\mathcal{L}(\theta, y, x) = \underbrace{\mathrm{KL}[\, q(w|\theta) \,|\, \mathrm{P}(w) \,]}_{\text{Divergence}} - \underbrace{\mathbb{E}\big[\, \log \mathrm{P}(y|w,x) \big]}_{\text{Log likelihood}}. \tag{2.1.3}$$

The obtained loss can be divided into two parts: the divergence and the negative log likelihood (which we are going to simply refer to as the likelihood). The divergence punishes the model for deviating from the distributions of the priors (which inhibits overfitting) while the likelihood forces the model to learn the features of the data. The likelihood is the same loss that would be optimized in a MLE environment (except that the weights are sampled in this case). The divergence on the other hand is is a new metric without any resemblance from the frequentist case.

With a loss function available, networks can be trained using a traditional optimization algorithm. The most simple of such algorithms is gradient descent. Given the gradient $\nabla_\theta \mathcal{L}$, the model can iteratively be updated as

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t) \tag{2.1.4}$$

for iteration t and learning rate α. This is the same as the frequentist case but instead of updating the weights directly the underlying latent parameters are updated.

[2]The order of q and P do matter here. $\mathrm{KL}[\, q \,|\, \mathrm{P} \,]$ is known as the reverse KL while $\mathrm{KL}[\, \mathrm{P} \,|\, q \,]$ is called forward KL. The reason we use reverse KL is that is more punishing for spreading the distribution and is therefore more suitable for neural networks.

[3]One obtains an additional term $\log \mathrm{P}(y,x)$, which can be ignored since it is an argument minimization and $\mathrm{P}(y,x)$ is independent of θ.

Mean-field approximation

The parameterized distribution $q(w|\theta)$ should be chosen to be fairly flexible and yet simplify calculations. It is popular to factorize the distribution as

$$q(w|\theta) = \prod_i q_i(w_i|\theta_i). \tag{2.1.5}$$

This is a mean-field approximation that avoids interdependencies between variables w_i. This is more efficient for large data and scales better. Keep in mind here that each θ_i still can consist of several variables, but we factor them out into independent distributions.

Stochastic variational inference

In general, gradient descent using the entire dataset in each weight update (so-called "batch optimization") is slow and problematic. For example, there is no guarantee that the full training dataset can be fit into the physical memory of the computer. The established solution is Stochastic Gradient Descent (SGD) where each iteration is based on a small subset of data (a "mini-batch"). However, SGD requires the loss of the full data set to be the sum of the loss of its mini-batches [3];

$$\mathcal{L}(\theta, y, x) = \sum_i \mathcal{L}_i(\theta, y_i, x_i). \tag{2.1.6}$$

This does not hold for the loss in eq. (2.1.3). To allow for mini-batches, it is therefore motivated to scale the divergence for each mini-batch , to get

$$\mathcal{L}_i(\theta, y_i, x_i) = \pi_i \mathrm{KL}[\, q(w|\theta) \,|\, \mathrm{P}(w) \,] - \mathbb{E}\big[\log \mathrm{P}(y_i|w, x_i) \big]. \tag{2.1.7}$$

This is know as KL-scaling or KL-reweighting. This solution is called stochastic variation inference [7], and from now on, this is what we mean by VI.

There are several ways of choosing KL scaling. If we label the scaling of the ith mini-bath as π_i, then an intuitive scaling is $\pi_i = 1/M$ where M is the number of minibatches. However, one could also construct the scalings dynamically, for example as

$$\pi_i = \lambda^i / \sum_{k=1}^{M} \lambda^k. \tag{2.1.8}$$

for some $\lambda \in \mathbb{R}$. This alternative scaling was shown to improve learning in [3] using the value $\lambda = 1/2$. Another interesting remark is that if one sets $\pi_i = 0$ then a frequentist loss is obtained (ignoring the normalization).

2.2 Reparameterizable gradients

In order to update the parameters θ it is preferable to use backpropagation since that gives access to traditional tools for optimization. However, it requires one to be able to evaluate the gradients of the loss function. In order to catch all dependencies one needs to use a total gradient (this is specially important to catch the information in the likelihood term). This is not straightforward, since our weights w are stochastic and a differentiation with respect to a random variable is ill-defined. However, there are some tricks to evaluate the total derivative $\hat{\nabla}_\theta \mathcal{L}$. One can use a score function [8, p. 24–29] or use reparameterized gradients.

Since the expectation value is not dependent on w the term $\nabla_w \mathcal{L}$ will vanish and the total gradient becomes

$$\hat{\nabla}_\theta \mathcal{L} = \nabla_\theta \, \mathbb{E}_{w\sim q} \left[\log \left(\frac{q(w|\theta)}{\mathrm{P}(w|y,x)} \right) \right]. \tag{2.2.1}$$

One can from here obtain[4]

$$\hat{\nabla}_\theta \mathcal{L} = \mathbb{E}_{w\sim q} \left[\log \left(\frac{q(w|\theta)}{\mathrm{P}(w|y,x)} \right) \nabla_\theta \log q(w|\theta) \right]. \tag{2.2.2}$$

In this expression the only derivative that needs to evaluated is $\nabla_\theta \log q(w|\theta)$. This can be identified as a score function from statistics.

In eq. (2.2.2) the gradients do not propagate through the model. This allows using the score function approach for discontinuous distributions, but will yield a high variance [9]. That is why, if possible, one should use something called reparameterizable gradient instead.

Reparameterizable gradients is a method that allows the gradients to explicitly flow through w by rewriting w as a function $t(\theta, \epsilon)$ where ϵ is a random variable from a fixed distribution. For a normal distribution this function is $t(\mu, \sigma, \epsilon) = \mu + \epsilon\sigma$ (where it is understood that $\theta = (\mu, \sigma)$). Using the reparameterization, the expected value can be taken from w to ϵ, as proven in [3], resulting in

$$\hat{\nabla}_\theta \mathcal{L} = \hat{\nabla}_\theta \mathbb{E}_\epsilon \left[\log \frac{q(w|\theta)}{\mathrm{P}(w|y,x)} \right]. \tag{2.2.3}$$

The loss expectation value cannot be calculated directly, due to the unknown posterior, and is therefore approximated by the mean of samples $\mathcal{L}_\epsilon$. The implicit θ-dependence in $w = t(\theta, \epsilon)$ then implies

$$\hat{\nabla}_\theta \mathcal{L}_\epsilon = \nabla_\theta \mathcal{L}_\epsilon + \left[\nabla_\theta \, t(\theta, \epsilon) \right] \nabla_w \mathcal{L}_\epsilon \tag{2.2.4}$$

which can be written in more detail as (2.2.5) which has two terms that are divided into a path derivative and a score function;

$$\hat{\nabla}_\theta \mathcal{L}_\epsilon = \underbrace{\nabla_\theta t \, \nabla_w \big(\log q(w|\theta) - \log \mathrm{P}(w|y,x) \big)}_{\text{Path derivative}} + \underbrace{\nabla_\theta \log q(w|\theta)}_{\text{Score function}}. \tag{2.2.5}$$

[4] Write it in integral form, apply the product rule and realize the term with the differentiated logarithm vanishes. To show that it indeed vanishes, remember that $\nabla_\theta \int q(w|\theta) \, \mathrm{d}w = 0$.

It was shown by G. Roeder et al. [9] that omitting the score function reduces the gradient variance and results in lower loss after converging. They did so using different datasets from image classification with variational autoencoders in the frame of stochastic VI.

Finally, the standard deviation must be non-negative, but that is not part of our formalism. Putting this restriction directly onto a network will impede its performance. This can be resolved by reparameterizing σ as

$$\sigma(\rho) = \log(1 + e^{\rho}). \tag{2.2.6}$$

2.3 Priors

Priors are one of the things that distinguish Bayesian networks from frequentist ones. Having priors corresponds to an encoding of previous expectations. If there is no such information one typically uses normal priors. In regression problems it can also be motivated by being a conjugate prior if the data is believed to feature Gaussian noise.

Initialization

Analogously with frequentist networks, initialization of parameters cannot be done in an arbitrary way if the networks are to be efficient. Unless the initialization scheme conserves variance throughout the network, there is a risk of exploding or vanishing gradients and overall slower rate of convergence. Gloriot et. al [10] proposed considering number of channels[5] going into the layer, n_{in}, and channels going out, n_{out}, in the initialization. One approach to stabilize the variance throughout the network is the Xavier scheme;

$$w_{\text{Xavier}} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{2.3.1}$$

Although the Xavier scheme has shown great success in improving rate of convergence, it does not account for non-symmetric activation functions such as ReLU. One modification by K. He [11] to overcome this drawback is the Kaiming scheme;

$$w_{\text{Kaiming}} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{(1 + a^2)n_{\text{in}}}}\right) \tag{2.3.2}$$

where a is a coefficient depending on what type of ReLU is used (e.g. Leaky ReLU). For standard ReLU this constant is simply $a = 1$.

Considering Bayesian networks, there are typically several parameter groups to be initialized (e.g. means and variances of the weights). In the case of normal priors, the variances are set to the variances given from a Xavier or Kaiming initialization schedule. Then the means can be drawn from a zero-mean Gaussian with associated variance.

[5]Number of channels here is the number of inputs. If the previous layer consists of ten feed-forward nodes, then n_{in} is 10. If instead it is a convolutional layer with 6 output channels each with a dimension $4 \cdot 4$, then n_{in} is $6 \cdot 4 \cdot 4$. The same logic applies to n_{out}.

2.4 An example

This section will summarize this chapter by building a Bayesian version of Rosenblatt's perceptron for the AND problem. On top of the perceptron the model output will be interpreted as input to a modelling distribution to be consistent with the presented formalism. The model will also use ReLU instead of the Heaviside function for activation function (also this to be consistent with earlier formalism).

In the AND problem there is discrete data. Therefore it is suitable to model $\mathrm{P}(y|x)$ using a Bernoulli distribution (as put forward in appendix B). In chapter 1, Monte Carlo sampling was used to estimate $\mathrm{P}(y|x)$;

$$\mathrm{P}(y|x) = \frac{1}{M} \sum_{w \sim q(w|\theta)} \mathrm{P}(y|x, w) \tag{2.4.1}$$

which features the quantities M and q. The constant M is the number of samples. For this toy example only a single sample will be used. The distribution q is the variational posteriors which is required in VI (as shown in section 2.1.1). To simplify matters further, only a single batch will be used.

The neuron can be put together as done in fig. 2.1. It will have an output of $\tilde{y}$ and an activation function $\varphi = \mathrm{ReLU}$. Its bias b, weights w_1 and w_2 are sampled from the variational posterior q.

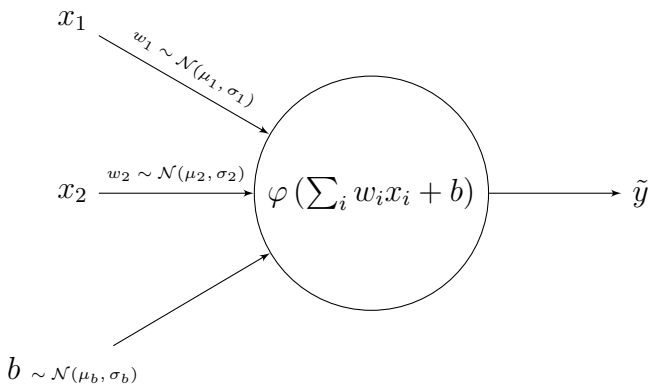

Figure 2.1: A single Bayesian node where its weights and bias are sampled from normal distributions.

In 2.2, reparameterizable gradients was discussed. In the same section, there was also an example with normal distributions. Following that outline, assume that q are Gaussians here also. As there is two weights and a bias one gets

$$w_1 \sim \mathcal{N}(\mu_1, \sigma_1), \qquad w_2 \sim \mathcal{N}(\mu_2, \sigma_2) \quad \text{and} \quad b \sim \mathcal{N}(\mu_b, \sigma_b). \tag{2.4.2}$$

Section 2.3 discussed priors. As normal distributions are used for the variational posteriors, the priors should preferably be within the same family. For simplicity, we set the priors to $\mathcal{N}(0, 1)$ for both the weights and the bias.

The variance of q will be initialize from a Kaiming scheme (eq. (2.3.2) with $n = 2$) which allows initialization of $\mu_1 \sim \mathcal{N}(0, \sigma_1)$. The same procedure can be used to initialize μ_2, σ_2, μ_b and σ_b.

Evaluating the gradients For the sake of clarity, one can evaluate the gradients analytically. For a single update four gradients are required - one for each latent parameter. For example, consider eq. (2.2.5) for μ_1,

$$\hat{\nabla}_{\mu_1}\mathcal{L} = \mathbb{E}_\epsilon\Big[\nabla_{\mu_1} t\, \nabla_{w_1}\big(\log q(w_1|\mu_1,\sigma_1) - \log \mathrm{P}(y|w_1,x) - \log \mathrm{P}(w_1)\big)\Big]$$

$$+ \mathbb{E}_\epsilon\Big[\nabla_{\mu_1} \log q(w_1|\mu_1,\sigma_1)\Big] \tag{2.4.3}$$

In the Gaussian case where $t = \mu_1 + \epsilon\sigma_1$ it follows that $\nabla_{\mu_1} t = 1$. With $\mathrm{P}(y|w_1,x) = \mathrm{Bernoulli}(\tilde{y})$, the loss gradient becomes;

$$\hat{\nabla}_{\mu_1}\mathcal{L} = \mathbb{E}_\epsilon\Big[\nabla_{w_1} \log \mathrm{Bernoulli}(\tilde{y}) + 1\Big] \tag{2.4.4}$$

The Bernoulli term amounts to a one-dimensional cross-entropy loss that can be recognized from the frequentist case, and the prior contributes with a simple 1. The different derivatives of the variational posterior cancel each other.

To actually simulate the model, one needs a probabilistic programming language that includes some form of gradient handling. At at the time of writing this, PyTorch features a capable autograd engine and can to some (minor) extent handle probabilistic programming.

This implementing this setup one can obtain the results in fig. 2.2. Here Adam was used with a learning rate of 10^{-2}. By scaling up the network and fully implementing the concepts discussed earlier, these networks can become on par with frequentist networks.

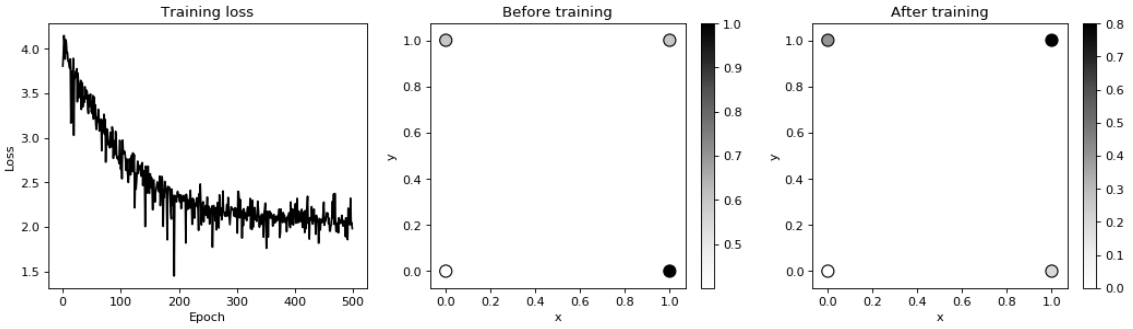

Figure 2.2: Training loss (left) and prediction before and after training (middle and to the right).

Chapter 3

Optimization

"There is always space for improvement, no matter how long you've been in the business.", Oscar De La Hoya.

This chapter will discuss the optimization of the loss function obtained from our networks. It will also review the general structure of optimizers, stochastic gradient descent and how to treat the learning rate for gradient descent. This chapter will also look into robust regions and learning rate schedules for the learning rate. Finally, a more intricate discussion of available optimizers is featured in appendix C.

There are various optimization algorithms used for training artificial neural networks. These algorithms strive to find a minima to some loss function f and are classified as first order if utilizing ∇f or second order if they also employ $\nabla^2 f$. However, second order methods requires calculation of the Hessian, something that with current methods is unviable for any larger network (especially deep networks). Therefore we will only consider first order algorithms.

In general, first order algorithms can be written as done in algorithm 1 (following the formalism outlined by L. Luo et al. [12]) for some functions ϕ, φ and θ. The choice of ϕ and φ determines the properties of the algorithm. Typically, we want ϕ and φ to be approximations for the first and second moment of ∇f since that guarantees all updates to be of the same magnitude. Lastly, the function θ corresponds to a dynamic learning rate that depends on some initial value α.

Algorithm 1 Generic optimizer for a model $\mathcal{L}$ and weights w.

1: **for** $t = 1$ to T **do**
2: $\quad g_t = \nabla \mathcal{L}_t(w_t)$
3: $\quad m_t = \phi_t(g_1, ..., g_t)$ and $v_t = \varphi_t(g_1, ..., g_t)$
4: $\quad \alpha_t = \theta(\alpha, t)$
5: $\quad w_{t+1} = x_t - \alpha_t m_t / \sqrt{v_t}$
6: **end for**

Having considered algorithm 1, one can find ϕ and φ for some of the most common optimizers, as done in 3.1. The used exponential moving average is defined as

$$\langle x_t \rangle_\beta = \beta^t x_0 + (1 - \beta) \sum_{i=1}^{t} \beta^{t-i} x_i \tag{3.0.1}$$

where β is called the smoothing constant.

Table 3.1: The functions ϕ and $\varphi^{-1/2}$ for a few optimizers. How the smoothing constant is labelled and structured may vary between the established versions of the optimizers. The clipping function clips its first value to between its other arguments. The functions η_l and η_u are some given bounding functions.

$\varphi^{-1/2}$ \ ϕ	g_t	$\langle g_t \rangle_{\beta_1}$
$\mathbb{I}$	SGD	Momentum SGD
$\langle g_t^2 \rangle_{\beta_2}^{-1/2}$	RMSProp	Adam
$\text{Clip}\left[\langle g_t^2 \rangle_{\beta_2}^{-1/2}, \eta_l(t), \eta_u(t)\right]$		AdaBound

Algorithms utilizing φ, i.e. the second order estimate, are called adaptive optimizers. These algorithms can be used to be obtain satisfying results but one study shows that they tend to end up in dubious minima for several over-parameterized problems [13].

3.1 Gradient descent

Gradient descent is a form of an Euler step where one traverses the gradient landscape to find a minima. Mathematically, this can be expressed as

$$w_{t+1} = w_t - \alpha \nabla \mathcal{L}_t. \tag{3.1.1}$$

If the $\mathcal{L}$ is a convex function, this is guaranteed to converge to the global optima [14]. However, there is no guarantee that it is a convex function and the loss function function to artifical neural networks tends not to be [15]. Further, gradient descent is also vulnerable to large gradients.

The problem of not escaping local minima can be countered to some extent with momentum, which applies the update rule[1]

$$\begin{cases} m_{t+1} = \mu m_t + \alpha \nabla \mathcal{L}_t \\ w_{t+1} = w_t - m_{t+1} \end{cases} \tag{3.1.2}$$

where α is the learning rate, μ is the momentum and m_0 is set to 0. However, tuning the hyperparameters α and μ requires some thought. Consider the update m_t. The term m_t can be expanded once as

$$m_t = \mu \left(\mu m_{t-2} + \alpha \nabla \mathcal{L}_{t-2}\right) + \alpha \nabla \mathcal{L}_{t-1}, \tag{3.1.3}$$

[1]This calculation is carried out in one dimension, but it is easily extended to an arbitrary number of dimensions.

which can be recursively unfolded to

$$m_t = \mu^t m_0 + \alpha \sum_{i=0}^{t-1} \mu^{t-i} \nabla \mathcal{L}_i. \tag{3.1.4}$$

By studying the expected value of the update in eq. (3.1.5) one can see that it is scaled up by the momentum, something that can be countered by choosing a smaller learning rate.

$$\mathbb{E}[m] = \mathbb{E}[\nabla \mathcal{L}]\, \alpha \left(\frac{1-\mu^t}{1-\mu} \right). \tag{3.1.5}$$

For realistic applications the momentum should $\mu < 1$ which allows one to neglect μ^t for large t. Hence one obtains

$$\mathbb{E}[m] \approx \mathbb{E}[\nabla \mathcal{L}] \left(\frac{\alpha}{1-\mu} \right). \tag{3.1.6}$$

Using too large updates will cause to model to settle for non-optimal minima or even diverge. Assume there is an upper threshold a to $\frac{\alpha}{1-\mu}$ for when the model is stable and able to find minima that generalize well. This results in

$$\alpha + a\mu \leq a, \tag{3.1.7}$$

which is a linear upper bound. This upper bound should be observed when probing the (μ, α)-landscape. Here, a corresponds to a fixed learning rate in an update without momentum. There are algorithms designed to find suitable values on such a learning rate, e.g. YellowFin [16], but none of them have become de facto standard.

A powerful tool to escape local minima is to introduce mini-batches. This results in the enhanced gradient descent algorithm is known as Stochastic Gradient Descent (SGD) [17]. It has shown to function well with noisy gradients (which is common for real-world applications). SGD can be easily combined with momentum, and fine-tuned SGD is the de-facto standard for obtaining state-of-the-art models [16].

3.2 Robust regions

There is an interesting result from classical optimization theory that can serve as a guide in the choice of α and μ. There are regions in the (μ, α)-landscape for gradient descent with optimal rate of convergence [16]. These regions are called robust regions and can in the convex case be found as

$$\frac{1}{h}(1-\sqrt{\mu})^2 \leq \alpha \leq \frac{1}{h}(1+\sqrt{\mu})^2 \tag{3.2.1}$$

where h is the curvature of the loss function. An example of robust regions for different curvatures is shown in fig. 3.1.

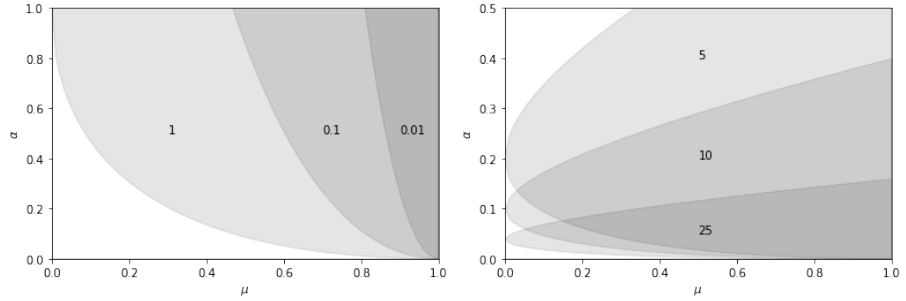

Figure 3.1: Robust regions for different values on the curvature h.

3.3 Learning rate schedules

Considering the robust regions, derived in the previous section, there is little motivation to rely on a constant learning rate. Since the loss landscape changes as the network is trained it is plausible that one should use a non-constant learning rate. Building upon robust regions, there are maximal and minimal viable learning rates for a given network and problem. This concept can be confirmed further by performing a training where the learning rate varies from a very low learning rate to a very high one. This procedure, also known as a learning rate range test, was developed by L. N. Smith [18]. Letting the function θ in algorithm 1 be time-dependent is known as learning rate schedules. Here are a few examples:

Cyclic learning rate A cyclic learning rate is when one updates the learning rate using a periodic function. The frequency of the function might change during training.

Warm-up One can start out with a small learning rate which is increased during training until some threshold have been reached. This is called warm-up and can be used to e.g. counter the initial high variance of Adam [19].

Plateau schedules A plateau schedule is when one uses a metric as guide and when this metric does not improve, one decreases the learning rate. One can also include a so-called patience, by letting the algorithm wait a given number of epochs before updating the learning rate.

Some other example schedule types are shown in fig. 3.2.

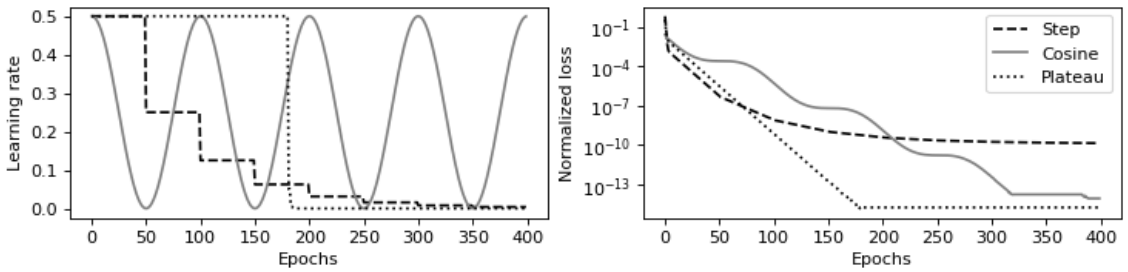

Figure 3.2: Examples of different learning rate schedules for linear regression using SGD.

Chapter 4

Methodology

"March on. Do not tarry. To go forward is to move toward perfection. March on, and fear not the thorns, or the sharp stones on life's path", Khalil Gibran.

This section will discuss the goals of this study, the experimental setup and network architectures that are to be used. In order to see if the results generalize well, we will use various different networks and datasets. Among the architectures, we will study a Bayesian version of LeNet [20].

4.1 Objective

The goal is to learn how Bayesian neural networks train efficiently using VI. There are several apparent differences between training Bayesian networks and frequentist ones. One such difference is the choice of distributions for the priors and variational posteriors - which is out of scope for this thesis. However, other distinct differences are:

1. The loss function has a different shape. One can utilize the path derivative (eq. (2.2.5)) instead of the standard score function and choose several KL scaling methods.

2. The trainable parameters have natural groupings. In the case of normal variational posteriors, the parameters can be divided into means and standard deviations. There are no reason to assume that these groups should be optimized in the same manner.

3. One can sample the gradient. There are methods for subsampling which utilize the mean of the samples but one could also incorporate a variance measure into the optimization process.

Using observations 1 and 2 as a baseline we are going to design and run several different simulations to see how each of these points can be used to an advantage. Later, following this outline, the results will be structured into these two parts.

The goal of the training is to minimize the sum of the divergence and the negative log likelihood (see eq. (2.1.3)). A low negative log likelihood indicates that the network conforms to the data while a low divergence signals that the network conforms to our initial hypothesis. Hence, a high divergence can indicate that either our initial hypothesis is wrong or the network is too driven by data. However,

since divergence is not a clear-cut measure, the most satisfactory networks are not necessarily the ones with the lowest total loss. Therefore, we will try to review what options one has when training these networks and how they affect performance.

4.2 Setup

The aim of the experimental setup is to facilitate reproducibility across different models and different datasets. We also attempt to minimize the impact of hyperparameters that are not of interest to the simulations.

Priors and variational posteriors For priors $\mathrm{P}(w)$ and variational posteriors $q(w|\theta)$ we will use normal distributions. Keeping the variational posteriors in the same family of distributions facilitates VI, and the normal distribution is the standard priors in VI literature.

The initialization process will be done in several steps, which are the same for both convolutional layers as well as conventional feed-forward layers. The priors are set to to have zero mean and standard deviations from a Kaiming scheme. For the variational posteriors the initial means are sampled from uniform Kaiming scheme (which is standard in the framework PyTorch) using the prior standard deviation. The standard deviation of the posteriors is instead initialized to a value selected by hand. Setting this value too high will impair network performance while a too low value results in frequentist networks, rendering all our efforts redundant. We found that a σ as in eq. (2.2.6) of $\ln(1 + e^{-10})$ was a satisfactory trade-off between these constraints (see further discussion in appendix A).

Loss normalization In order for the loss to scale independently of the dataset the loss will be divided by the number of samples in the data set. This should allow losses and learning rates to be more easily comparable between different datasets. Note that it is the full loss that is normalized and this operation will not affect the KL scaling. We apply mini-batches with a size of 10^2, since that works for moderate CPU and memory capabilities.

Datasets We will use two popular datasets from computer vision, namely MNIST [20] and CIFAR10 [21]. MNIST is a labelled dataset of $28 \cdot 28$ pixel monochrome images of handwritten digits. It has 60 000 training images and 10 000 validation images with balanced classes. CIFAR10 is a balanced, labelled dataset as well but contains colored $32 \cdot 32$ pixel images. The images are of airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships and trucks. It consists of 50 000 training images and 10 000 validation images.

The data will be preprocessed as little as possible. The data will be normalized to a mean of 0 and a variance of 1 to equal the playing field for features with different scales. However, no data augmentation will be utilized.

4.3 Model

As the chosen tasks are in the field of computer vision, using convolutional neural networks is suitable. In order to have available benchmarks we will opt for the established LeNet models [20] invented by Y. Lecun et al. These models were created for character recognition. One particular model, known as LeNet-5, obtained an accuracy of 99.05% on MNIST [20]. This is an impressive feat considering the small size of the model. A modern version of LeNet-5 (see fig. 4.1) has obtained 76.23% [22] on CIFAR-10 when utilizing data augmentation, preprocessing and weight decay.

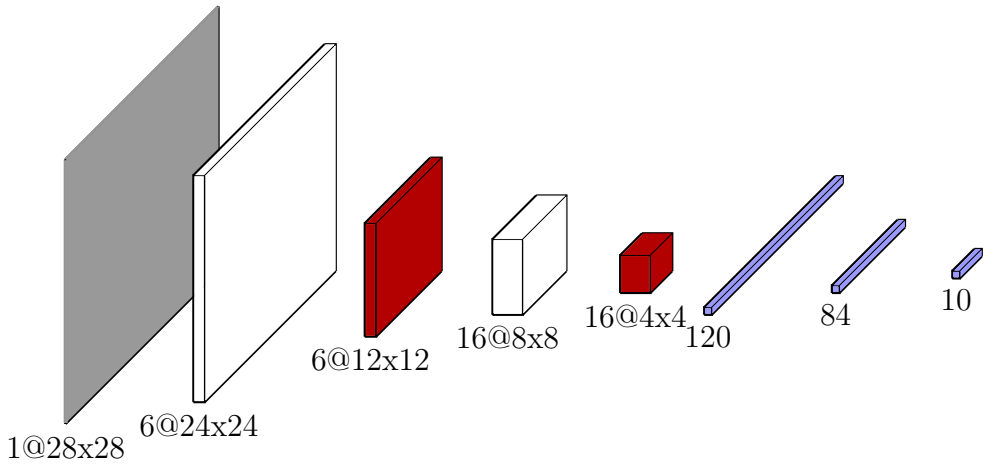

Figure 4.1: The LeNet-5 convolutional network. Here white layers are convolutional layers with ReLU, red are max-pooling and blue are fully connected feed-forward layers. ReLU are used as activation functions.

Typically, the softmax function is used for outputs in mutliclass tasks, as it ensures unitarity and allows for a probability interpretation of the outputs. However, in order to estimate the epistemic uncertainty in a Bayesian network, one needs to be able to draw multiple samples from a single network output. For this reason, we use logistic outputs as parameters in a multinoulli distribution, which we estimate with a single sample Monte Carlo approximation.

We construct a Bayesian version of LeNet-5 using the theory from chapter 2 and section 4.2. We replaced average pooling with max pooling in order to speed up training, and used ReLU[1] as activation function which is standard practice in modern deep learning. This model was capable of obtaining a validation accuracy of 97.74% on MNIST and 59.38% on CIFAR10 with a vanilla training process.

4.4 Optimizers

This study will consider the SGD and Adam optimizers since these two are the most common ones. The optimizers AdaBound, RAdam and YellowFind will also be included, to feature some new non-conventional optimizers. All three of these are designed to battle the problems with learning rates that SGD and Adam have. Optimizers such as Adagrad, Adadelta and RMSProp (see appendix C) are excluded

[1]Using ReLU suits well with the Kaiming scheme we use for initialization.

as we believe those are outdated and limited by their shortcomings. Bellow follows a brief summary of the picked optimizers.

SGD with momentum, as described in section 3.1.

Adam [15] keeps track of first and second order moments using exponential moving averages. For more details, see table 3.1 in chapter 3.

AdaBound [12] is a hybrid between Adam and SGD. Initially it behaves as Adam but as training progresses the individual learning rates are clipped down to match the ones of SGD.

RAdam [19] counters the problem of Adam having high variance in the initial stage of training. By approximating the exponential moving average as a simple average it is capable of rectifying the variance for a better estimate.

YellowFin [16] is an optimizer built around the principle of robust regions (section 3.2). It is SGD but with a scheme to enforce the learning rate and momentum to said robust regions.

All these optimiziers behave well in the frequentist case and their performance for Bayesian neural networks will be studied in the next section.

Chapter 5

Results and discussion

"To acquire knowledge, one must study; but to acquire wisdom, one must observe.", Marilyn vos Savant.

This chapter will investigate and discuss each of the points in section 4.1. It begins by investigating the score function in the VI framework and how the divergence should be weighted. It is followed by how different common optimizers behave and lastly, how one can optimize different parameter groups to obtain better performance.

5.1 Score function gradients

Using only the path derivative leads to decreased variance during training and an overall increase in performance in some cases including Bayesian variational autoencoders using VI on image classification tasks [9]. We are here going to extend this study and consider the impacts of path derivative on LeNet-5.

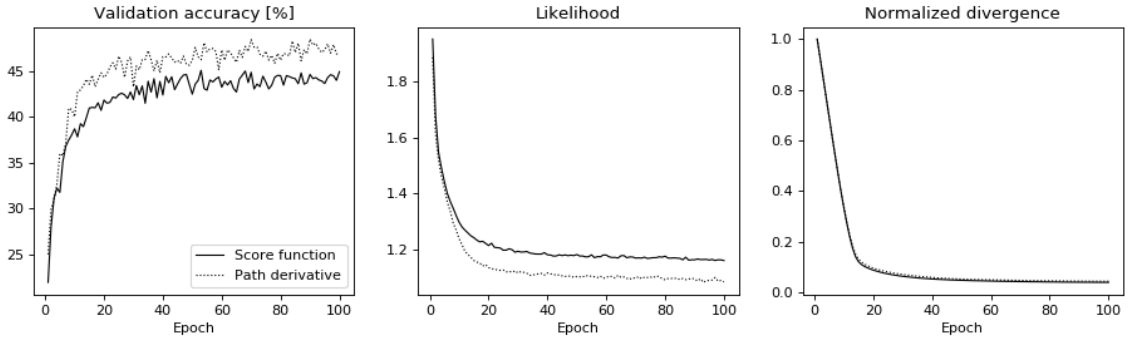

Figure 5.1: A comparison between total derivative including the score function (solid line) and only path derivative (dashed line) using a LeNet-5 on CIFAR10 with Adam with a learning rate of 10^{-3}.

A comparison between including the score function and using only path derivative is shown in fig. 5.1. Omitting the score function does in fact lead to better performance. However, the same simulation was also performed on MNIST and showed no significant difference between the two approaches. Although these are narrow results, it seems that using only the path derivative is at least as suitable, or better, for small Bayesian convolutional networks. In the following all results are obtained with the score function omitted.

5.2 KL re-weighting

According to [3], using mini-batch KL re-weighting of the divergence in eq. (2.1.3) facilitates obtaining better accuracies than using a fixed scaling. Using an alternating KL scale relies on the assumption that parts of the epoch is better off data-driven while others should be divergence-driven. As discussed in section 2.1, one can weight each mini-batch as

$$\pi_i = \lambda^i / \sum_{k=1}^{N} \lambda^k \tag{5.2.1}$$

where i is the index of respective mini-batch. We will study how this performs for a few different values on λ:

0 For $\lambda = 0$ we define $\pi_i = 0$. This ignores the divergence and is equivalent to frequentist training.

1/2 In each mini-batch, training is initially mainly influenced by the divergence. This allows the network to focus on learning the uncertainties in the beginning of each epoch and focusing on the data towards the end. This was proposed in [3].

1 This corresponds to π_i= 1 / number of batches. This is the standard heuristic.

2 We propose the value $\lambda = 2$. This corresponds to the opposite of $\lambda = 1/2$ where each epoch initially focuses on data and divergence towards the end.

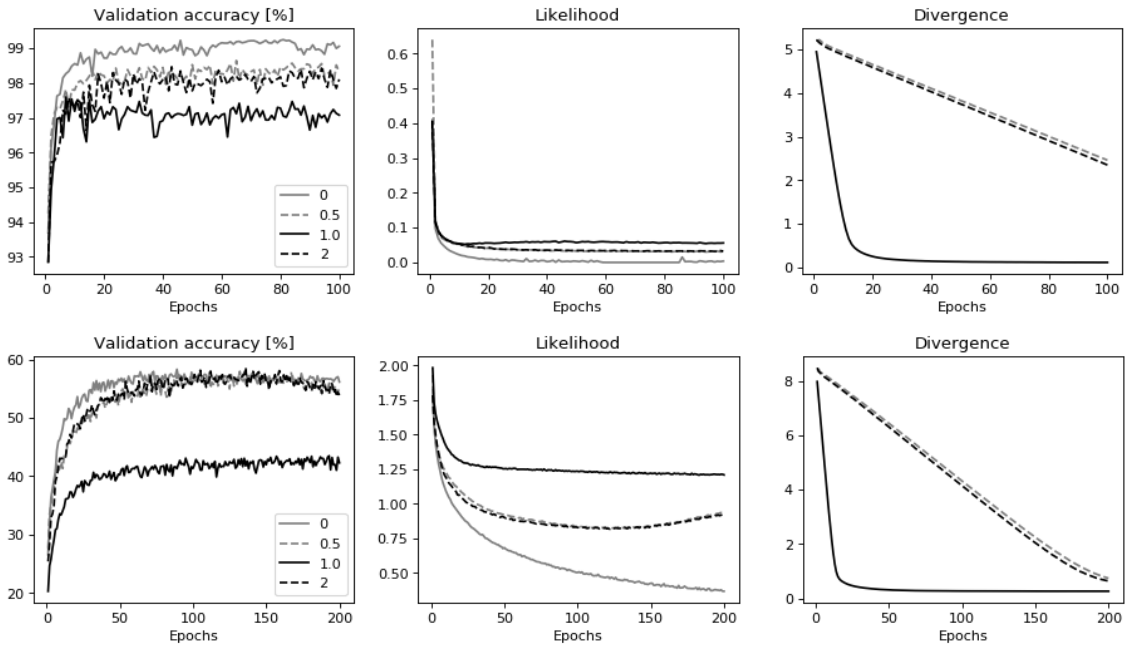

Figure 5.2: Different values on λ for MNIST (top) and CIFAR10 (bottom). The training used Adam with a learning rate of 10^{-3}.

In the simulations shown in fig. 5.2 it is clear that using $\lambda \notin \{0, 1\}$ results in better performance. However, the divergence (and total loss) is worse off. This is because most part of the training is frequentist training. For CIFAR10, which has

500 batches in our setting, only 2% of the batches is assigned a π above $1/500$ (and only 3% has a π above $1/500^2$). Very similar percentages apply to MNIST. This offset is what causes the poor performance in the divergence. It would be more reasonable to use $\lambda \approx 1 \pm 0.03$, which would yield a percentage of 50%. However, doing so results in networks that are worse off in both performance and divergence. Lastly, there are a few batches that are very divergence-driven which hinders the networks from obtaining the same performances as the network using $\lambda = 0$.

The similarity of the performances for $\lambda = 1/2$ and $\lambda = 2$ contradicts the argument that the order of data-driven and divergence-driven learning matters put forward in [3]. This combined with a worse off total loss favors using a constant π.

5.3 Optimizers in a Bayesian setting

To investigate how different optimizers behave in a Bayesian framework, we ran almost 500 simulations for the optimizers in table 5.1, with different hyperparameter settings. To avoid the fallacies of grid search as discussed in [23], the used values was sampled from $\mathcal{N}(v, v/2)$ where v is a value from table 5.2.

Table 5.1: This table describes the optimizers used in the simulation and which hyper-parameters that was altered.

Optimizer	Hyperparameter
Adam	Learning rate, use AMSGrad
AdaBound	Learning rate, final learning rate
RAdam	Learning rate
SGD	Learning rate, momentum
YellowFin	Learning rate, momentum

Table 5.2: The different base values for the hyper-parameters in table 5.1. Values are picked from here using all possible combinations.

Hyperparameter	Values
Learning rate	10^{-3}, 10^{-2}, 10^{-1}
Final learning rate	10^{-2}, 10^{-1}
Momentum	0, 0.1, 0.5, 0.9, 0.95

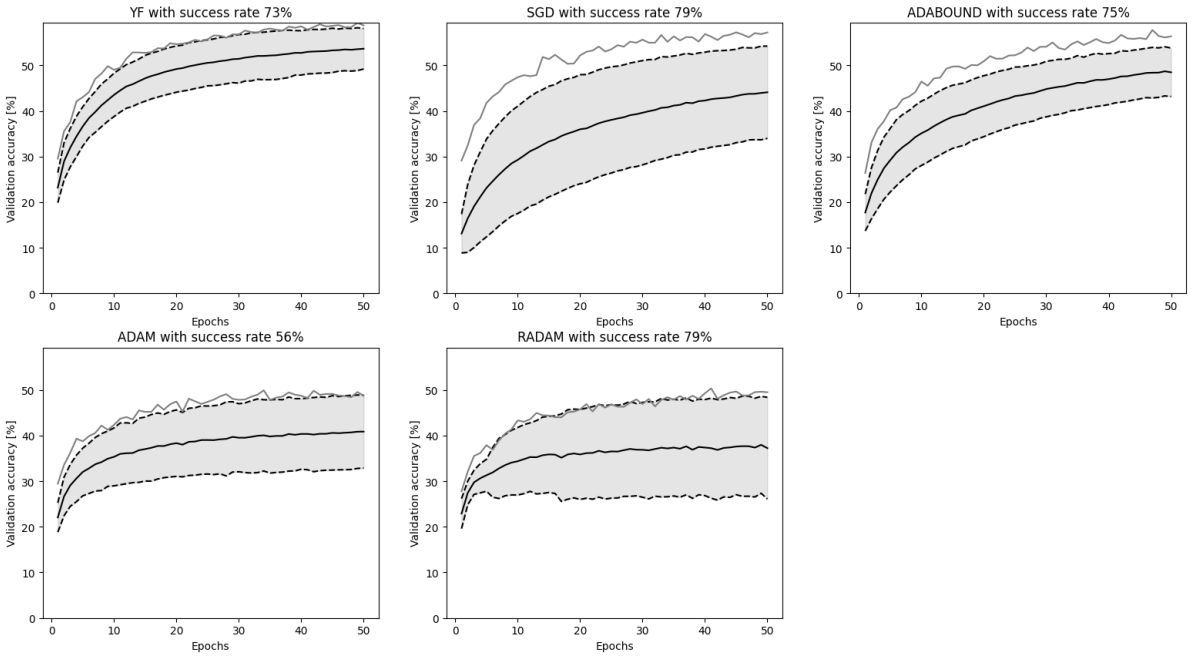

Figure 5.3: Comparison between different optimizers for CIFAR10 with means (solids), mean ± a standard deviation (dashed) and top performer (solid gray line). The statistics are based on 486 runs in total. Here all networks unable to obtain a validation accuracy above 15% have been ignored. The success rate is how many networks succeeded in beating said 15%.

The results from the simulation are presented in fig. 5.3. The top performers (AdaBound, YellowFin and SGD) only rely on first moment information while the weaker performers (Adam and RAdam) also utilize estimates of the second moment. There are two possible explanations to this; adaptive optimization is inherently inefficient or the estimates of the second moment are inadequate. There is a range of evidence supporting the former [12, 24, 25] for late stage training.

The poor results for RAdam are disagreement to observations in non-Bayesian applications [19], which could suggest that RAdam has problems in a Bayesian setting. Studying the individual runs of RAdam, we found several runs that reaches an accuracy of ca 30% and then immediately drops back to 10%. Future investigations are needed to determine if this behavior is caused, e.g., by some numerical instability or by a too large step in σ.

SGD

We performed simulations with 1 epoch in MNIST and 10 epochs on CIFAR10, to study how SGD behaves for different values on learning rate and momentum. Although the training likely has not fully converged in any of these cases, it allows us too look into the behaviour of the optimization algorithm. Training for a single epoch allows more simulations to be done, creating the dense result plot in fig. 5.4, while fig. 5.5 offers less detail but still show similar tendencies. For plot clarity, the hyperparameters were sampled from uniform distributions.

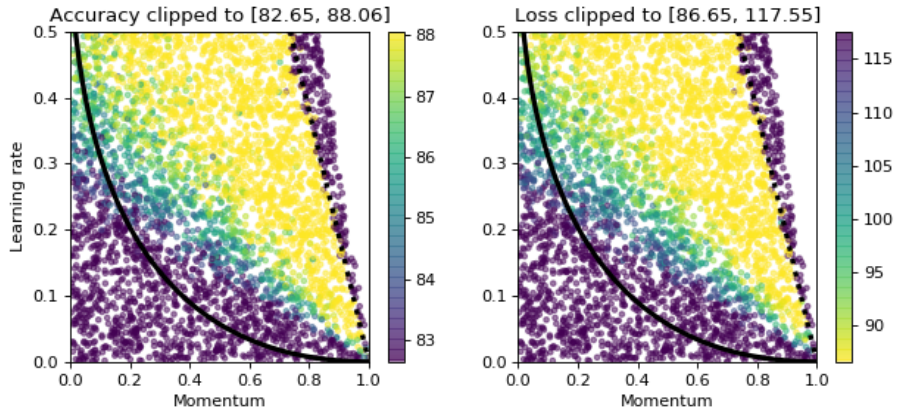

Figure 5.4: Learning rate *vs* momentum landscape for a single epoch on MNIST. Data in the outer 3-quartiles have been clipped and a lower bound for a robust region with curvature 1.5 has been added. The upper line has a coefficient of 1.9.

In both figure 5.4 and 5.5 there are stable, seemingly bounded regions. These do not completely coincide with the robust regions for SGD derived by J. Zhang et.al. [16], although it can be that it converges to this lower bound later in the training process. Further research is still required to investigate methods to determine the curvature of the lower bound and slope of the upper bound.

Another interesting aspect is that there is a "region of death" in the dark area to right of the yellow part. Networks trained here gain low accuracy and very high loss, which is in agreement with the existence of an upper bound on $\alpha + a\mu$ for model stability, discussed in section 3.1. Interestingly enough, this upper bound is different between the MNIST and CIFAR simulations.

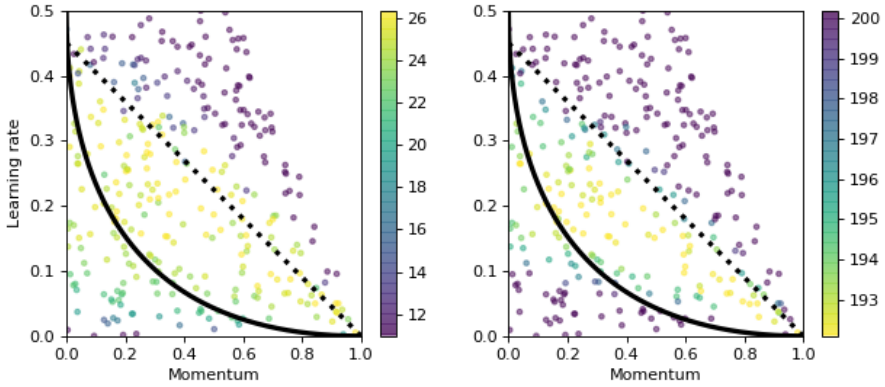

Figure 5.5: Learning rate momentum landscape for 10 epochs on CIFAR. The loss have been clipped to in the lower 2-quantile and a lower bound for a robust region with curvature 2.0. The upper line has a coefficient of 0.45.

The optimal regions in the MNIST and CIFAR simulations have similar shape but the CIFAR one is significantly smaller. This can be for several reasons: different datasets and hence widely different loss landscapes or dependence on number of epochs trained.

To summarize our findings, there is a strong covariance between α and μ and they cannot be tuned independently. There seems to be a clear robust region, motivating optimizers such as YellowFin.

Adam

The optimizer Adam has three parameters of interest[1]: α, β_1 and β_2. The parameter β_1 is more or less equivalent to momentum parameter in SGD. In fact, SGD is a special case of Adam, with $\beta_2 = 0$. In order to see how Adam depends on our betas we will construct landscape plots, similar to the previous section.

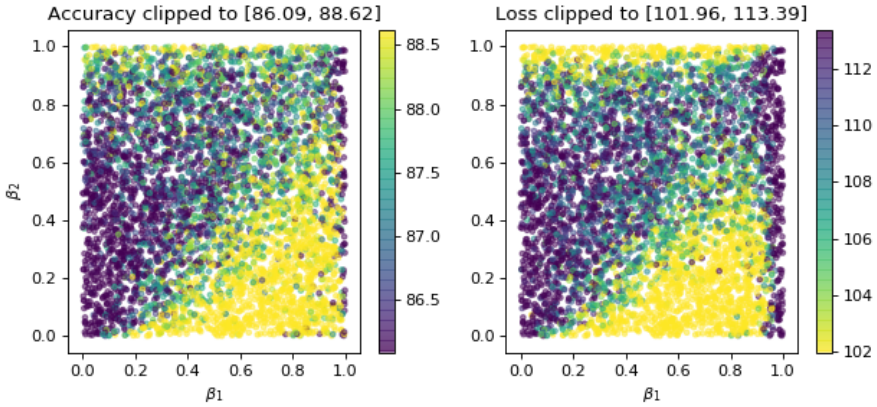

Figure 5.6: Beta landscape for 1 epoch on MNIST. The outermost 10-quantiles have been removed to more clearly convey the division of the different regions.

Figure 5.6 and 5.7 show that that low values for β_2 give better performance, which is in conflict with the $\beta_2 = 0.999$ default value[2] recommended by the authors of Adam [15]. Since a low value for β_2 makes Adam more similar to SGD, the result instead agrees with the claim that fine-tuned SGD outperforms Adam [24].

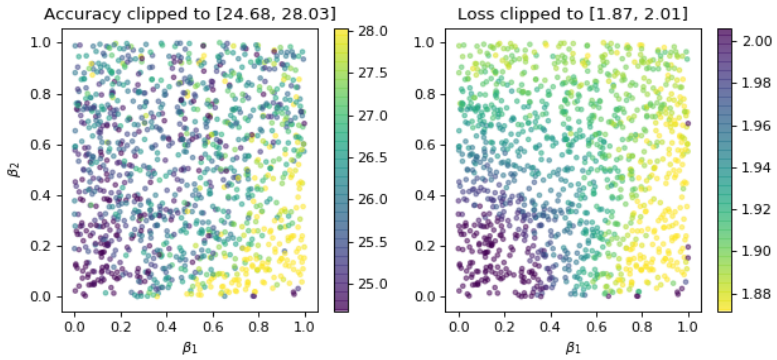

Figure 5.7: Beta landscape for 10 epochs on CIFAR. The outermost 10-quantiles have been removed to more clearly convey the division of the different regions.

[1]There is also a parameter ϵ for numerical stability that was shown to be of interest in [19]. However, we are going to ignore it in this study.

[2]This is default in all major frameworks, such as e.g. PyTorch, Tensorflow and MXNet

5.4 Individual training

In Bayesian frameworks, there are typically several different groups of latent parameters, and there is no reason that these groups should be optimized in identical manners, perhaps not even with the same algorithm. As we use Gaussian distributions, we will here study individual training of means as one group and standard deviations as another.

One straightforward approach is to alternate between optimizing μ and σ for a fixed number of epochs during training. This also gives insights to how training the different parameter groups impacts performance. The results from one such simulation is shown in fig. 5.8. Comparing conventional training and this sequential training results in more or less the same accuracy. However, the likelihood loss and divergence differ greatly. When training μ, the likelihood decreases, while training σ decreases the divergence.

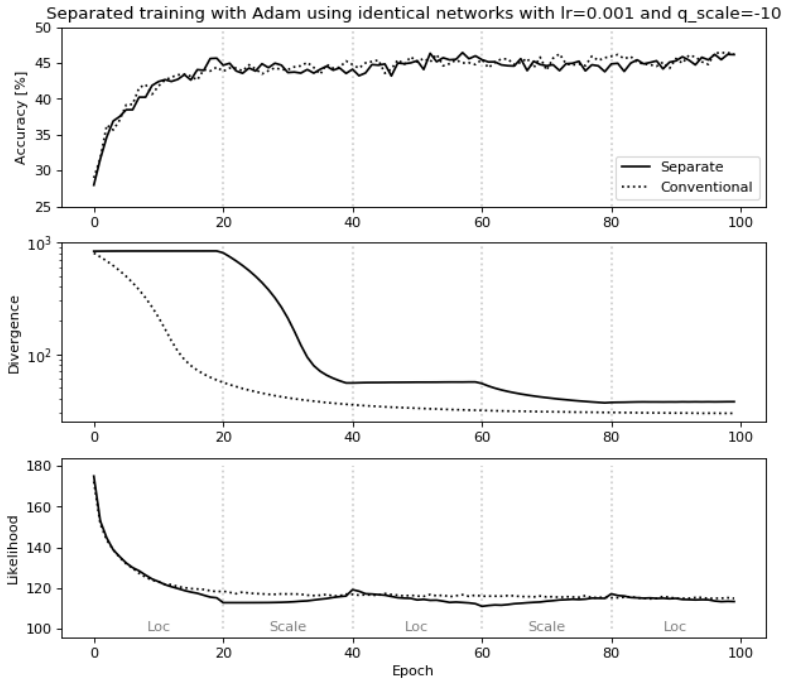

Figure 5.8: Comparison between conventional training and training variable groups separately. The training used LeNet-5 on CIFAR with Adam with a learning rate of 10^{-3}.

One further conclusion we can draw from fig. 5.8 is that the learning rate for μ, here named α_μ, affects the likelihood. Similarly, the learning rate for σ, which we label α_σ, affects the divergence. To look for optimal learning rates, we ran a random search for Adam and AdaBound, with results in fig. 5.9.

AdaBound has regions with optimal values for α_μ, but seems invariant to the specification of α_σ. This is likely due to the fact that the initial learning rate set in AdaBound is less important than the final learning rate (which where fixed to 0.1 in this experiment). Hence, the only requirement on the initial learning rate is to stop the system from becoming unstable (which is what happens at $\alpha_\mu \gtrsim 10^{-1}$). In terms of divergence, there is a clear limit on where α_μ becomes suboptimal.

While AdaBound did not benefit from separating the training groups, Adam did. There are optimal values for both α_μ and α_σ, although the width of the region for α_μ is smaller then for α_σ. In terms of divergence, it seems the larger α_σ, the better.

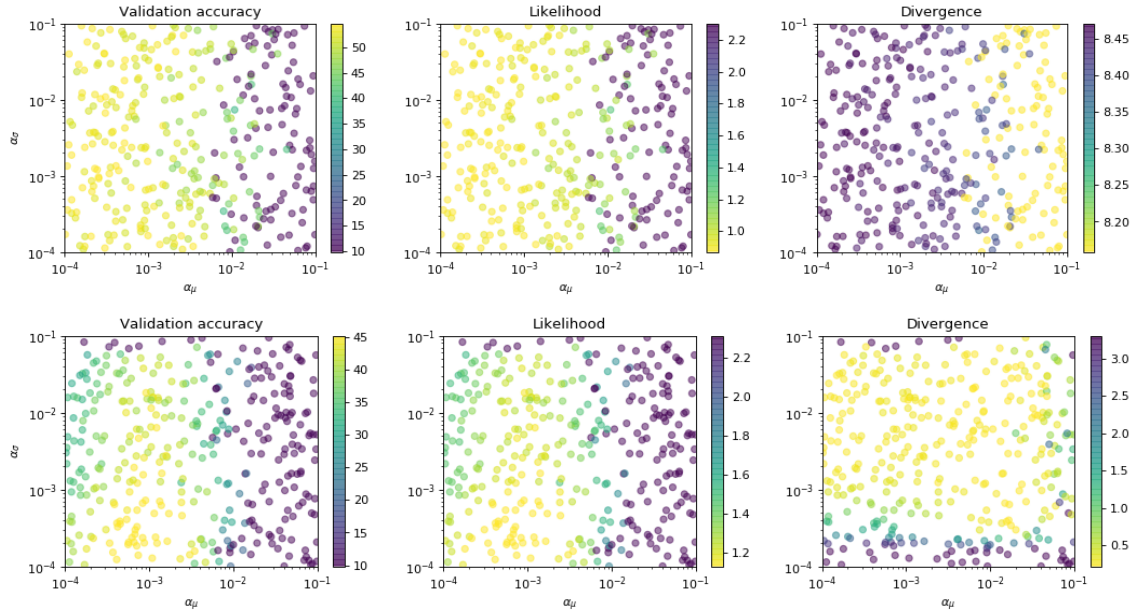

Figure 5.9: Training with different learning rate for means and standard deviations on CIFAR where each network has been trained for 50 epochs. The top plots are for AdaBound and lower plots for Adam.

For Adam, fig. 5.9 suggests that there are optimal regions for both α_μ and α_σ, and that the region for α_μ is more narrow. To investigate their relations in more detail, we adopted a heuristic approach, where we first determined a good α_μ from fig. 5.9, then different values were probed for the ratio

$$r = \frac{\alpha_\sigma}{\alpha_\mu}. \tag{5.4.1}$$

with a fixed $\alpha_\mu = 0.01$. The result in fig. 5.10 shows that α_μ should be at least a factor 10 larger than α_σ.

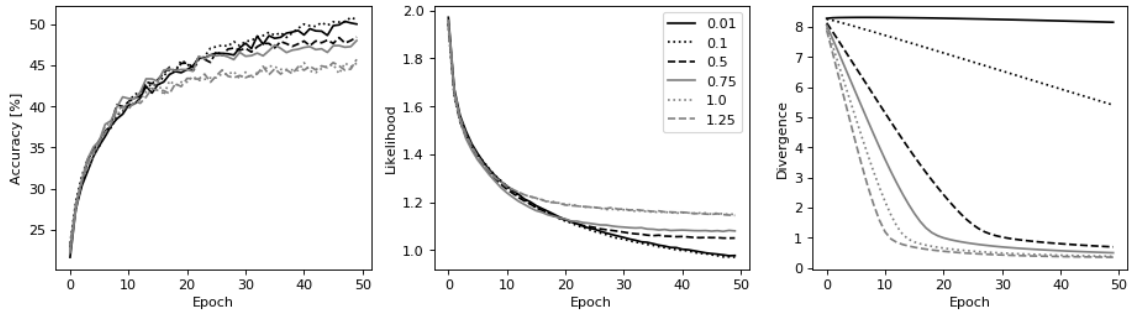

Figure 5.10: Networks trained using different values on r where $\alpha_\mu = 10^{-3}$ using Adam, LeNet-5 and CIFAR10. Each curve is an average over 4 runs.

To automatize the simultaneous search for good values for α_μ and α_σ, we ran a plateau scheduler (as presented in section 3.3). Based on the results in fig. 5.8, we

decided to let the likelihood be the metric for α_μ and the divergence the metric for α_σ. The results with this training is shown for CIFAR in fig. 5.11, where it seems to improve the training process significantly. However, the same plateau scheduler on MNIST made no significant difference.

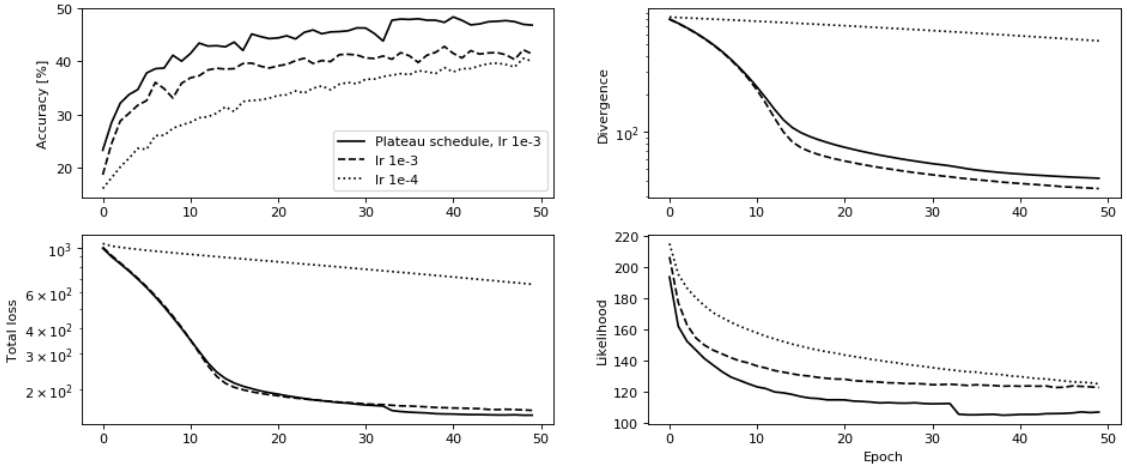

Figure 5.11: Comparison for Adam between conventional training (dashed) and training variable groups separately using a plateau learning rate schedule (solid). The schedule had a patience of 1, a decay factor of 0.1 and an initial learning rate of 10^{-3}. The conventional trainings used the learning rates 10^{-3} and 10^{-4}.

In each simulation, we saw a decreasing divergence. This is in contrast to what you would expect. As μ is learnt, the divergence should increase (as we deviate from the prior). Since we have opted for initialization of rather narrow variational posteriors, this is likely countered by σ increasing.

Chapter 6

Conclusion

"A conclusion is simply the place where you got tired of thinking.", Dan Chaon.

In chapter 5 it was shown that the choice of optimizer, learning rates and divergence weighting matters. Section 5.3 studied optimizers and had the most success with YellowFin and AdaBound. These optimizers had the lowest training variance and quite high robustness to misspecification of hyper-parameters. Among these AdaBound seem to be most robust to misspecifications and the one we recommend. On the other hand, Adam and RAdam had the least satisfactory results with high training variance and low mean performance.

In section 5.4 it was seen that if one uses normal distributions, then one should consider optimizing the means and standard deviations separately. Our applications benefited from a lower learning rate for the standard deviations of the variational posteriors than for the means. Moreover, one can use separate learning rate schedules for the means and standard deviations. We had some success using a plateau schedule where we used the negative log likelihood as metric for the learning rate for the means and the divergence as metric for the standard deviation learning rate.

Lastly, one must weight the divergence properly. The most optimal weight is one over the number of batches (see section 5.2) and one should use path derivative over a score function (section 5.1).

We have to note that this study was done for a relatively small network (LeNet-5) for image classification tasks on MIST and CIFAR10. Hence, these results should apply for these kind of networks. It should also be studied if these results hold for larger architectures and other data domains. We have also used rather narrow variational distributions to obtain satisfying results, however, this is in contradiction with Bayesian philosophy. Hence, it is left to confirm if this work holds for wider distributions.

To conclude, Bayesian neural networks are capable of obtaining performances similar to frequentist networks but it requires a clever choice of optimizer and learning rate schedule. One should also consider using custom learning rates for the different variable groupings.

Bibliography

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

[2] A. Kendall and Y. Gal, "What uncertainties do we need in bayesian deep learning for computer vision?," in *Advances in neural information processing systems*, pp. 5574–5584, 2017.

[3] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural networks," *arXiv preprint arXiv:1505.05424*, 2015.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] T. Pearce, M. Zaki, A. Brintrup, and A. Neel, "Uncertainty in neural networks: Bayesian ensembling," *arXiv preprint arXiv:1810.05546*, 2018.

[6] T. Salimans, D. Kingma, and M. Welling, "Markov chain monte carlo and variational inference: Bridging the gap," in *International Conference on Machine Learning*, pp. 1218–1226, 2015.

[7] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," 2012.

[8] A. Pickles, *An introduction to likelihood analysis*. No. 42, Geo Books, 1985.

[9] G. Roeder, Y. Wu, and D. K. Duvenaud, "Sticking the landing: Simple, lower-variance gradient estimators for variational inference," in *Advances in Neural Information Processing Systems*, pp. 6925–6934, 2017.

[10] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

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

[12] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," *arXiv preprint arXiv:1902.09843*, 2019.

[13] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning," in *Advances in Neural Information Processing Systems*, pp. 4148–4158, 2017.

[14] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[16] J. Zhang and I. Mitliagkas, "Yellowfin and the art of momentum tuning," *arXiv preprint arXiv:1706.03471*, 2017.

[17] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[18] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464–472, IEEE, 2017.

[19] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the Variance of the Adaptive Learning Rate and Beyond," *arXiv e-prints*, p. arXiv:1908.03265, Aug 2019.

[20] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012.

[22] W. Li, "cifar-10-cnn." `https://github.com/BIGBALLON/cifar-10-cnn`, 2019.

[23] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[24] L. Wu, Z. Zhu, *et al.*, "Towards understanding generalization of deep learning: Perspective of loss landscapes," *arXiv preprint arXiv:1706.10239*, 2017.

[25] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," *arXiv preprint arXiv:1902.09843*, 2019.

[26] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[27] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," 2018.

Appendix A

Setup details

Initial σ for q

The choice of the standard deviation σ for creating the initial distributions for q features some trade-offs. Using a too small value will lose the benefits of a Bayesian framework while using a too high value will result in poor performance. To investigate this we run a few runs for different values the scales. The results are presented in fig. A.1.

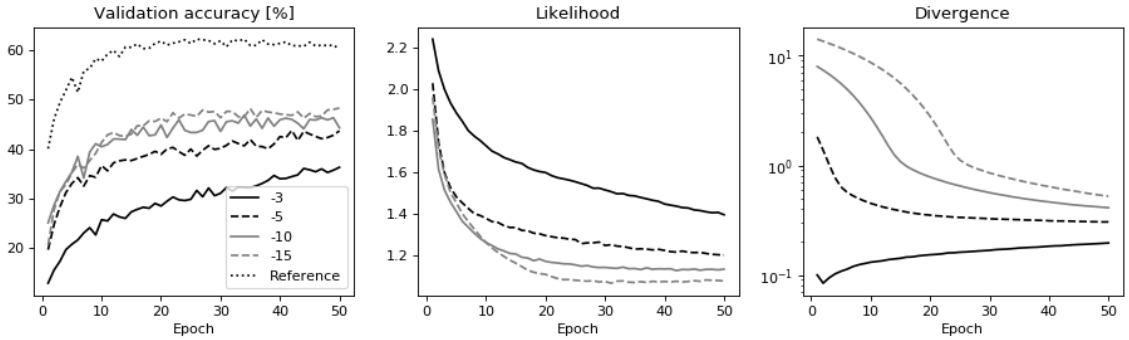

Figure A.1: Comparison of different ρ for the variational posteriors for LeNet-5 on CIFAR10. The relation to the standard deviation σ is $\sigma(\rho) = \ln(1 + e^{\rho})$.

Benchmarks

Using the setup described in this paper, each optimizer was able to obtain the accuracies in fig. A.2.

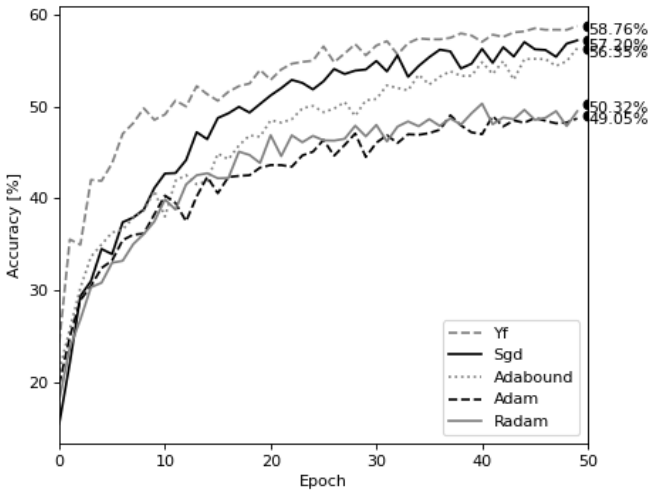

Figure A.2: Comparison between different optimizers for CIFAR10. These are best performers for each optimizer out of 486 runs in total.

Appendix B

Maximum likelihood networks

In this appendix we will look more closely into maximum likelihood networks. We will focus on the loss functions and derive it for the cases of regression and classification.

For predictive tasks, we want to model the likelihood $\mathrm{P}(y|x, w)$ for some given input data x, labels y and model parameters w chosen using maximum likelihood. Mathematically, this can be written as

$$w = \arg\max_{w^*} \mathrm{P}(y|x, w^*). \tag{B.0.1}$$

For this equation to be useful, we need to make some assumptions on the data (and from there on choose our model). Depending on what y we expect we can make some inferences. If we expect the output data to be continuous we should opt for normal distributions (unless we have information persuading us to do otherwise). If we instead expect discrete output, the go-to choice is to use categorical distributions. Now, let us see what happens with our framework for these two choices.

Regression

Regression tasks usually treat continuous output data, rendering normal distributions rather suitable, i.e. $\mathrm{P}(y|x, w) = \mathcal{N}(\mu, \sigma)$. This leaves is with the mission to find μ and σ. For μ we can model it directly in the network as $\mu = \mu(x, w)$. For σ use a constant value, for reasons that will be clear soon. This gives us sufficient information to derive the likelihood $\mathcal{L}$. Let us write out explicitly;

$$\mathcal{L} = (2\pi\sigma)^{-1/D} \exp\left[-\left(\frac{y_i - \mu(x_i, w)}{2\sigma}\right)^2\right]. \tag{B.0.2}$$

This heinous beast is not something one wants to optimize directly. However, maximizing the likelihood is equivalent to minimizing the log likelihood. If we instead consider the log likelihood, we obtain

$$\log \mathcal{L} = -\sum_i \frac{1}{D} \log(2\pi\sigma) + \left(\frac{y_i - \mu(x_i, w)}{2\sigma}\right)^2. \tag{B.0.3}$$

When using log likelihood, we choose our w from argument minimization instead of maximization. This also allows us to ignore the constant terms and factors (as they

will not matter in an optimization problem). Then the weights should be chosen from

$$w = \arg\min_w \sum_i \left(y_i - \mu(x_i, w)\right)^2 . \tag{B.0.4}$$

This is nothing but the standard mean square error loss. We obtained this simple expression because we assumed σ to be a constant. If we had instead also modelled $\sigma = \sigma(x, u)$ for some other weights u, the log likelihood would turn into something that is not as easily treated.

Classification

In the case of classification tasks, we have discrete output data. Then we assume our data obeys a categorical distribution,

$$\mathrm{P}(y_i|x, w) = p_i(x, w). \tag{B.0.5}$$

In order words, our network will model the probabilities of the categorical distribution. We can reformulate our distribution and impose our maximum likelihood framework on it;

$$\mathrm{P}(y_i|x, w) = \arg\max_w \prod_k p_k^{\delta_{ik}}(x, w) \tag{B.0.6}$$

where δ_{ik} is the Kronecker delta. As in the previous section, it is much easier to minimize the log likelihood than maximizing the likelihood;

$$\log \mathcal{L}_i = \sum_k \delta_{ik} \log p_k(x, w) \tag{B.0.7}$$

Interestingly enough, this can be identified as cross-entropy[1] between the produced distribution and the true distribution.

[1] Cross entropy is a information theoretic measure between two distinct distributions.

Appendix C

More on optimizers

Here we will peek into the reign of adaptive algorithms. The idea behind adaptive algorithms is to use a unique learning rate for each parameter. This is accomplished by scaling the base learning rate with some variance-like measure. This measure is often the raw, uncentered variance[1].

In order to make things as clear as possible I will not use vector notation. It is easily extendable to vectors but requires element-wise operations that will make the derivations unnecessarily cluttered and fuzzy.

Adagrad

The adaptive gradient algorithm (Adagrad) uses a cumulative gradient to improve the learning rate. It was introduced in [26] and was among the first the approaches to efficiently utilize accumulated square gradients. Define the cumulative gradient as

$$G_t = \sum_k^t g_k^2. \tag{C.0.1}$$

Using these definitions (and introducing some small ϵ for numerical stability), Adagrad takes on the update rule

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t. \tag{C.0.2}$$

Adagrad performs great if it is capable of converging quickly. However, as G_t grows $1/G_t$ decreases. This results in an ever decreasing learning rate and the algorithms stops learning at some point. This is the major drawback of Adagrad.

Adam

Adaptive moment estimation (Adam) is the cornerstone of adaptive moment algorithms [15]. It estimates both the mean m_t and variance v_t of the gradient and uses them in the update scheme. Analogously to the definition of classical momentum, mean and variance are defined with exponential moving averages. Introducing the smoothing parameters β_1 and β_2, the estimation update rule becomes

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t, \qquad v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2. \tag{C.0.3}$$

[1] This is equivalent to assuming the true mean of the gradient is zero, which we have no reason not to assume.

However, since the moving sum is finite for finite t, one must introduce the correction

$$\hat{m}_t = m_t/(1-\beta_1^t) \qquad \hat{v}_t = v_t/(1-\beta_2^t). \tag{C.0.4}$$

Combining the corrected moments, the update rule becomes

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t. \tag{C.0.5}$$

AMSGrad

In [27] several suggestions to fix the flaws of Adam-like algorithms are put forward. Adam and RMSprop has several occasions when they converge less than desirable minimas. The AMSGrad algorithm is an attempt to fix this. Instead of the bias correction, they utilize

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t) \tag{C.0.6}$$

and an update scheme for β_2.

Adabound and AMSBound

Adaptive optimizers tend to perform well initially but worse than SGD after a given amount of time [24]. The approach with AdaBound [12] is to set dynamic upper and lower bounds on $\alpha/\sqrt{\varphi^*}$ and clip it whenever it exceeds those bounds. Initially these bounds are set to $[0, \infty)$ (and the optimizer is equivalent to Adam), which then during training approaches $[\alpha^*, \alpha^*]$ to which it starts behaving as SGD. Analogously, AMSBound is obtained by enforcing max-choice of the variance as in eq. (C.0.6). AdaBound accomplishes the update clipping by clipping the second order estimation, that is

$$\alpha/\sqrt{\varphi^*} = \mathrm{Clip}(\alpha/\sqrt{\varphi^*}, \eta_l(t), \eta_u(t)) \tag{C.0.7}$$

where the boundary functions are

$$\eta_l(t) = \alpha^* - \frac{\alpha^*}{(1-\beta_2)t+1} \qquad \eta_u(t) = \alpha^* + \frac{\alpha^*}{(1-\beta_2)t}. \tag{C.0.8}$$

Here α is the initial learning rate (the one used by Adam) and α^* is the final learning rate (the one used by SGD). We also note that due to existence of two different learning rates, setting learning rate schedules is no longer as straightforward.

YellowFin

YellowFin is an algorithm that keeps (α, μ) in the robust region while simultaneously attempting to minimize the distance to the nearest local minimum. This is achieved using rough estimates of the distance to the minimum

$$d_t = \left\langle \frac{\langle ||g_t|| \rangle}{\langle ||g_t^2|| \rangle} \right\rangle_\beta, \tag{C.0.9}$$

the central variance

$$v_t = \langle ||g_t^2|| \rangle_\beta - \langle ||g_t|| \rangle_\beta^2, \tag{C.0.10}$$

and upper and lower bounds on the curvature

$$\begin{cases} h_{\max} = \langle || \max g || \rangle_\beta \\ h_{\min} = \langle || \min g || \rangle_\beta \end{cases} . \tag{C.0.11}$$

The step is then taken using SGD and with μ_t and η_t determined by those constraints [16].