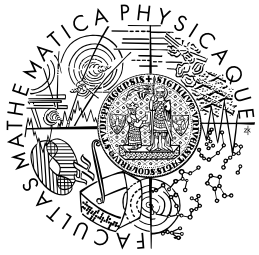


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Michal Belák

Active learning for Bayesian neural networks in image classification

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Ing. Tomáš Šabata
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my most sincere gratitude to my supervisor, Ing. Tomáš Šabata, for guiding me through my work on this thesis, his insightful advice and suggestions for improvements. I would also like to thank my family, who gave me endless support and encouragement throughout my studies.

Title: Active learning for Bayesian neural networks in image classification

Author: Bc. Michal Belák

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Ing. Tomáš Šabata, Department of Applied Mathematics, Faculty of Information Technology, Czech Technical University in Prague

Abstract: In the past few years, complex neural networks have achieved state of the art results in image classification. However, training these models requires large amounts of labelled data. Whereas unlabelled images are often readily available in large quantities, obtaining labels takes considerable human effort. Active learning reduces the required labelling effort by selecting the most informative instances to label. The most popular active learning query strategy, uncertainty sampling, uses uncertainty estimates of the model being trained to select instances for labelling. However, modern classification neural networks often do not provide good uncertainty estimates. Bayesian neural networks model uncertainties over model parameters, which can be used to obtain uncertainties over model predictions. Exact Bayesian inference is intractable for neural networks, however several approximate methods have been proposed. We experiment with three such methods using various uncertainty sampling active learning query strategies.

Keywords: machine learning, active learning, neural networks, Bayesian neural networks

Contents

Introduction

Image classification is a machine learning task of selecting (or predicting) a class from a set of classes, given a 2-dimensional input image. *Convolutional neural networks* (CNNs) [1] have shown excellent performance at such tasks, surpassing human performance at many [2], given a large collection of labelled training images, which can be difficult to obtain in practice.

Active learning addresses the issue of demand for large amounts of labelled training data by interactively querying a labeller for selected labelled examples to learn from. Querying the most informative examples can be utilized to reduce the number of samples required to reach a certain level of model performance compared to normal supervised learning. Training examples can be chosen according to various criteria – *active learning frameworks*. This thesis focuses on one particular active learning framework called *uncertainty sampling*, in which new training examples are queried from a pool of unlabelled examples according to the highest uncertainty of predictions of the current model. However, it is difficult to estimate uncertainty of predictions of traditional neural networks, which tend to make over-confident predictions.

Bayesian neural networks augment traditional neural networks with *Bayesian inference* to infer a posterior probability distribution over parameters of a neural network given a set of training data. The distribution over model parameters can be used to obtain a distribution over predictions, which captures the uncertainty. However, exact Bayesian inference is intractable for practical neural networks due to the high number of model parameters. Therefore, various methods have been used to find distributions approximating the true posterior.

Goals

The main goals of the thesis are the following:

1. Review Bayesian neural networks for image classification.
2. Study active learning and uncertainty sampling active learning methodology.
3. Compare various approximate Bayesian neural network methods in the setting of uncertainty sampling active learning.
4. Evaluate and compare chosen methods on commonly-used image classification datasets in terms of model performance and computational complexity of training and inference.

Structure

This thesis is divided into five chapters. The first chapter reviews the problem of image classification. The second chapter introduces Bayesian neural networks, convolutional Bayesian neural networks and approximate Bayesian inference methods for neural networks. The third chapter provides an overview of

active learning and reviews uncertainty sampling active learning methods. In the fourth chapter, we describe how uncertainty estimates can be obtained from Bayesian neural networks and utilized in uncertainty sampling active learning. Finally, in the fifth chapter we describe our experiments and present our results.

1. Image classification

Classification

Classification is one of the two main task studied in the field of supervised machine learning, the other one being regression. In classification, the task is to assign (or *predict*) the correct class from a given set of classes. Binary classification refers to a classification problem with two classes (typically referred to as 0 and 1, or positive and negative) and multi-class classification refers to classification into more than two classes.

Image classification

Image classification is a special type of classification, where the input data consist of 2-dimensional images, usually represented as numeric matrices. Images often contain multiple *channels*, for example red, green and blue for an RGB color image. Image classification tasks are typically multi-class.

1.1 Traditional methods

Due to the typically high dimensionality and importance of spatial structure of images, most common machine learning algorithms, which are suited mainly for tabular data, tend to perform poorly in image classification tasks. To use most typical classifiers, such as support-vector machines [3], logistic regression [4], or decision-tree-based classifiers [5, 6, 7], one typically needs to first extract *features* from given images using various computer vision techniques – for example positions of edges [8, 9], Gabor features [10], scale-invariant feature transform (SIFT) [11]. However, it may not clear a priori what types of features may be useful for a given task.

Using many possible types of features can lead to problems with high dimensionality of classifier inputs. Therefore, a concept feature reduction is often employed to select a subset of the most informative features for classification

Nowadays, many of these problems are circumvented by using convolutional neural networks for image classification, which are capable of automatic feature extraction, also known as representation learning, where a model is used both to learn useful features and to classify according to these features.

1.2 Neural networks for image classification

In this section, we introduce methods of image classification based on *artificial neural networks* (ANNs). Artificial neural networks are a class of machine learning models originally inspired by nervous systems of living organisms.

Artificial neural networks are currently one of the most studied and applied methods in image processing thanks to the ability to learn highly complex non-linear feature representations.

1.2.1 Feed-forward neural networks

In this section we briefly introduce *feed-forward neural networks*. A feed-forward neural networks generally consists of neurons connected by directed connections, which do not form a cycle.

The most basic feed-forward neural network, the perceptron [12], consists of just a single neuron. A perceptron is a simple binary classification model classifying into classes $\{0, 1\}$. A neuron consists of a vector of weights $\boldsymbol{w}$, a bias term b and an *activation function* f, which in case of the perceptron is the Heaviside step function:

$$H(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases} .$$

For an input vector $\boldsymbol{x}$, the output of a neuron is $y = f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$, where $\boldsymbol{w} \cdot \boldsymbol{x} = \sum_i w_i x_i$ signifies the dot product.

Neurons in more complex feed-forward neural networks are often organized in one or more disjunct ordered *layers*. Such networks are commonly referred to as *multi-layer perceptrons*. In a multi-layer perceptron, the input of each neuron in a given layer can only be connected to outputs of neurons in the previous layer, and conversely, the output can only be connected to the input of the next layer. The first layer of a feed-forward neural network is called the input layer, the last layer is called the output layer, and any layers in-between are called hidden layers. The number of layers is referred to as *depth*.

An example of a class of artificial neural networks, which are not feed-forward neural networks, is the class of recurrent neural networks (RNNs) [13, 14, 15], which contain cyclical connections, often from a cell (similar to a layer in a MLP) to itself. RNNs are widely used in sequence modelling, and especially, natural language processing.

Fully-connected layers

There are multiple types of layers which are commonly used in layered feed-forward neural networks. The most basic type of layer is a *fully-connected layer*, where the input of each neuron is connected to the outputs of *all* neurons in the previous layer.

The output y_j of a single neuron j in the l-th layer L_l is computed as follows:

$$y_j = f_j \left(\sum_{i \in L_{l-1}} w_{i,j} y_i + b_j \right) ,$$

where $w_{i,j}$ denotes the weight of the connection from the neuron i to the neuron j and b_j. The above computation can be conveniently expressed in matrix notation as:

$$\boldsymbol{y_l} = f_l \left(\boldsymbol{W_l} \boldsymbol{y_{l-1}} + \boldsymbol{b_l} \right) ,$$

where $\boldsymbol{y_l}$ is the vector of outputs of the l-th layer, consisting of outputs of all individual neurons in the l-th layer, $\boldsymbol{y_0} = \boldsymbol{x}$ is the input to the neural network, $\boldsymbol{W_l}$ is the weight matrix between the $l-1$-st layer and l-th layer, $\boldsymbol{b_l}$ is the

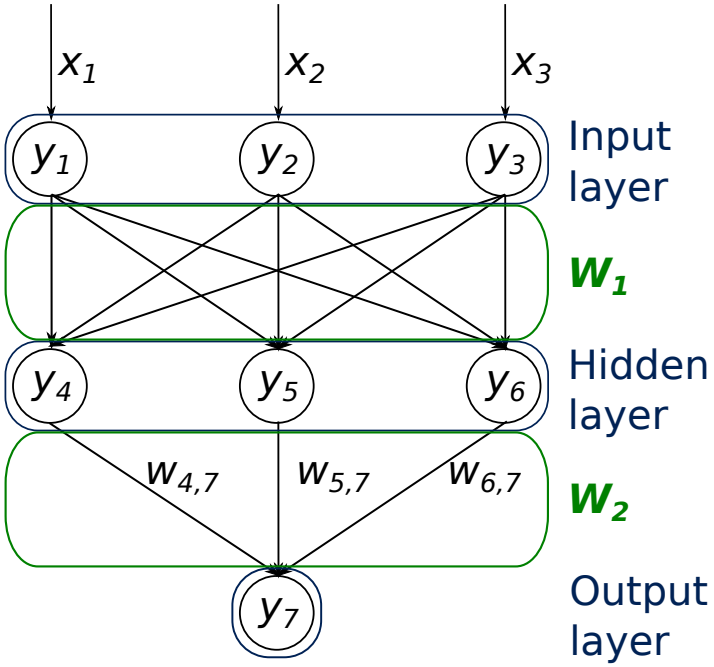


Figure 1.1: Schematic of a multi-layer perceptron with a single fully-connected hidden layer.

bias vector of the l-th layer, and $f_l \colon \mathbb{R}^{|L_l|} \to \mathbb{R}^{|L_l|}$ is the element-wise activation function of the l-th layer.

Figure 1.1 illustrates a feed-forward neural network with 3 fully-connected layers: an input layer containing 3 neurons, a single hidden layer containing 3 neurons and a single neuron in the output layer. The nodes correspond to neurons, labelled with the inputs. The connection weights from the input layer are not labelled; they are summarized in the matrix $\boldsymbol{W_1} \in \mathbb{R}^{3\times3}$. The weights to the output layer are labelled both by the weight matrix and the individual weights: $\boldsymbol{W_2} = [w_{4,7}, w_{5,7}, w_{6,7}]$.

Activation functions

Examples of commonly used activation functions include:

- linear: $f(x) = ax$,
- logistic sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$,
- hyperbolic tangent: $\tanh(x) = \frac{2}{1+e^{-2x}} - 1 = 2\sigma(2x) - 1$,
- ReLU: $\mathrm{ReLU}(x) = \max(0, x)$,
- softplus: $\mathrm{softplus}(x) = \ln(1 + e^x)$.

Linear activation is only useful in output layers (in regression models), because adding a hidden layer with linear activation does not bring any benefit –

a composition of linear transformations is a linear transformation, which can be learned in a single layer.

Sigmoid and hyperbolic tangent functions are very similar in that they *squash* the input to a narrow range. Sigmoid activation is typically used in the output layer of binary classification models, because the output between 0 and 1 can be interpreted as the estimated probability of the 1-class. Hyperbolic tangent is more commonly used in hidden layers mostly because it is symmetric, however it does not solve optimization issued of sigmoid activation in deep architectures [16].

ReLU (rectified linear unit) [17] is used to introduce non-linearity and is commonly used in hidden layers of deep networks. Since the output of ReLU is 0 for any non-positive input value, it can happen during training that a neuron gets stuck in an inactive state, always outputting 0. Various modifications of ReLU have been proposed to mitigate this issue, for example $LeakyReLU(x) = \max(0.01x, x)$.

The softplus activation function is a smooth approximation of ReLU commonly used in probabilistic models, which will be introduced in later chapters. The derivative of softplus is the logistic sigmoid function, which is a smooth approximation of the Heaviside step function – the derivative of ReLU.

Additionally, we introduce an activation function called softmax, which is applied on the complete output vector of a layer instead of each term separately, defined as follows:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}([z_1, \ldots, z_n]) = \left[\frac{e^{z_1}}{\sum_{i=1}^{n} e^{z_i}}, \ldots, \frac{e^{z_n}}{\sum_{i=1}^{n} e^{z_i}} \right]. \tag{1.1}$$

The output vector of a softmax layer in n-class classification consists of n non-negative numbers with a sum of 1 and is commonly interpreted as an estimated probability of each class. Therefore, softmax is often used in the output layer of neural networks for multi-class classification. Note that softmax is a generalization of logistic sigmoid for multiple outputs and the following holds: $\sigma(x) = \text{softmax}([x, 0])_1$.

1.2.2 Convolutional neural networks

Convolutional neural networks (CNNs), first introduced by LeCun [1], are a special type of neural networks used for data with a grid-like spatial or temporal structure [18]. Examples of such data include time series and other sequences, which can be thought of 1-dimensional data, or images, which can be thought of as a 2-dimensional pixel grid, or objects, which can be thought of as a 3-dimensional voxel grid. As the name suggests, CNNs apply a special type of mathematical operation called *convolution*, in contrast to previously mentioned fully-connected neural networks, which perform matrix multiplication. In addition to convolution, an operation called *pooling* is used in almost all CNNs.

Convolution operation

Convolution, in its most general form, is an operation on two functions (usually denoted with an asterisk) f and g, defined as follows:

$$(f * g)(t) = \int f(x)g(x-t)dt.$$

In convolution terminology, the first argument (f in this case) is often referred to as *input*, the second argument as *kernel* or *filter* and the output as *feature map*.

For example, if kernel g is a probability density function, then convolving any input function f with g produces a feature map corresponding to a weighted average of the values of f.

The continuous convolution operation has a discrete version for functions defined on integers, which is defined as follows:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[n]g[n-m]\,.$$

Convolution can also be performed over multiple dimensions at a time. For example, the 2-dimensional discrete convolution for a 2-dimensional input image I and a 2-dimensional kernel K is defined as follows:

$$(I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m,j-n] = \sum_m \sum_n I[i-m,j-n]K[m,n]\,,$$

where the second equality is due to the commutative property of the convolution operation. In the above formula, we can observe that the kernel is *flipped* in relation to the input, i.e. as the index into the input increases, the index into the kernel decreases. Convolution is usually defined this way to obtain the commutative property, which is useful in mathematical proofs, however it is not necessary in neural networks. Therefore, CNN implementations usually perform an operation called *cross-correlation*, which is equivalent to discrete convolution without flipping the kernel:

$$(I * K)[i,j] = \sum_m \sum_n I[i+m,j+n]K[i,j].$$

The convolution kernel could be of any size up to the size of the input, however, a kernel smaller than the inputs is normally used, and can be thought of as overlaying a sliding window over the input and multiplying the corresponding positions in the input and the kernel, summing up the products to produce a single output value for each position of the sliding window.

Only positions where the entire kernel sliding window lies within the input image are considered. This means that the sliding window cannot be placed on edge positions if the kernel is larger than 1×1, resulting in a reduced resolution of the output feature map. Where this effect is undesirable, a method called *padding* can be employed. It appends virtual values (typically a constant such as 0) to the edges of the input, allowing the kernel sliding window to visit positions on the edge.

Finally, the number of positions the sliding window is moved each time is referred to as *stride* and it can be different for every spatial dimension. Using a stride with a value other than 1 results in a reduction of resolution of the output feature map.

Motivation

The main advantages of using the convolution operation for data with spatial structure, such as images, are the following:

- sparse interactions,
- parameter sharing,
- equivariant representations,
- ability to work with inputs of variable size.

Next we briefly describe each of these concepts. A full description can be found in a book by Goodfellow et al. [18].

Sparse interactions refer to the fact that each output does not necessarily depend on all inputs, which is accomplished by using a smaller kernel than the input. This is in contrast to matrix multiplication used in fully-connected networks, where each output depends on all inputs. For example, an input image might contain millions of pixels, however detection of small local features can be accomplished by only looking at a small area of a few pixels. The benefits are mainly reduced number of parameters and required computations. For example, for an input image of $N \times N$ pixels, to produce a feature map of the same size requires $O(N^4)$ operations in a fully connected manner, while a convolution with a kernel of size $K \times K$ requires $O(N^2K^2)$ operations, since an input image may be hundreds by hundreds of pixel, compared to commonly used kernel sizes, which can be for example 5×5 pixels.

Parameter sharing refers to the property of the convolution operation, where each parameter is used in multiple outputs. This is once again a contrast to fully-connected layers, where each parameter is used in the output exactly once. The parameter sharing used in convolutions allows a convolutional neural network to learn a single set of parameters to represent a certain concept rather than learning a separate set of parameters for each position. The benefits lie mostly in efficiency of model storage and statistical efficiency – being able to express a given model in fewer parameters.

This type of parameter sharing results in **equivariance** of convolutional layers to input translation. Two functions f and g are equivariant iff $f(g(x)) = g(f(x))$ for all x, or equivalently $f \circ g = g \circ f$. In image processing, this means that if an image is used as an input of a convolution and then the same image is used, with every pixel shifted by the same number of positions, the resulting feature maps will be the same up to the shift between the input images. This property is particularly beneficial in early layers of a convolutional neural network, where it is desirable to extract the same simple features – such as edges – in every position of an input image.

Ability to work with inputs of variable size comes from the fact that a small local convolution operation is repeated multiple times in an input image, instead of a global operation such as matrix multiplication, where the size of the matrix is determined by the size of the input.

1.3 Training of neural networks

In general, training of a machine learning refers to the process of finding an optimal set of parameter values for a given *loss function* which best fit a given set of training data.

1.3.1 Maximum likelihood estimation

As most machine learning models, neural networks are usually trained by finding optimum point estimates for a set of model parameters $\boldsymbol{w}$ using *maximum likelihood estimation* [19, 18]. Maximum likelihood estimation finds a set of parameters maximizing a *likelihood function* $P(\mathbb{X}; \boldsymbol{w})$ ($P(\mathbb{X} \mid \boldsymbol{w})$ is also used in literature), so that under the assumed model, the observed data is the most probable.

Let $\mathbb{X} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ be a set of training examples, which are drawn independently from an unknown underlying data-generating distribution p_{data}. For classification tasks, we assume labels y_i to be integer indices of individual classes and a parametric family of classification models $p_{\text{model}}(. \mid \boldsymbol{x}; \boldsymbol{w})$, parameterized by a set of parameters $\boldsymbol{w}$, to output a categorical probability distribution estimating probability of each class, given a feature vector $\boldsymbol{x}$, with $p_{\text{model}}(c \mid \boldsymbol{x}; \boldsymbol{w})$ denoting estimated probability of class c. Let $\hat{p}_{\text{data}}$ be the empirical data distribution of data observed in the set of training data $\mathbb{X}$. Applying maximum likelihood estimation yields a *loss function* referred to as **negative log-likelihood** or **cross-entropy** loss.

$$\begin{aligned}
\boldsymbol{w}^*_{MLE} &= \arg\max_{\boldsymbol{w}} p_{\text{model}}(\mathbb{X}; \boldsymbol{w}) \\
&= \arg\max_{\boldsymbol{w}} \prod_{i=1}^{N} p_{\text{model}}(y_i \mid \boldsymbol{x}_i; \boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{w}} \sum_{i=1}^{N} -\log p_{\text{model}}(y_i \mid \boldsymbol{x}_i; \boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{w}} \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\text{data}}}[-\log p_{\text{model}}(y \mid \boldsymbol{x}; \boldsymbol{w})] \\
&= \arg\min_{\boldsymbol{w}} H(\hat{p}_{\text{data}},\, p_{\text{model}}(. \mid .; \boldsymbol{w})) \\
&= \arg\min_{\boldsymbol{w}} D_{\text{KL}}(\hat{p}_{\text{data}} \,\|\, p_{\text{model}}(. \mid .; \boldsymbol{w}))\,,
\end{aligned} \tag{1.2}$$

where

$$H(P, Q) = \mathbb{E}_{x\sim P(x)}\left[\log Q(x)\right] \geq H(P) = \mathbb{E}_{x\sim P(x)}\left[\log P(x)\right]$$

stands for the *cross-entropy* between two distributions P and Q, where generally $H(P, Q) \neq H(Q, P)$, $H(P, Q) \geq H(P)$ and $H(P, Q) = H(P)$ iff $P(x) = Q(x)$ almost everywhere. Further,

$$\begin{aligned}
D_{\text{KL}}(P \,\|\, Q) &= \mathbb{E}_{x\sim P(x)} \log \frac{P(x)}{Q(x)} = \mathbb{E}_{x\sim P(x)}\left[\log P(x) - \log Q(x)\right] \\
&= H(P, Q) - H(P) \geq 0
\end{aligned}$$

stands for the *Kullback-Leibler (KL) divergence* (also called relative entropy) [20] from a distribution Q to a reference distribution P. KL divergence is commonly

used as a measure of difference between two distributions. Analogously to cross-entropy, $D_{\mathrm{KL}}(P \,\|\, Q)$ is in general not equal to $D_{\mathrm{KL}}(Q \,\|\, P)$ and $D_{KL}(P \,\|\, Q) = 0$ iff $P(x) = Q(x)$ almost everywhere.

Finding the optimal set of parameters can be thought of as an optimization problem of finding the global minimum of a loss function, specifically in the case of assumed categorical output distribution:

$$\mathcal{L}_{\mathrm{NLL}} = \frac{1}{N}\sum_{i=1}^{N} -\log p_{\mathrm{model}}(y_i \mid \boldsymbol{x}_i; \boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\mathrm{data}}}[-\log p_{\mathrm{model}}(y \mid \boldsymbol{x}; \boldsymbol{w})] \quad (1.3)$$

Applying similar reasoning to a model which is assumed to output the mean of a normal (Gaussian) distribution with a fixed variance gives us the **mean squared error (MSE)** loss, which is the default loss function for regression tasks, defined as:

$$\mathcal{L}_{\mathrm{MSE}} = \frac{1}{N}\sum_{i=1}^{N} (y_i - \hat{y}(\boldsymbol{x}_i; \boldsymbol{w}))^2 = \mathbb{E}_{(x,y)\sim\hat{p}_{\mathrm{data}}}[(y - \hat{y}(\boldsymbol{x}; \boldsymbol{w}))^2]\,, \quad (1.4)$$

where $\hat{y}(\boldsymbol{x}; \boldsymbol{w})$ is the output of a model with weights $\boldsymbol{w}$ for input $\boldsymbol{x}$ and labels y_i are real numbers.

1.3.2 Gradient-based optimization

The most commonly used methods of optimizing a given loss function are based on *gradient descent*, which involves iteratively computing partial derivatives of a loss function $\mathcal{L}$ w.r.t. to each parameter of a model and adjusting each parameter by a small amount in the opposite direction of the respective partial derivative. The gradient descent update rule, performed for each weight w_i at each iteration t is:

$$w_i(t+1) \leftarrow w_i(t) - \alpha \frac{\partial \mathcal{L}}{\partial w_i(t)}\,,$$

or, using vector and gradient notation:

$$\boldsymbol{w}(t+1) \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}(t)} \mathcal{L}\,,$$

where α is commonly referred to as *learning rate* and is a *hyper-parameter*, meaning that it is not learnt from data. Learning rate can either be constant or change (decrease) with time according to a pre-determined *schedule*. Training is typically divided into *epochs*, where in each epoch, weight updates for all of the training examples are performed.

Stochastic and mini-batch gradient descent

Note that in Equations (1.2), (1.3) and (1.4), each of the loss functions is expressed as an expectation over a training data distribution. Therefore, using an independently drawn sample from the training data to compute loss and gradient of the loss provides unbiased estimates of the respective true values. Specially, computing the gradient from each example individually (sample size 1) is usually referred to *stochastic gradient descent (SGD)* [21].

An algorithm using a sample of a size between 1 and the size of the whole dataset (called *mini batch*) is referred to as *mini-batch gradient descent*. Mini-batch size is a hyperparameter. Using larger mini-batches results in lower variance of computed gradients, but fewer updates for a given number of processed examples.

Practically, neural networks are typically trained on graphical processing units (GPUs) or specialized hardware, which are well-suited for batch computation and using larger batches typically results in faster training given sufficient operating memory.

SGD is guaranteed to converge to a *local* optimum of the loss function provided correct step sizes (learning rates) are used. In particular, a sufficient condition for convergence is using a sequence of positive learning rates α_t at step t such that [22]

$$\sum_{t=1}^{\infty} \alpha_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty \,.$$

Backpropagation

Backpropagation (term coined by Rumelhart et al. [23]) is an algorithm used to efficiently compute partial derivatives of a loss function with respect to each parameter of a parametric model.

Each layer of a layered feed-forward neural network can be thought of as a function of inputs and parameters of the layer. Thus, a multi-layered feed-forward neural network can be thought of as a function composed of individual layer functions. Since multiplication, addition, all commonly used activation functions, and other operations, are all differentiable w.r.t. their respective inputs and parameters, it is possible to compute partial derivatives of a loss function w.r.t. to every parameter in every layer by repeatedly applying the *chain rule*. Let $y = f(x)$, $z = g(y) = g(f(x))$. The chain rule states:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

Applying the chain-rule to the loss function of a neural network yields the backpropagation algorithm. Let

$$z_j(t) = \sum_{i \in L_{l-1}} w_{i,j}(t) y_i(t) + b_i(t)$$

be the pre-activation *potential* of neuron j at time t and let

$$\delta_j(t) = \frac{\partial \mathcal{L}}{\partial z_j(t)} \,.$$

For a weight $w_{i,j}$ and a bias b_j we have:

$$\frac{\partial \mathcal{L}}{\partial w_{i,j}(t)} = \frac{\partial \mathcal{L}}{\partial z_j(t)} \frac{\partial z_j(t)}{\partial w_{i,j}(t)} = \delta_j(t) y_i(t)$$
$$\frac{\partial \mathcal{L}}{\partial b_j(t)} = \frac{\partial \mathcal{L}}{\partial z_j(t)} \frac{\partial z_j(t)}{\partial b_j(t)} = \delta_j(t)$$

The error term δ_j is computed differently for output neurons and for hidden neurons. For an output neuron j it can be computed directly:

$$\delta_j(t) = \frac{\partial \mathcal{L}}{\partial y_j(t)} \frac{\partial y_j(t)}{\partial z_j(t)} = \frac{\partial \mathcal{L}}{\partial y_j(t)} f_j'(z_j(t))$$

For a hidden neuron j in hidden layer l:

$$\begin{aligned} \delta_j(t) &= \frac{\partial \mathcal{L}}{\partial y_j(t)} \frac{\partial y_j(t)}{\partial z_j(t)} = \sum_{k \in L_{l+1}} \left(\frac{\partial \mathcal{L}}{\partial z_k(t)} \frac{\partial z_k(t)}{\partial y_j(t)} \right) f_j'(z_j(t)) \\ &= \sum_{k \in L_{l+1}} \left(\delta_k(t) w_{j,k}(t) \right) f_j'(z_j(t)) \end{aligned} \tag{1.5}$$

As we can see in Equation (1.5), to compute δ values for neurons a given layer, we need to have first computed all δ values for all following layers. The process of computing δ values (and subsequently partial derivatives w.r.t. to weights) from the output layer towards the input layer is called *backward propagation*. Finally, we can formulate the SGD algorithm using backpropagation to compute gradients in simple pseudocode:

Algorithm 1: SGD training with backpropagation

Data: learning rate α, training data $\mathbb{X}$, initial weights $\boldsymbol{w}(0)$

$t \leftarrow 0$
repeat
  for $(\boldsymbol{x}, y)$ *in random permutation of* $\mathbb{X}$ **do**
    Set the output vector of neurons in the input layer $\boldsymbol{y}_0(t)$ to the input $\boldsymbol{x}$
    Compute $y_j(t)$ for each neuron j
    Compute loss at the output layer using the correct output y
    Compute $\delta_j(t)$ for each neuron using backpropagation from the output
    $w_{i,j}(t+1) \leftarrow w_{i,j}(t) - \alpha \delta_j(t) y_i(t)$ for each weight $w_{i,j}$
    $b_j(t+1) \leftarrow b_j(t) - \alpha \delta_j(t)$ for each bias b_j
    $t \leftarrow t + 1$
until *convergence*;

Momentum

Momentum [24] is a simple modification to the gradient descent algorithm. In addition to the gradient of the loss function at the current iteration, the weights are also adjusted in the direction of the gradient in the previous iteration. Let $\Delta \boldsymbol{w}(t) = \boldsymbol{w}(t+1) - \boldsymbol{w}(t)$ denote the change in the weight vector at iteration t. In case of vanilla SGD $\Delta \boldsymbol{w}(t)$ is given by

$$\Delta \boldsymbol{w}(t) = -\alpha \nabla_{\boldsymbol{w}(t)} \mathcal{L} \,.$$

Momentum also adjusts the weights in the direction of the change in the previous step:

$$\Delta \boldsymbol{w}(t) = -\alpha \nabla_{\boldsymbol{w}(t)} \mathcal{L} + \beta \Delta \boldsymbol{w}(t-1),$$

where the momentum term β controls how much the previous change influences the current change. Momentum keeps a running exponentially decaying estimate of the first moment referred to as *velocity*. Denoting the velocity vector by $\boldsymbol{v}$, the momentum update can be equivalently written as:

$$\begin{aligned} \boldsymbol{v}(t) &\leftarrow \beta \boldsymbol{v}(t-1) - \alpha \nabla_{\boldsymbol{w}(t)} \mathcal{L} \\ \boldsymbol{w}(t+1) &\leftarrow \boldsymbol{w}(t) + \boldsymbol{v}(t)\,. \end{aligned}$$

Using momentum helps convergence by reducing variance of the gradients. To illustrate a case where momentum is especially effective consider a loss surface descending in a long and narrow valley with steep sides with an optimum at the bottom. In such case, the gradients are almost perpendicular to the long axis of the valley with only a small component aiming in the desired direction down the valley, which causes the parameters to oscillate back and forth between the sides of the valley. Momentum dampens the oscillations by averaging out the large lateral components and summing up the small lengthwise components of the gradients, which causes the adjustments to gradually point closer to the direction of the long axis.

Adaptive optimizers

Choosing the learning rate in standard SGD or SGD with momentum can be difficult, often due to different sensitivity of the loss function in different directions. Under the assumption that these directions are somewhat aligned with the axes of the parameters, it is beneficial to use a separate learning rate for each parameter.

Various adaptive optimizers have been proposed. An early example is the heuristic called delta-bar-delta [25], which increases the local learning rate for parameters where the sign of the gradient has not changed from the last iteration, and decreases it otherwise. More recent optimizers use approaches which maintain a running estimate of the moments of the gradient. These include AdaGrad [26], which was later extended to AdaDelta [27], RMSProp (unpublished) and Adam [28]. Adam is widely used as the default optimizer in various tasks.

The Adam adaptive optimizer [26] maintains a unbiased estimates of the first moment and the second uncentered moment of the gradients. The estimate of the second moment is used to scale the global learning rate by a per-parameter adaptive factor. Algorithm 2 sketches the optimization procedure using Adam.

The default values in the algorithm are suggested by the authors of Adam [28].

Algorithm 2: Adam mini-batch optimization

Data: Training data $\mathbb{X}$, parameters $\boldsymbol{w}$, learning rate α (default 10^{-3}), momentum β_1 (default 0.9), momentum β_2 (default 0.999), constant for numeric stability ϵ (default 10^{-8})

$\boldsymbol{s} \leftarrow \boldsymbol{0}, \boldsymbol{r} \leftarrow \boldsymbol{0}, t \leftarrow 0$

repeat

  Compute gradient of the loss for a batch of $B \subseteq \mathbb{X}$: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{w}} \mathcal{L}$

  Update first moment estimate: $\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} + (1 - \beta_1)\boldsymbol{g}$

  Update second moment estimate: $\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} + (1 - \beta_2)\boldsymbol{g} \odot \boldsymbol{g}$

  Apply bias correction: $\boldsymbol{s} \leftarrow \boldsymbol{s}/(1 - \beta_1^t)$, $\boldsymbol{r} \leftarrow \boldsymbol{r}/(1 - \beta_2^t)$

  Update parameters: $\boldsymbol{w}(t + 1) \leftarrow \boldsymbol{w}(t) - \frac{\alpha}{\sqrt{\boldsymbol{r}} + \epsilon} \boldsymbol{s}$

until *stopping criterion*;

1.3.3 Regularization

Regularization refers to a set of techniques designed to mitigate *overfitting* – a common problem in machine learning. Overfitting refers to the phenomenon where a model learns to fit a particular empirical training data distribution $\hat{p}_{data}$ too closely, which may have a negative effect on *generalization*, or, in statistical terms, fitting the true data generating distribution p_{data} (using notation introduced in Section 1.3). Complex models with high capacity and many parameters, such as neural networks, are able to fit observed data very closely, which makes susceptible to overfitting if not properly regularized. Developing effective regularization techniques has long been one of the major areas of research in machine learning and especially deep learning [18].

Early stopping

Due to the iterative nature of neural network training, one of the simplest regularization techniques called *early stopping* can be utilized. Early stopping involves stopping of neural network training before parameters have converged to a local optimum of the loss function. The learned parameters therefore do not fit the observed training data as closely, which can benefit generalization.

Ensembling

Ensembling or *ensemble learning* refers to the practice of training multiple models and combining their respective outputs to obtain an ensemble model with better predictive performance than each of the constituent models.

Bias-variance tradeoff is a property of machine learning models, whereby models with lower bias in parameter estimation usually have high estimation variance and vice versa. High bias is typically due to the the model not fitting the observed data closely enough – underfitting, while high variance is often due to the model fitting the noise observed in the training data – overfitting.

A simple way of reducing variance of any general random variable X is to average multiple independent samples of the variable due to the simple identity $\text{Var}\left[\frac{1}{N}\sum_{i=1}^{N} X_i\right] = \frac{1}{N}\text{Var}\left[X_1\right]$. Thus, using a set of independent models decreases

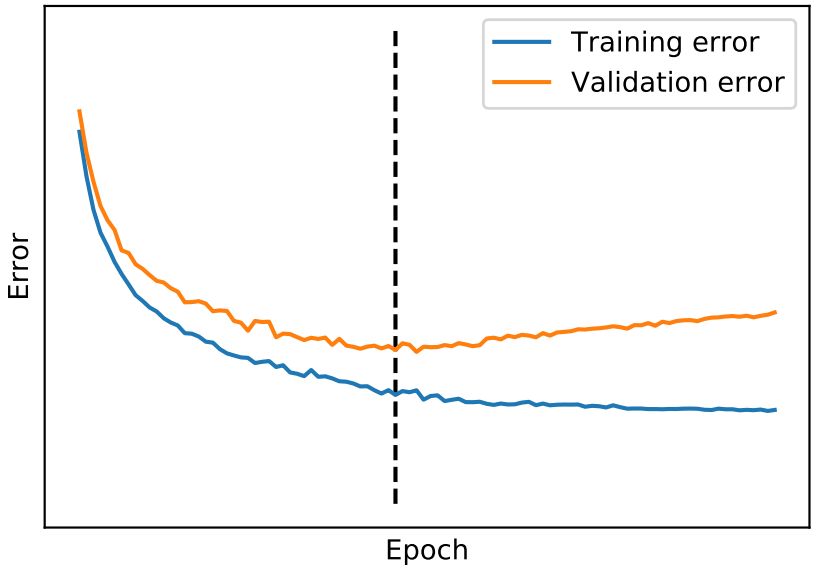


Figure 1.2: Schematic of training and validation error for an iteratively trained model. The dashed line shows approximately where overfitting starts – validation error stops decreasing and starts slightly increasing, while training error keeps slowly decreasing.

variance without increasing bias, theoretically improving predictive performance of the resulting model.

Although it can be difficult to obtain a set of models with fully independent predictions, even using an ensemble of highly correlated models can be beneficial. In case of neural networks, a simple ensemble can be obtained by repeatedly training the same architecture with different initial parameter values, which ideally converge to different local optima.

Norm regularization

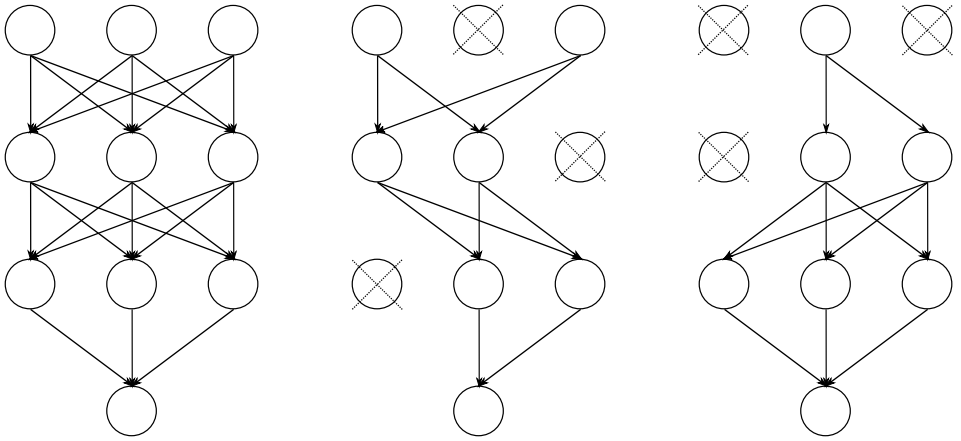
A common example of regularization technique is a family of regularization techniques called *norm regularization*, whereby a term representing the norm parameter values is added to the loss function. For example, a common norm used for regularization of neural networks is the L_2 norm:

$$\mathcal{L}_{reg} = \mathcal{L} + \lambda \|\boldsymbol{w}\|_2^2 ,$$

where $\mathcal{L}$ is a loss non-regularized loss such as negative log-likelihood (Equation (1.3)), λ is a *regularization factor* and $\mathcal{L}_{reg}$ is the resulting regularized loss which is optimized. A gradient update of weight w_i using L_2 regularized loss yields the following:

$$w_i(t+1) \leftarrow \alpha * \left(\frac{\partial \mathcal{L}}{\partial w_i(t)} + \frac{\partial \lambda \|\boldsymbol{w}(t)\|_2^2}{\partial w_i(t)} \right) = \alpha * \left(\frac{\partial \mathcal{L}}{\partial w_i(t)} + 2\lambda w_i(t) \right) .$$

Therefore, L_2 normalization is also commonly referred to as *weight decay* because when updating a weight, it is decayed towards 0 proportionally to its size. Simi-



(a) A neural network before applying dropout

(b) An example of a dropout mask applied to the network

(c) Another example of a dropout mask applied

Figure 1.3: A neural network without dropout and two examples of a binary dropout mask applied. Inspired by Figure 1 of [29].

larly, applying L_1 normalization (sum of absolute values) results in the following update:

$$w_i(t+1) \leftarrow \alpha * \left(\frac{\partial \mathcal{L}}{\partial w_i(t)} + \lambda \operatorname{sgn}(w_i(t)) \right) ,$$

where

$$\operatorname{sgn}(x) = \begin{cases} -1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ +1, & \text{if } x > 0 \end{cases} .$$

L_1 normalization decays each weight towards 0 by a constant amount λ, which results in many weights becoming exactly 0. On the other hand, L_2 regularization pushes smaller weights towards zero by a proportionally smaller amount, which results in many weights becoming small in absolute value, but not 0.

Dropout

Dropout refers to a family of regularization techniques, which aim to make neural networks more robust by injecting random parametric noise into the outputs of individual layers during training.

The most basic form (and the most widely used), commonly referred to as binary dropout, or simply dropout, introduced by Srivastava et al. [29] in 2014 (in pre-print since 2012 [30]), injects multiplicative Bernoulli noise into the output of a layer it is applied to. Injecting Bernoulli noise means randomly selecting a value of either 0 or 1, which is equivalent to randomly choosing between zeroing-out and keeping unchanged outputs of individual neurons in the layer.

Dropout rate (probability p of zeroing-out an output in case of binary dropout or variance in Gaussian dropout) is typically a hyperparameter, however a family of dropout variants called *variational dropout* ([31]) make dropout rate a learnable parameter.

Gaussian dropout [29], another common variant of dropout, multiplies outputs by Gaussian noise with mean of 1 and a variance corresponding to the dropout rate.

A network with dropout can be seen as an ensemble (with an infinite number of members in case of continuous dropout noise) of neural networks, where every possible dropout mask corresponds to a member of the ensemble. Figure 1.3 illustrates two possible members of a dropout ensemble using binary dropout in a multi-layer perceptron, where dropout is applied to every layer (except the output). Note that input neurons can be also dropped, which is equivalent to feature subsampling.

Standard dropout is only applied during training, using the same noise sample during the forward pass and the backpropagation backward pass. During testing, outgoing weights of layers which use dropout are multiplied by a constant ensuring the expected value of the outputs remains unchanged between training and testing.

1.4 Convolutional neural network architectures for image classification

Architecture of a neural network refers to the arrangement, types and sizes of layers of the network. The architecture of a neural network can be viewed as a subset of hyperparameters.

Typical CNNs for image classification follow a common pattern of multiple convolutional layers interleaved with downsampling layers, which extract increasingly complex features from an input image. This feature extractor component is then followed by a flattening operation leading into a classifier consisting of one or more fully-connected layers, with the output layer typically using softmax activation (Equation (1.1)).

In practice, a convolutional layer has multiple kernels, with each kernel producing an output feature map. Each kernel is 3-dimensional (generally $(n+1)$-dimensional for n-dimensional inputs), containing a separate weight matrix for each input channel. Therefore, a 2D convolution layer with kernels of width W, height H, with I input feature maps and O output feature maps has $(W * H * C + 1) * O$ parameters, where the additive term 1 is to count biases.

1.4.1 Downsampling

Downsampling is used to progressively reduce spatial dimension of the image, which helps increase the *receptive field* (the region of input pixels which affects the output of a neuron) of each neuron and reduce computational complexity. Also, as more complex features are extracted, their exact position becomes less important – the location relative to other features is sufficient.

The two main methods of downsampling are strided convolutions and *pooling layers* – most commonly *max pooling*, which partitions an input image into non-overlapping rectangular sub-regions and outputs the maximum value for each. Another common pooling layer is the average pooling layer, which as the name suggests, output the average value for each sub-region. Pooling layers most com-

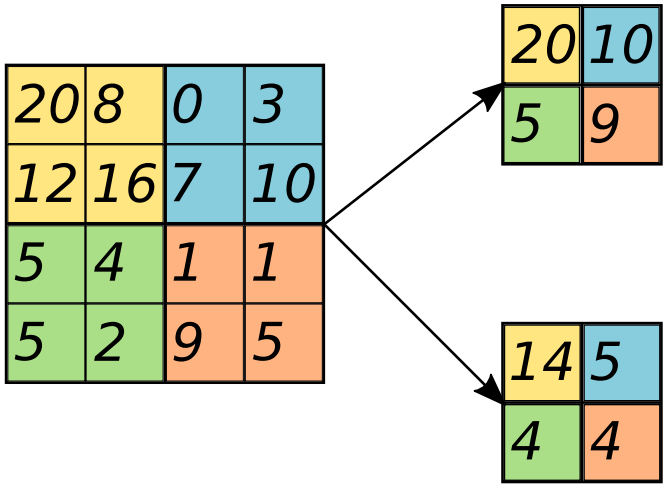


Figure 1.4: Pooling operations with 2×2 sub-regions. Top: max pooling. Bottom: average pooling.

monly use a dimension of 2×2, halving both spatial dimensions. Figure 1.4 illustrates 2×2 max pooling and average pooling.

The number of filters of convolutional layer typically increases after each pooling layer. A common scheme for increasing is doubling the number of filters after each layer, although there are many examples where a different scheme is used [32].

As more computational power becomes available and more advanced techniques for training large networks are invented, the number of layers in state-of-the-art CNNs tends to increase. To demonstrate this, let us consider the winning entry of the ImageNet Large Scale Visual Recognition Challenge [33], an annual competition in classification using a very large set of images into 1000 non-overlapping classes. The first time the winning entry used CNNs was in 2012 [34], used 8 weight layers. The winner in 2014 [35] used 22 layers and the 2015 winner used 152 layers [2]. The total number of parameters of state-of-the-art models can reach orders of tens of millions to hundreds of millions [36].

1.4.2 Flattening

Flattening refers to an operation transforming a multi-dimensional (3-dimensional $W \times H \times C$ for 2D images) set of feature maps into a flat (1-dimensional) vector, which becomes the input vector of the (MLP) classifier in a CNN.

The basic flattening operation simply takes this feature map and serializes it into a vector of length $W*H*C$. The problem with this approach is that $W*H*C$ can be a very large number, which results in a huge number of features arriving into the classifier. A method called *global pooling* has been more successful in deep models. Global pooling is a pooling operation as described above, where the whole image is treated as a single region. Global average pooling (GAP) [37] has recently been the most widely used flattening method, while the number of following fully-connected layers in the classifier has been reduced to just the output softmax layer [2, 35, 38, 39].

1.4.3 Batch normalization

Since the technique called *batch normalization (BN)* was introduced in 2015 [40], it has been extensively used in convolutional neural networks. Batch normalization dramatically increases training speed of deep CNNs by computing sample means and variances across a processed batch and normalizing them to fixed values, which are trainable parameters. Let $\boldsymbol{H}$ be a matrix of activations for a mini-batch of M examples, with the activations for each example arranged in rows of the matrix. Batch normalization replaces replaces $\boldsymbol{H}$ with

$$\boldsymbol{H'} = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}},$$

where the matrix-vector arithmetic is element-wise in each row, i.e. $H'_{ij} = \frac{H_{ij} - \mu_j}{\sigma_j}$ and

$$\boldsymbol{\mu} = \frac{1}{M} \sum_i \boldsymbol{H}_i$$

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{M} \sum_i (\boldsymbol{H} - \boldsymbol{\mu})_i^2},$$

where δ is a small constant for numeric stability. The output is then rescaled by the trainable parameters of standard deviation $\boldsymbol{\gamma}$ and mean $\boldsymbol{\beta}$ to $\boldsymbol{H''} = \boldsymbol{\gamma}\boldsymbol{H'} + \boldsymbol{\beta}$. Crucially, all operations leading from the input $\boldsymbol{H}$ to the final output $\boldsymbol{H''}$ are differentiable and we backpropagate through them.

Batch normalization greatly speeds up training of deep models by stabilizing the distributions of layer inputs, which makes the loss landscape smoother [41]. In addition, using noisy sample means and variances introduces small random noise in training, which has a slight regularization effect. Batch normalization is typically used after each layer before the activation function is applied.

The bias terms are usually omitted in batch-normalized layers because the mean parameter $\boldsymbol{\beta}$ of the batch normalization layer supplies the same effect.

2. Bayesian neural networks

In this section, we introduce one of the main focuses of this thesis - Bayesian neural networks, which augment previously described neural networks with *Bayesian statistics*.

First, we review the fundamentals of Bayesian statistics. Next, we describe approximate methods of performing Bayesian statistics and extend maximum likelihood estimation and interpret it in the framework of Bayesian statistics. Finally, we present three methods for approximate Bayesian used for neural networks and specifics of these methods when used in convolutional neural networks.

2.1 Bayesian statistics

Bayesian statistics is a sub-field of statistics, where probability represents a *degree of belief* in an event, as opposed to *frequentist statistics*, where probability is seen as the limit of relative frequency of an event as the number of trials approaches infinity.

2.1.1 Bayesian inference

Our main focus within Bayesian statistics, called *Bayesian inference*, is used to update a degree of belief in an event or a *hypothesis* using the central theorem of Bayesian statistics – *Bayes' theorem*:

$$P(H \mid D) = \frac{P(D \mid H)P(H)}{P(D)},$$

where

- H stands for any hypothesis, which may be influenced by observing data.
- $P(H)$ is referred to as a prior probability or simply *prior*, representing a degree of belief in the hypothesis before observing any data.
- $P(H \mid D)$ is the *posterior* distribution, which is a degree of belief in a hypothesis after observing data D.
- $P(D \mid H)$ is a *likelihood* function, which represents a probability of observing particular data under hypothesis H. It can be interpreted as compatibility of hypothesis H with observed data.
- $P(D)$ is referred to as *marginal likelihood*, which is not a function of H and can therefore be treated as a normalizing parameter when determining relative probabilities of hypotheses.

A typical task where Bayesian inference is used, which is our case as well, is to infer a posterior probability distribution of a parametric model after observing a set of (training) data.

The more commonly used alternative to Bayesian inference is frequentist inference, which maximum likelihood estimation (see Section 1.3.1) is an example

of. The main distinction between Bayesian inference compared and frequentist inference is that using Bayesian inference, a full probability distribution over a set of parameters is inferred. Whereas frequentist inference treats model parameters as scalar values by finding a point *point estimate* of the most likely set of parameters, Bayesian inference treats model parameters as random variables, inferring a joint distribution over model parameters.

To obtain a prediction or *posterior predictive distribution* of a Bayesian model, we need to integrate over the inferred distributions of model parameters:

$$p(\hat{y} \mid \boldsymbol{x}; \mathbb{X}, \boldsymbol{\alpha}) = \int p(\hat{y} \mid \boldsymbol{x}; \boldsymbol{w}) p(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})\, d\boldsymbol{w} = \mathbb{E}_{\boldsymbol{w} \sim p(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})} \left[p(\hat{y} \mid \boldsymbol{x}; \boldsymbol{w}) \right] \quad (2.1)$$

where

- $\boldsymbol{w}$ is a set of parameters of a Bayesian model, e.g., weights of a neural network,
- $\mathbb{X} = \{(\boldsymbol{x}, y)\}$ is a set of observed training data, which were used to infer the model parameters,
- $\boldsymbol{\alpha}$ is a vector of hyperparameters of the prior distribution over parameters $p(\boldsymbol{w}) = p(\boldsymbol{w} \mid \boldsymbol{\alpha})$. We will sometimes omit conditioning on the hyperparameters for the sake of brevity.
- $\boldsymbol{x}$ is a feature vector of an example, for which we wish to obtain a prediction,
- $\hat{y}$ is a prediction of the model.

As we can see in Equation (2.1), we need to perform integration over the model parameters to obtain a prediction, for which there is often no closed-form solution. Numerical integration of the posterior can be computationally very expensive or even intractable for many models, such as neural networks.

A typical way to approximate the integral is to take T samples of concrete parameter values from the posterior parameter distribution $\boldsymbol{w}^{(i)} \sim p(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})$ and use each sample for prediction, treating the sampled values as parameters of a frequentist model. The individual samples are then averaged to obtain an unbiased estimate of the true predictive posterior:

$$p(\hat{y} \mid \boldsymbol{x}; \mathbb{X}, \boldsymbol{\alpha}) \approx \frac{1}{T} \sum_{i=1}^{T} p(\hat{y} \mid \boldsymbol{x}; \boldsymbol{w}^{(i)}) \,.$$

A Bayesian model can also be interpreted as an ensemble of infinitely many models weighed according the posterior probability.

2.1.2 Approximate Bayesian inference

Since the true Bayesian posterior distribution can be represented by any probability density function, inferring the true posterior requires optimization over a space of functions. Exact Bayesian inference thus becomes intractable quickly as model complexity increases, and therefore needs to be approximated.

Maximum a posteriori estimation

The roughest commonly used approximate Bayesian estimator is the *maximum a posteriori (MAP) estimator*, which approximates the true distribution using a point estimate at the *mode* (the point with the maximum probability density) of the posterior distribution. Note that in the following derivation, we use notation introduced in Section 1.3.1:

$$\begin{aligned}
\boldsymbol{\theta}^*_{MAP} &= \arg\max_{\boldsymbol{w}} p(\boldsymbol{w} \mid \mathbb{X}) \\
&= \arg\max_{\boldsymbol{w}} p(\mathbb{X}; \boldsymbol{w}) p(\boldsymbol{w}) \\
&= \arg\max_{\boldsymbol{w}} p(\boldsymbol{w}) \prod_{i=1}^{N} p_{\text{model}}(y_i \mid \boldsymbol{x}_i; \boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{w}} -\sum_{i=1}^{N} \log p_{\text{model}}(y_i \mid \boldsymbol{x}_i; \boldsymbol{w}) - \log p(\boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{x}} -\mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{\text{data}}}[\log p_{\text{model}}(y \mid \boldsymbol{x}; \boldsymbol{w})] - \log p(\boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{w}} H(\hat{p}_{\text{data}}, p_{\text{model}}(. \mid .; \boldsymbol{w})) - \log p(\boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{w}} D_{\text{KL}}(\hat{p}_{\text{data}} \,\|\, p_{\text{model}}(. \mid .; \boldsymbol{w})) - \log p(\boldsymbol{w})
\end{aligned}$$

Maximum likelihood estimation (see Section 1.3.1) can be interpreted in the framework of Bayesian statistics as a special case of MAP estimation, where the prior distribution over parameters $p(\boldsymbol{w})$ is uniform.

Regularization as MAP estimation

In Section 1.3.3, we introduced norm regularization techniques from the frequentist point of view, where the loss function of models trained by maximum likelihood is modified by addition of a penalty term, which penalizes the model according the norm of the weights. These techniques can be reformulated and justified in the framework of Bayesian statistics as using MAP estimation with suitable priors, for example:

- L_2 regularization is achieved using a 0-mean Gaussian prior

  $$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \mu = 0, \sigma^2)\,,$$

  where lower variance σ^2 of the prior results in a stronger regularization effect, i.e. regularization factor $\lambda \propto \frac{1}{\sigma^2}$,

- L_1 regularization corresponds to a 0-mean Laplacean prior

  $$p(\boldsymbol{w}) = \text{Laplace}(\boldsymbol{w}; \mu = 0, b) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}$$

  where a lower scale factor b of the prior results in stronger regularization – regularization factor $\lambda \propto \frac{1}{b^2}$.

Because the probability density of Laplacean distribution is much higher around 0 for a given variance, L_1 regularization promotes parameter values of exactly 0 (*sparse solutions*), while a Gaussian prior merely promotes low absolute values of parameters.

2.2 Variational inference

Variational inference is another approximate Bayesian method, which can provide a much closer approximation of the true Bayesian posterior distribution. We provide a brief description, while focusing on methods which are applicable to neural networks. A more detailed description can be found in a review paper by Blei et al. [42] or a book by Bishop [43].

Variational inference aims to approximate the unrestricted general true posterior distribution $P(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})$, which is often intractable, by a *variational distribution* $Q(\boldsymbol{w})$ from a restricted set $\mathcal{Q}$ of tractable distributions and finding the distribution closest to the true posterior from this set.

The closeness of $Q(\boldsymbol{w})$ to $P(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})$ is usually measured in terms of KL divergence:

$$D_{KL}(Q(\boldsymbol{w}) \,\|\, P(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})) \to \min_{Q \in \mathcal{Q}} .$$

Note that in the above formula, the arguments are reversed compared to the intuitive order, where the true posterior would be the first term representing the reference distribution. In that case, however, we would need to compute an expectation over the true posterior, which is assumed to be intractable.

We already saw an example of variational inference in the maximum a posteriori estimator, which is equivalent to variational inference with a *delta posterior*, which assigns probability 1 to a particular set of weights (infinite probability density at the given point) and probability 0 to all other weights.

2.2.1 Parametric variational inference

Parametric variational distribution restricts the variational posterior $Q(\boldsymbol{w} \mid \boldsymbol{\theta})$ to a family of parametric distributions over weights $\boldsymbol{w}$ parameterized by a set of parameters $\boldsymbol{\theta}$.

Parameters $\boldsymbol{\theta}$ of are inferred by minimizing the KL divergence between the approximate variational distribution $Q(\boldsymbol{w} \mid \boldsymbol{\theta})$ and the true posterior $P(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})$:

$$\begin{aligned}
\boldsymbol{\theta}^*_{VI} &= \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}(Q(\boldsymbol{w} \mid \boldsymbol{\theta}) \,\|\, P(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha})) && (2.2)\\
&= \arg\min_{\boldsymbol{\theta}} \int Q(\boldsymbol{w} \mid \boldsymbol{\theta}) \log \frac{Q(\boldsymbol{w} \mid \boldsymbol{\theta})}{P(\boldsymbol{w} \mid \boldsymbol{\alpha}) P(\mathbb{X} \mid \boldsymbol{w})} d\boldsymbol{w} + \int Q(\boldsymbol{w} \mid \boldsymbol{\theta}) \log P(\mathbb{X}) d\boldsymbol{w} && (2.3)\\
&= \arg\min_{\boldsymbol{\theta}} \int Q(\boldsymbol{w} \mid \boldsymbol{\theta}) \log \frac{Q(\boldsymbol{w} \mid \boldsymbol{\theta})}{P(\boldsymbol{w} \mid \boldsymbol{\alpha})} d\boldsymbol{w} - \int Q(\boldsymbol{w} \mid \boldsymbol{\theta}) \log P(\mathbb{X} \mid \boldsymbol{w}) d\boldsymbol{w} && (2.4)\\
&= \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}(Q(\boldsymbol{w}) \,\|\, P(\boldsymbol{w} \mid \boldsymbol{\alpha})) - \mathbb{E}_{\boldsymbol{w} \sim Q(\boldsymbol{w} \mid \boldsymbol{\theta})} \left[\log P(\mathbb{X} | \boldsymbol{w})\right] . && (2.5)
\end{aligned}$$

Note that in Equation (2.5) we expanded the first integral from the previous equality and left out the second integral from the first equation because

$$\int Q(\boldsymbol{w} \mid \boldsymbol{\theta}) \log P(\mathbb{X}) d\boldsymbol{w} = \log P(\mathbb{X}) \int Q(\boldsymbol{w} \mid \boldsymbol{\theta}) d\boldsymbol{w} = \log P(\mathbb{X}) ,$$

which does not depend on $\boldsymbol{\theta}$.

The above derivation yields a loss function, which we refer to as *evidence lower bound* or *ELBO* from here on. Other names of this loss function used in

literature include *variational lower bound* or *variational free energy*. The ELBO loss is defined as:

$$\mathcal{L}_{\text{ELBO}} = -\sum_{(\boldsymbol{x},y)\in\mathbb{X}} \Big(\mathbb{E}_{\boldsymbol{w}\sim Q(\boldsymbol{w}|\boldsymbol{\theta})}\left[\log P(y \mid \boldsymbol{x};\boldsymbol{w})\right]\Big) + D_{\text{KL}}(Q(\boldsymbol{w}) \,\|\, P(\boldsymbol{w} \mid \boldsymbol{\alpha}))\,. \quad (2.6)$$

The name "evidence lower bound" originates from the equality

$$\log P(\mathbb{X}) = \mathcal{L}_{\text{ELBO}} - D_{\text{KL}}(Q(\boldsymbol{w} \mid \boldsymbol{\theta}) \,\|\, P(\boldsymbol{w} \mid \mathbb{X}, \boldsymbol{\alpha}))$$

where the KL divergence term is non-negative, which implies that $\mathcal{L}_{\text{ELBO}}$ is a lower bound the marginal evidence log-likelihood $\log P(\mathbb{X})$.

As we can see, the ELBO loss in Equation 2.6 consists of a sum of two terms:

1. The first term is a data-dependent negative log-likelihood, which is a loss function resulting from maximum likelihood estimation (Section 1.3.1), with a slight modification: instead of using a point estimate of the weights, an expectation over the variational posterior is used. Maximizing this term encourages the variational distribution to fit the observed data well by choosing parameters $\boldsymbol{\theta}$ which increase probability density $q(\boldsymbol{w} \mid \boldsymbol{\theta})$ at the mode of the likelihood function. We will refer to this term as expected log-likelihood and denote it as $\mathcal{L}_D$.

2. The second part is a prior-dependent term, which acts as regularization, encouraging parameters to remain close to the prior. A prior is usually chosen such it promotes model simplicity, such as the Gaussian or the Laplacean priors mentioned in section 2.1.2. We will denote this term as $\mathcal{L}_C$.

Variational inference provides us with a way to perform approximate Bayesian inference by optimizing a loss function. In other words, it casts inference as an optimization problem. In cases where the derived ELBO loss function is tractable, it can be directly optimized for example by gradient descent or other optimization techniques. However, it might happen that it is still intractable, in which case it needs to be further approximated, which we will describe in the following sections.

2.3 Bayesian neural networks

Variational inference and particularly parametric variational inference have been successfully applied to infer distributions over weights of neural network. The foundations were laid by Hinton and van Camp [44], who interpret the regularization term in Equation (2.6) term as as a term minimizing the amount of information required to encode weights of the neural network.

A further major contribution was made by Graves [45], who introduced a stochastic variational gradient-based optimization method, which fits well into the general framework of neural networks optimized by back-propagation.

A common choice of variational posterior used in Bayesian neural networks is an independent (fully factorized) Gaussian distribution $Q(\boldsymbol{w} \mid \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}; \boldsymbol{\sigma}^2)$ with parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\sigma}\}$ of the same length as the number of weights of the neural network [45, 44, 46, 47].

Because of the highly complex functional form of practical neural networks, the predictive posterior is intractable. As a result, the expectation over the

posterior distribution in ELBO loss cannot be evaluated analytically, which means that the ELBO loss function cannot be evaluated analytically, and so cannot the gradients of the loss function with respect to the parameters of the variational distribution. Therefore, we need a need a differentiable approximation of the predictive posterior in order to train the model and use it in prediction.

2.3.1 Reparameterization trick

The naïve Monte Carlo estimator of the expected gradient of a function of weights w.r.t. the variational parameters is given by the following equation:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{w}\sim q(\boldsymbol{w}|\boldsymbol{\theta})} \left[f(\boldsymbol{w})\right] &= \mathbb{E}_{\boldsymbol{w}\sim q(\boldsymbol{w}|\boldsymbol{\theta})} \left[f(\boldsymbol{w}) \nabla_{q(\boldsymbol{w}|\boldsymbol{\theta})} \log q(\boldsymbol{w} \mid \boldsymbol{\theta})\right] \\ &\approx \frac{1}{T} \sum_{i=1}^{T} f(\boldsymbol{w}^{(i)}) \nabla_{q(\boldsymbol{w}^{(i)}|\boldsymbol{\theta})} \log q(\boldsymbol{w}^{(i)} \mid \boldsymbol{\theta}) \,.\end{aligned}$$

Although this estimator is unbiased, it exhibits high variance, which hurts optimization. Kingma and Welling [48] propose a reparameterization reducing the variance of the estimator, which was originally used in the stochastic latent representation layer in variational autoencoders. The same principle can be applied at a larger scale to the stochastic weights. The proposed reparameterization of the random variable $\boldsymbol{w} \sim q(\boldsymbol{w} \mid \boldsymbol{\theta})$ uses a differentiable deterministic function t of an auxiliary random noise variable $\boldsymbol{\epsilon} \sim r(\boldsymbol{\epsilon})$ drawn from a simple noise distribution r: $\boldsymbol{w} = t(\boldsymbol{\epsilon}, \boldsymbol{\theta})$. The function t transforms a sample of parameter-free auxiliary noise and the variational parameters into a sample of weights from the variational posterior. This allows us to estimate gradients of an expectation over the variational posterior more efficiently:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{w}\sim q(\boldsymbol{w}|\boldsymbol{\theta})} \left[f(\boldsymbol{w})\right] &= \mathbb{E}_{\boldsymbol{\epsilon}\sim r(\boldsymbol{\epsilon})} \left[\nabla_{\boldsymbol{\theta}} f(t(\boldsymbol{\epsilon}, \boldsymbol{\theta}))\right] \\ &\approx \frac{1}{T} \sum_{i=1}^{T} \nabla_{\boldsymbol{\theta}} f(t(\boldsymbol{\epsilon}^{(i)}, \boldsymbol{\theta})) \,.\end{aligned}$$

Applying the above to the expected log-likelihood term of the ELBO loss yields an unbiased and differentiable Monte Carlo estimate of the gradient, referred to by the authors as *SGVB (stochastic gradient variational Bayes)*:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{ELBO}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}_D + \nabla_{\boldsymbol{\theta}} \mathcal{L}_C$$

where

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{D}} \approx \frac{1}{T} \sum_{i=1}^{T} \sum_{(\boldsymbol{x},y)\in\mathbb{X}} \nabla_{\boldsymbol{\theta}} \log p(y \mid \boldsymbol{x}; \boldsymbol{w} = t(\boldsymbol{\epsilon}^{(i)}, \boldsymbol{\theta})) = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{SGVB}} \,.$$

2.3.2 Bayes by Backprop

The reparameterization trick was in an algorithm called *Bayes by Backprop* [46], which uses Monte Carlo estimates of gradients to develop an algorithm similar to normal mini-batch gradient descent with backpropagation for parametric variational inference in fully-connected feed-forward neural networks. Bayes by backprop was described using neural networks with an independent Gaussian weight

posterior, which can be simply reparameterized using $t(\boldsymbol{\epsilon}, \boldsymbol{\mu}, \boldsymbol{\sigma}) = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, where values in $\boldsymbol{\epsilon}$ are independently sampled from the standard normal distribution $\mathcal{N}(0, 1)$. The standard deviation σ of the posterior is parameterized pointwise as $\sigma = \log(1 + \exp(\rho)) = \text{softplus}(\rho)$, ensuring that the standard deviation always remains non-negative. The variational parameters are $\boldsymbol{\theta} = \{\boldsymbol{\sigma}, \boldsymbol{\rho}\}$. Another proposed parameterization with the desired property of non-negativity of σ is $\sigma = \exp(\rho)$ [47].

The intractable exact ELBO loss is approximated by taking T samples of concrete weights and averaging the loss values computed for each particular draw:

$$\mathcal{L}_{ELBO} = \mathcal{L}(\mathbb{X}, \boldsymbol{\theta}) \approx \frac{1}{T} \sum_{i=1}^{T} \log q(\boldsymbol{w}^{(i)} \mid \boldsymbol{\theta}) - \log P(\boldsymbol{w}^{(i)}) - \log P(\mathbb{X} \mid \boldsymbol{w}^{(i)}) \,, \quad (2.7)$$

where the first two terms approximate the KL divergence regularization term $\mathcal{L}_C$. The authors claim that this approximation of the KL divergence performs no worse in practice than the closed-form solution even in cases where the closed form exists (for example between a pair Gaussians). A clear advantage is that it allows the use of a prior, for which there is no simple closed-form solution of KL divergence between the prior and the Gaussian posterior. In particular, the authors propose a prior consisting of a scale mixture of two zero-mean diagonal Gaussians:

$$P(\boldsymbol{w} \mid \sigma_1, \sigma_2, \pi) = \prod_j \gamma \mathcal{N}(w_j \mid 0, \sigma_1^2) + (1 - \gamma) \mathcal{N}(w_j \mid 0, \sigma_2^2) \,,$$

where $\sigma_1 > \sigma_2$ and $\sigma_2 \ll 1$ and $0 < \gamma < 1$, with the first term providing a heavier tail and the second term providing a tighter a priori concentration of weights around 0. There is no known closed-form solution to KL divergence between a Gaussian and a scale mixture of Gaussians [49]. Experiments performed by the authors show an improvement in classification when using a scale mixture prior with grid-optimized hyperparameters compared to a Gaussian prior.

Gradients of this loss w.r.t the variational parameters can be computed using a backpropagation-like algorithm. One epoch of the optimization is sketched in Algorithm 3.

Note that the term $\frac{\partial f(\boldsymbol{w}, \boldsymbol{\theta})}{\partial \boldsymbol{w}}$, which is present in gradients of both mean and variance parameters, is the gradient calculated by standard backpropagation for the particular draw of weights. Thus, we can learn the variational parameters by simply calculating the usual backpropagation gradients, and then scaling and shifting them as shown.

They also propose a reweighing of the KL divergence regularization term in the ELBO loss suitable for use with mini-batch gradient descent. Let us consider splitting the training data into M batches and let $\mathbb{X}_i$ denote the i-th batch. Also, let $\boldsymbol{\pi} \in [0, 1]^M$ and $\sum_{i=1}^{M} \pi_i = 1$. The general scheme is as follows:

$$\mathcal{L}(\mathbb{X}_i, \boldsymbol{\theta}) = \pi_i D_{\text{KL}}(q(\boldsymbol{w} \mid \boldsymbol{\theta}) \,\|\, P(\boldsymbol{w} \mid \boldsymbol{\alpha})) - \mathbb{E}_{q(\boldsymbol{w}|\boldsymbol{\theta})} \left[\log P(\mathbb{X}_i \mid \boldsymbol{w})\right] \,,$$

which is equivalent to the ELBO loss because $\sum_i \mathcal{L}(\mathbb{X}_i, \boldsymbol{\theta}) = \mathcal{L}_{\text{ELBO}}$. The standard scheme for weighing the KL divergence term would distribute it evenly between batches in an epoch by setting each $\pi_i = \frac{1}{M}$. Their proposed scheme, which they

Algorithm 3: Bayes by Backprop

Data: Training data $\mathbb{X}$, variational parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\rho}\}$

1 Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, 1)$

2 Let $\boldsymbol{w} = \boldsymbol{\mu} + \text{softplus}(\boldsymbol{\rho}) \odot \boldsymbol{\epsilon}$

3 Let $f(\boldsymbol{w}, \boldsymbol{\theta}) = \log q(\boldsymbol{w} \mid \boldsymbol{\theta}) - \log P(\boldsymbol{w}) - \log P(\mathbb{X} \mid \boldsymbol{w})$

4 Calculate the gradient with respect to the mean

$$\Delta_{\mu} \leftarrow \frac{\partial f(\boldsymbol{w}, \boldsymbol{\theta})}{\partial \boldsymbol{w}} + \frac{\partial f(\boldsymbol{w}, \boldsymbol{\theta})}{\partial \boldsymbol{\mu}}$$

5 Calculate the gradient with respect to the variance parameters $\boldsymbol{\rho}$

$$\Delta_{\rho} \leftarrow \frac{\partial f(\boldsymbol{w}, \boldsymbol{\theta})}{\partial \boldsymbol{w}} \frac{\boldsymbol{\epsilon}}{1 + \exp(-\boldsymbol{\rho})} + \frac{\partial f(\boldsymbol{w}, \boldsymbol{\theta})}{\partial \boldsymbol{\rho}}$$

6 Update the variational parameters

$$\boldsymbol{\mu} \leftarrow \boldsymbol{\mu} - \alpha \Delta_{\mu}$$
$$\boldsymbol{\rho} \leftarrow \boldsymbol{\rho} - \alpha \Delta_{\rho}$$

empirically found to work well, is $\pi_i = \frac{2^{M-i}}{2^M - 1}$. This scheme assigns high influence of the complexity cost to the first few mini-batches, while the cost in the last few mini-batches is highly data-dependent. According to the authors, this property is especially useful in the beginning of training, where changes to weights due to the data are slight in the first few mini-batches, only becoming significant as more data is seen.

The main disadvantage of Bayes by Backprop is that it doubles the number of parameters compared to an equivalent point-estimate neural network without increasing the model capacity, which increases computation times and memory demands. Additionally, obtaining a prediction in practice involves performing averaging results of multiple forward passes with different samples from the variational posterior, which drastically increases the prediction time compared to an equivalent frequentist model.

On the other hand, the Bayesian approach provides a strong regularization effect and an effective way of estimating and decomposing model uncertainty. Additionally, the authors demonstrate that by pruning weights minimizing the signal-to-noise ratio $|\mu_i|/\sigma_i$, it is possible to achieve a pruning ratio of up to 95% with minimal deterioration of prediction performance.

2.3.3 Local reparameterization trick

Next, we describe a method introduced in 2015 by Kingma et al. [31] called the *local reparameterization trick (LRT)*. The local reparameterization is an extension of the reparameterization trick introduced in Section 2.3.1. LRT aims to speed up training of models with parametric stochastic weights by reducing variance of the Monte Carlo gradients and also decrease computational cost. It can be applied to various types of neural networks with stochastic weights, such as Bayes by

Backprop, or others, which we introduce further in this chapter.

Consider a dataset $\mathbb{X}$ with N examples and and let us estimate the data-dependent part $\mathcal{L}_D$ of the ELBO loss using a single mini batch of size M and a single sample of the posterior weights, auxiliary noise:

$$\mathcal{L}_D \approx \mathcal{L}_{\text{SGVB}} = \frac{N}{M} \sum_{i=1}^{M} \log p(y_i \mid \boldsymbol{x}_i; \boldsymbol{w}^{(i)} = t(\boldsymbol{\epsilon}^{(i)}, \boldsymbol{\theta})) = \frac{N}{M} \sum_{i=1}^{M} L_i \,.$$

Variance of the estimator $\mathcal{L}_{\text{SGVB}}$ is given by:

$$\text{Var}\left[\mathcal{L}_{\text{SGVB}}\right] = \frac{N^2}{M^2} \left(\sum_{i=1}^{M} \text{Var}\left[L_i\right] + 2 \sum_{i=1}^{M} \sum_{j=i+1}^{M} \text{Cov}\left[L_i, L_j\right] \right) \tag{2.8}$$

$$= N^2 \left(\frac{1}{M} \text{Var}\left[L_i\right] + \frac{M-1}{M} \text{Cov}[L_i, L_j] \right) , \tag{2.9}$$

where the variances and covariances are w.r.t both the empirical training data distribution and the auxiliary noise distribution r:

$$\text{Var}[L_i] = \text{Var}_{\boldsymbol{\epsilon}^{(i)} \sim r(\boldsymbol{\epsilon}), (\boldsymbol{x}_i, y_i) \sim \mathbb{X}} \left[\log p(y_i \mid \boldsymbol{x}_i; \boldsymbol{w}^{(i)} = t(\boldsymbol{\epsilon}^{(i)}, \boldsymbol{\theta})) \right] .$$

As we can see in Equation (2.9), the contribution of the variance $\text{Var}\left[L_i\right]$ to the total variance decreases with the batch size M. However, the contribution of the covariances does not decrease with M. According to the authors, the covariance term can become dominant even for moderately large batch sizes, negating the variance-reducing property of using larger batch sizes, one of the main motivations behind mini-batch gradient descent.

In order for the variance to scale as $1/M$, which is the case with frequentist inference, the authors propose a different sampling method, which ensures that $\text{Cov}\left[L_i, L_j\right] = 0$. A naïve way of achieving this property is sampling a separate weight matrix for each example in a mini batch. However, this approach in practice requires sampling of a very large random tensor, proportional in size to the number of weights multiplied by the batch size. Sampling such a large random vector requires a non-trivial amount of time and introduces a large amount of noise in the training. Additionally, layer operations would need to be performed with a different set of weights for each example in a batch, negating the computational efficiency of batch computation.

The sampling method proposed by Kingma et al. [31], called the *local reparameterization trick*, is based on the observation that the expected log-likelihood is only influenced by the weights (and the random noise) through the neuron activations, of which there are normally much fewer compared to the variational parameters.

Let us consider a single fully-connected layer with a stochastic weight matrix $\boldsymbol{W}$, an input batch represented by a matrix $\boldsymbol{A}$ and the stochastic batch activation represented by a matrix $\boldsymbol{B} = \boldsymbol{AW}$. For a fully-factorized Gaussian variational posterior over weights (such as in Bayes by Backprop), the posterior for the activations $\boldsymbol{B}$ conditioned on the input $\boldsymbol{A}$ is also a fully-factorized Gaussian:

$$q(w_{i,j} \mid \boldsymbol{\theta}) = \mathcal{N}(\mu_{i,j}, \sigma_{i,j}^2) \implies q(b_{m,j} \mid \boldsymbol{A}, \boldsymbol{\theta}) = \mathcal{N}(\gamma_{m,j}, \delta_{m,j}) \,,$$

where

$$\gamma_{m,j} = \sum_i a_{m,i} \mu_{i,j} \quad \text{and} \quad \delta_{m,j} = \sum_i a_{m,i}^2 \sigma_{i,j}^2 \,.$$

or in matrix notation:

$$\boldsymbol{\Gamma} = \boldsymbol{A}\boldsymbol{W}_\mu \qquad \text{and} \quad \boldsymbol{\Delta} = \boldsymbol{A}^2\boldsymbol{W}_\sigma^2\,,$$

where $\boldsymbol{W}_\mu$ denotes the matrix of weight means, $\boldsymbol{W}_\sigma$ denotes the matrix of standard deviations of individual weights and matrix powers are element-wise, i.e. $\boldsymbol{A}^2 = \boldsymbol{A} \odot \boldsymbol{A}$.

Thus, we can sample the random activations instead of sampling the random weights and using them to compute the activations:

$$b_{m,j} = \gamma_{m,j} + \sqrt{\delta_{m,j}}\epsilon_{m,j}$$

$$\boldsymbol{B} = \boldsymbol{\Gamma} + \sqrt{\boldsymbol{\Delta}} \odot \boldsymbol{\mathcal{E}}\,,$$

where $\epsilon_{m,j} \sim \mathcal{N}(0,1)$. The auxiliary random matrix $\boldsymbol{\mathcal{E}}$ for a batch of size M and a layer with O outputs has a dimension of $M \times O$. Note that the sampling procedure for the activations itself utilizes the reparameterization trick. Note that this is the reparameterization trick introduced in Section 2.3.1, not the LRT itself. In comparison, a single sample of the weight posterior requires sampling an auxiliary noise matrix of dimension $I \times O$, where I is the number of inputs, while forfeiting the advantage of variance inversely proportional to batch size. Using an independent weight sample for each example, which is unsuitable for batch computation, requires sampling a noise tensor of dimension $M \times I \times O$.

To get an intuitive sense of the variance-reducing properties of the local reparameterization trick, consider the estimated gradient with respect to the variational parameter $\sigma_{i,j}^2$ using batch size $M = 1$. Directly sampling the random weights $\boldsymbol{W}$, we get

$$\frac{\partial \mathcal{L}_{\text{SGVB}}}{\partial \sigma_{i,j}^2} = \frac{\partial \mathcal{L}_{\text{SGVB}}}{\partial b_j}\frac{\epsilon_{i,j}a_i}{2\sigma_{i,j}}\,.$$

Using the local reparameterization trick to sample the activations, we get

$$\frac{\partial \mathcal{L}_{\text{SGVB}}}{\partial \sigma_{i,j}^2} = \frac{\partial \mathcal{L}_{\text{SGVB}}}{\partial b_j}\frac{\epsilon_j a_i}{2\sqrt{\delta_j}}\,.$$

The latter gradient estimate depends on much fewer noise variables - only one stochastic variable rather than a noise variable for each element of the input. It is, therefore, much easier to estimate the influence of the back-propagated gradient since it depends on much fewer noisy variables. A more rigorous argument can be found in the original paper [31].

2.4 Dropout as a Bayesian approximation

In 2016, Gal and Ghahramani [50] interpreted dropout (described in Section 1.3.3), which was originally proposed as a regularization technique, within the Bayesian framework.

The ordinary weight matrices in neural networks are interpreted as variational parameters and the variational posterior is a distribution over the weight matrices with columns randomly set to zero, corresponding to setting the activation of a neuron to zero. The dropout rate remains fixed during training by setting of a hyperparameter.

Let us denote the number of neurons in the layer l for $l = 1, \ldots, L$ as K_l and the base (pre-dropout) matrix of connection weights from the $(l-1)$-st layer of the l-th layer as $\boldsymbol{M}_l \in \mathbb{R}^{K_{l-1} \times K_l}$. The (stochastic) weight matrix $\boldsymbol{W}_l$ of the l-th layer is parameterized as:

$$\boldsymbol{W}_l = \boldsymbol{M}_l \operatorname{diag}\left([\epsilon_{l,j}]_{j=1}^{K_l}\right)$$
$$\epsilon_{l,j} \sim \text{Bernoulli}(1 - p_l) \text{ for } l = 1, \ldots, L, j = 1, \ldots, K_{l-1}\,,$$

where p_l is the dropout rate of the l-th layer.

The approximate predictive posterior is obtained by Monte-Carlo integration over the weights, which is in practice performed by averaging results of multiple forward with different dropout noise samples (masks). The technique is known as *Monte-Carlo (MC) dropout*. The proposed method does not include any modifications to the standard frequentist loss function nor does it introduce any additional parameters or hyperparameters besides the number of samples used in MC dropout prediction.

A major advantage of this approach is that it can be applied to a wide range of existing models trained with dropout. Experiments conducted by Gal and Ghahramani [50] show slightly better results when evaluating models using Monte-Carlo dropout compared to using standard weights averaging. Using standard weight averaging, a single forward pass is performed during prediction, where outgoing weights of layers using dropout are multiplied by suitable constants depending on the dropout rate to maintain identical expected activations as during training. Although we describe the method specifically for binary dropout using Bernoulli multiplicative noise (following the original publication), it can also be applied to other dropout variants, such as Gaussian dropout.

2.4.1 Variational dropout

Variational dropout, proposed by Kingma et al. [31], makes the dropout rate of Gaussian dropout adaptive to the data as opposed to dropout where the dropout rate is a hyperparameter fixed during training.

Computing the batch output matrix $\boldsymbol{B}$ of a dropout layer (with any dropout noise distribution) corresponds to applying element-wise multiplication by an independently sampled noise matrix $\boldsymbol{\mathcal{E}}$ and the input $\boldsymbol{A}$ and multiplying the result by the weight matrix $\boldsymbol{W}$:

$$\boldsymbol{B} = (\boldsymbol{A} \odot \boldsymbol{\mathcal{E}})\boldsymbol{W}\,,$$

where $\epsilon_{i,j} \sim r(\epsilon_{i,j})$ is sampled from the dropout noise distribution r.

As shown by Wang and Manning [51], if the elements of the noise matrix are sampled independently from a Gaussian distribution $r = \mathcal{N}(1, \alpha)$, each element $b_{m,j}$ of the batch output activation matrix $\boldsymbol{B}$ follows a Gaussian distribution $\mathcal{N}(\gamma_{m,j}, \delta_{m,j})$ with

$$\boldsymbol{\Gamma} = \boldsymbol{A}\boldsymbol{W} \quad \text{and} \quad \boldsymbol{\Delta} = \alpha \boldsymbol{A}^2 \boldsymbol{W}^2\,.$$

The local reparameterization trick can be used to note that the distribution over activations is equivalent to using stochastic weights, where the posterior of each weight is independently sampled from a Gaussian:

$$q(w_{i,j} \mid \boldsymbol{\theta}) = \mathcal{N}(\theta_{i,j}, \alpha\theta_{i,j}^2)\,.$$

The variational parameters are weight means $\boldsymbol{\mu}$ (standard scalar weights, if we ignore variational dropout) and dropout rates $\boldsymbol{\alpha}$: $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\alpha}\}$. Training is performed by variational inference using the ELBO loss. Weight parameters $\boldsymbol{\theta}$ are trained using maximum likelihood estimation, i.e. by maximizing the expected log-likelihood $\mathbb{E}_{\boldsymbol{w}\sim q(\boldsymbol{w}|\boldsymbol{\theta})}\,\mathcal{L}_D$.

The prior is chosen so that the regularization term $D_{\mathrm{KL}}(q(\boldsymbol{w} \mid \boldsymbol{\theta}) \,\|\, p(\boldsymbol{w}))$ does not depend on the pre-dropout weight means $\boldsymbol{\mu}$. The authors show that the only prior which satisfies this condition is the scale invariant log-uniform prior: $p(\log|w_{i,j}|) \propto c$, where c is a constant. This is an improper prior, meaning that there exists no constant c for which p is a valid probability density function. KL divergence between the log-uniform prior and the posterior cannot be analytically computed; the authors derive an approximation, which they claim is extremely accurate [31]:

$$D_{\mathrm{KL}}(q(w_i \mid \boldsymbol{\theta}) \,\|\, p(w_i)) \approx c + 0.5\log(\alpha) + c_1\alpha + c_2\alpha^2 + c_3\alpha^3\,,$$

with

$$c_1 = 1.16145124, \quad c_2 = -1.50204118, \quad c_3 = 0.58629921\,.$$

Molchanov et al. [52] propose a different approximation, which provides a close approximation for a wider range of dropout rate values α.

The noise parameter α can be used at various levels of granularity. A separate dropout rate can be learned per layer, per neuron or even per weight [52].

Further, the authors set a constraint $\alpha \leq 1$, which corresponds to setting an upper bound on the posterior variance to be equal to the square of the posterior mean. This should help avoid local optima with high α, which are difficult to escape due to the resulting high variance of the gradients. A dropout rate of 1 results in noise with same variance as binary dropout with dropout rate 0.5, which maximizes the entropy of the multiplicative noise in case of binary dropout, which in turn maximizes the regularization effect.

2.5 Bayesian convolutional neural networks

2.5.1 Dropout as a Bayesian approximation in CNNs

Gal and Ghahramani, who proposed the Bayesian interpretation of dropout ([50], Section 2.4) in fully-connected feed-forward neural networks, applied a similar method to CNNs [53]. They show that their arguments posed for fully-connected layers with matrix multiplication can be applied to convolutional layers by showing that convolution is equivalent to multiplication by a matrix built by a particular arrangement of the values in the convolutional kernels and inputs.

They also experimentally demonstrate that classification performance of existing CNN models pre-trained with dropout can be improved by performing MC dropout at test time compared to the standard dropout output scaling. Improvement in classification performance is observed when applying MC dropout to convolutional layers, both compared to no dropout and standard dropout. The advantage of Monte-Carlo dropout increases as more outputs from more forward passes are averaged. The performance benefit over standard dropout becomes significant at around 20 Monte-Carlo samples, with little additional benefit observed beyond approximately 80 samples in their experiments.

The authors claim that standard dropout is usually not used after convolutional layers; this is no longer true as recent complex CNN architectures have benefited from adding dropout after convolutional layers [2, 54, 55]. Dropout rates used after convolutional layers are typically lower (such as 0.1) than dropout rates used after fully-connected layers (often up to 0.5).

2.5.2 Bayes by Backprop for CNNs

The algorithm Bayes by Backprop [46] (introduced in Section 2.3.2), originally presented for fully-connected feed-forward neural networks, was extended to recurrent neural networks in 2017 by Fortunato et al. [56] and to convolutional neural networks in 2018 by Laumann et al. [57, 58].

Their method for CNNs combines the ideas of Bayes by Backprop with an extension of the local reparameterization trick (described in Section 2.3.3) for CNNs. Namely, they follow the original Bayes by Backprop algorithm in the use of a Gaussian posterior over weights, the same estimate of the ELBO loss (Equation (2.7)) and a similar optimization procedure. One aspect where they diverge from the original

The variational posterior q over the kernel weights $w_{i,j,h,w}$ is parameterized as $q(w_{i,j,h,w} \mid \boldsymbol{\theta} = \{\boldsymbol{\mu}, \log \boldsymbol{\alpha}\}) = \mathcal{N}(\mu_{i,j,h,w}, \mu_{i,j,h,w}\alpha^2_{i,j,h,w})$. There are two marked differences from the original Bayes by Backprop: first, the variance of the variational posterior is relative to the mean, where the original Bayes by Backprop uses a variance independent of the mean, and second, the variance part of the Gaussian is obtained from the variational by exponentiation instead of using the softplus function.

As mentioned, Bayesian CNNs as proposed by Laumann et al., also utilize the idea of the local reparameterization trick, which is modified for use with convolutional layers. Sampling from the 4-dimensional tensor (for 2 spatial dimensions) of batch output activations $\boldsymbol{B}$ for a 4-dimensional tensor of batch inputs $\boldsymbol{A}$ is performed as follows:

$$\boldsymbol{B} = \boldsymbol{A} * \boldsymbol{W}_\mu + \boldsymbol{\mathcal{E}} \odot \sqrt{\boldsymbol{A}^2 * (\boldsymbol{W}_\alpha \odot \boldsymbol{W}_\mu^2)}\,,$$

where $*$ signifies the usual convolution operation a convolutional layer would perform on the input batch, $\boldsymbol{W}_\mu$ denotes the tensor of kernel means and $\boldsymbol{W}_\alpha$ signifies the tensor of kernel relative standard deviations. The dimensions of the tensor of auxiliary noise $\boldsymbol{\mathcal{E}}$ are the same as the dimensions of the output activations $\boldsymbol{B}$. Tensor powers are element-wise. Note that two full operations of a normal convolutional layer are performed per input image – one for the mean and one for the standard deviation – similarly to the local reparameterization trick for a fully-connected layer with Bayes by Backprop, where two matrix-vector multiplications are performed per example, i.e. two matrix-matrix multiplications per batch.

3. Active learning

Active learning is a machine learning technique where the *learner* can interactively query an information source, referred to as *oracle*, to label new data with correct outputs. The main motivation for active learning is that if a learner is able to select the most informative training examples to learn from, it should be able to learn more with less data.

Such a property would be especially useful in tasks, where unlabelled data is abundant, but labels are expensive to obtain. The focus of this thesis, image classification, is a good example of such a task. Images are usually readily available, but obtaining the normally required thousands or millions of labels requires considerable human effort.

The chapter is organized as follows. First, we describe active learning *scenarios*. An active learning scenario refers to the type of process used to deliver unlabelled candidate examples for labelling. Next, we present examples of commonly-used query strategy frameworks. A query strategy framework is a class of acquisition functions, which are functional criteria used for evaluating informativeness of unlabelled examples. Next, we describe *batch-mode active learning*, where multiple instances are acquired per iteration. Finally, we review applications of active learning to image classification tasks.

3.1 Scenarios

Active learning research deals with three main active learning scenarios:

- **Membership Query Synthesis** [59]. In the membership query synthesis scenario, the learner may request the oracle to label any instance in the input space. The learner is thus allowed to generate instances to label *de novo*. The main problem with implementing this scenario is that it may happen that many examples in the input space are out of domain as defined by the underlying data-generating distribution. For example, synthesizing an image in a particular domain is a difficult task itself and it often happens that most randomly generated images cannot be reasonably labelled [60].

- **Stream-based selective sampling** [61]. Stream-based selective sampling, also known as sequential active learning, uses unlabelled instances from the underlying distribution. It is assumed that obtaining an unlabelled instance from the underlying distribution is inexpensive. Unlabelled instances are sampled shown to the learner and the learner decides whether or not to ask for the label of each particular instance. If the learner decides not to label a presented instance, it is discarded and a new instance is presented in the next iteration.

- **Pool-based sampling** [62]. In the pool-based sampling scenario, we assume there is a small set of labelled data $\mathcal{L} = \{(\boldsymbol{x_i}, y_i)\}_i$ to learn from, and a larger set of unlabelled data $\mathcal{U} = \{\boldsymbol{x}_i\}_i$ to draw queries from. Once a label is obtained for an unlabelled sample, it is removed from the unlabelled set and added to the labelled set. The learner then learns from the extended

labelled set and iteratively queries more samples from the unlabelled set to be labelled. This process is continued until a desired predictive performance of the model is achieved or a labelling budget is exhausted. Algorithm 4 sketches pool-based active learning in the case where B best instances are queried in each iteration. Pool-based active learning has been successfully applied to various machine learning tasks, including image processing tasks [63, 64, 65, 66].

Algorithm 4: Pool-based batch-mode active learning

Data: Labelled data $\mathcal{L}$, unlabelled pool $\mathcal{U}$, query strategy $\phi(.)$, query batch size B

repeat
  $\boldsymbol{\theta} \leftarrow \text{train}(\mathcal{L})$
  for *b = 1 to B* **do**
    $\boldsymbol{x}_b^* \leftarrow \arg\max_{\boldsymbol{x} \in \mathcal{U}} \phi(\boldsymbol{x})$
    $y_b \leftarrow \text{label}(\boldsymbol{x}_b^*)$
    $\mathcal{L} \leftarrow \mathcal{L} \cup (\boldsymbol{x}_b^*, y_b)$
    $\mathcal{U} \leftarrow \mathcal{U} \setminus (\boldsymbol{x}_b^*, y_b)$
  end
until *stopping criterion*;

3.2 Query strategy frameworks

In all the active learning scenarios, the learner needs to be able to evaluate the informativeness of unlabelled instances. A wide variety of query strategies have been studied [67]. In this section, we present some common examples. We will denote the most informative instance according to a query selection algorithm A as $\boldsymbol{x}_A^*$.

3.2.1 Uncertainty sampling

Uncertainty sampling [62] is perhaps the most widely-used active learning query strategy framework. The learner queries the instances where the predictions are the most uncertain. To measure uncertainty in predictions, it is assumed the learner outputs a distribution over possible labellings. It is often straightforward to implement uncertainty sampling query strategies for probabilistic models.

Let $P(. \mid \boldsymbol{x})$ refer to the output categorical distribution of the learner and $P(y_c \mid \boldsymbol{x})$ refer to the predicted probability of labelling c from the set of possible labellings (classes) C. Examples of query strategies for multi-class classification problems include:

- **Least confident**:

$$\boldsymbol{x}_{\text{LC}}^* = \arg\max_{\boldsymbol{x} \in \mathcal{U}} 1 - P(\hat{y} \mid \boldsymbol{x}) \,.$$

  The least confident query strategy (also known as *variations ratios*) selects the sample where the class predicted by the probabilistic model is assigned the lowest probability. The value is equivalent to the sum of probabilities assigned to all the classes except the predicted class and can be interpreted as the probability of misclassification expected by the model.

- **Margin sampling**:

$$\boldsymbol{x}^*_{\mathrm{M}} = \arg\min_{\boldsymbol{x}\in\mathcal{U}} P(\hat{y}_1 \mid \boldsymbol{x}) - P(\hat{y}_2 \mid \boldsymbol{x})\,.$$

  In the above equation, $\hat{y}_1$ refers to the class with the highest probability and $\hat{y}_2$ refers to the class with the second highest probability. Whereas the least confident strategy ignores the rest of the model output distribution apart from the predicted class, the margin sampling strategy also takes into account the second most probable class. The rest of the distribution is still not taken into account. The motivation behind margin sampling is that instances where the difference between the first two probabilities is large are easy to classify, whereas those with a small margin are ambiguous.

- **Maximum entropy**:

$$\boldsymbol{x}^*_{\mathrm{H}} = \arg\max_{\boldsymbol{x}\in\mathcal{U}} = H[P(.\mid \boldsymbol{x}))] = -\sum_{c\in C} P(y_c \mid \boldsymbol{x}) \log P(y_c \mid \boldsymbol{x})\,.$$

  The maximum entropy takes an information-theoretic approach, using entropy of the output distribution as a measure of uncertainty. In the case of binary classification, entropy is equivalent to both least confident and margin sampling, querying the instance which has posterior probability closest to 0.5.

3.2.2 Query-by-committee

The query-by-committee uncertainty sampling framework [68] is based on maintaining a committee (also known as an ensemble) of n learners, which are all trained on the current labelled set $\mathcal{L}$, each representing a different competing hypothesis. Each member of the committee is allowed to vote on the labellings of query candidates. The instance where they most disagree is considered the most informative.

The main premise of query-by-committee is minimizing the set of hypotheses, which are consistent with the current labelled set $\mathcal{L}$.

There are two main approaches to measuring the amount of disagreement in a prediction by a committee:

- **Vote entropy** measures the entropy of the distribution consisting of predicted labellings by each member of the committee:

$$\boldsymbol{x}^*_{\mathrm{VE}} = \arg\max_{\boldsymbol{x}\in\mathcal{U}} -\sum_{c\in C} \frac{V(y_c)}{|C|} \log \frac{V(y_c)}{|C|}\,,$$

  where $V(y_c)$ is the number of votes that the label c receives, i.e. the number of committee members which assign the highest probability to class c. Note that vote entropy is equivalent to maximum entropy uncertainty sampling, where the output distribution of the committee is constructed by putting votes of committee members into bins and normalizing the result to produce a categorical distribution over possible labellings. Unlike uncertainty sampling, vote entropy can also be used with models which only produce a single class prediction instead of a full distribution.

- **Average KL-divergence** measures the average disagreement between predictions of individual committee members and the aggregate prediction of the committee:

$$\boldsymbol{x}^*_{\mathrm{KL}} = \arg\max_{\boldsymbol{x}\in\mathcal{U}} \frac{1}{n}\sum_{i=1}^{n} D_{\mathrm{KL}}(P_i(. \mid \boldsymbol{x}) \parallel P(. \mid \boldsymbol{x}))\,,$$

  where $P_i(. \mid \boldsymbol{x})$ is the output distribution of the i-th committee member consisting of predicted probabilities for each class and $P(y_c \mid \boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} P_i(y_c \mid \boldsymbol{x})$ for each class c is the average prediction of the whole committee. Kullback-Leibler (KL) divergence is an information-theoretic measure of difference of two distributions, described in Section 1.3.1.

3.2.3 Expected model change

The expected model change query strategy framework uses an approach based on decision theory, querying instances, which would most affect the current model, if it was provided the label for that instance. A typical query strategy in this framework is the expected gradient length (EGL) [69], which can be used for models trained by gradient-descent-like optimization algorithms. The learner queries the instance, which after addition to the labelled set produces the largest gradient of the loss function.

Let $\nabla_{\boldsymbol{\theta}} l(\mathcal{L};\boldsymbol{\theta})$ be the gradient of the loss function l with respect to the model parameters $\boldsymbol{\theta}$ for the current labelled set $\mathcal{L}$. Let $\nabla_{\boldsymbol{\theta}} l(\mathcal{L}\cup(\boldsymbol{x},y);\boldsymbol{\theta})$ be the gradient after the addition of a new instance $\boldsymbol{x}$ with label y to the labelled set. Since the correct label y is not known in advance, we take an expectation over the model posterior:

$$\boldsymbol{x}^*_{EGL} = \arg\max_{\boldsymbol{x}\in\mathcal{U}} \sum_{c} P(y_c \mid \boldsymbol{x}) \|\nabla_{\boldsymbol{\theta}} l(\mathcal{L}\cup(\boldsymbol{x},y_i);\boldsymbol{\theta})\|_2.$$

Note that at query time, $\|\nabla_{\boldsymbol{\theta}} l(\mathcal{L};\boldsymbol{\theta})\|$ should be very close to zero, since the model is assumed to have been trained to convergence. Thus, we can use an approximation $\nabla_{\boldsymbol{\theta}} l(\mathcal{L}\cup(\boldsymbol{x},y);\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} l((\boldsymbol{x},y);\boldsymbol{\theta})$.

EGL can be computationally expensive if both the feature space and the set of labellings are large. It is also sensitive to magnitude of features and parameters – the informativeness of an instance can be over-estimated if one of its features is large or a corresponding parameter has a large estimated value.

3.2.4 Density-weighed methods

Simple query strategy frameworks like uncertainty sampling, QBC and EGL have a disadvantage is that they query instances which are "controversial", which often correspond to outliers present in the data. Outliers do not necessarily provide the maximum amount of information about the underlying data distribution. Such a situation is illustrated in Figure 3.1. Additionally, outliers may even misrepresent the underlying data-generating distribution (i.e., be the result of an anomaly or an erroneous measurement), which may lead the learner astray.

To mitigate this issue, Settles and Craven [70] describe the *information density* framework, which extends a base acquisition function ϕ_A by a term incorporating a measure of representativeness of the instance. Instances which inhabit dense

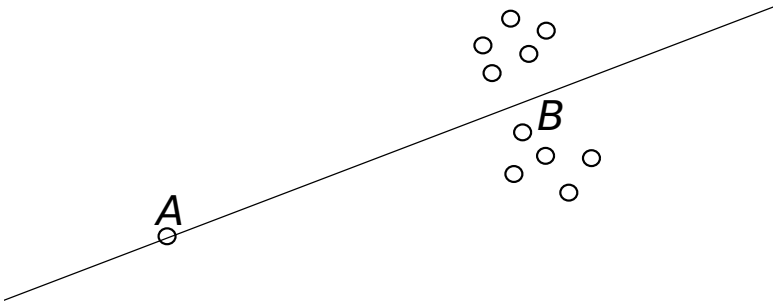


Figure 3.1: The circles represent points in the unlabelled pool. The line represents the decision boundary of a classifier. Outlier point A lies on the decision boundary, and would therefore be likely queried by uncertainty sampling. However, point B is likely to provide more information about the underlying data distribution because it lies in a high-density region of the input space.

regions of the input space are considered to be more representative. Density is measured by the mean similarity to the other instances in the input space, which is approximated by the unlabelled set $\mathcal{U}$:

$$\boldsymbol{x}^*_{\text{ID}} = \arg\max_{\boldsymbol{x}\in\mathcal{U}} \left(\phi_A(\boldsymbol{x}) \times \left(\frac{1}{|\mathcal{U}|} \sum_{\boldsymbol{x}_u \in \mathcal{U}} \text{sim}(\boldsymbol{x}, \boldsymbol{x}_u) \right)^\beta \right) .$$

In the above equation, sim is a function representing similarity between two instances in a given input space. The second multiplicative term weighs the informativeness of an instance by the approximate density of the input space around the instance. The parameter β determines how much the informativeness of an instance is influenced by the density term.

A disadvantage of the information density is that computing the density for all samples in $\mathcal{U}$ grows quadratically with the number of samples in $\mathcal{U}$. Since the unlabelled pool is usually relatively large, this could pose a problem. However, these densities are independent of the base informativeness measure and thus, the learner, which means that they can be pre-computed once and cached for later use. The computation of the densities can be sped-up by clustering $\mathcal{U}$ and computing the similarity only to other instances in the same cluster [71].

3.3 Batch-mode active learning

In all of the query strategy frameworks presented in Section 3.2, we considered querying a single most informative instance. The learner is then typically retrained after each query. However, it may slow or expensive to select the instance to query or to retrain the model after each query. For example, in a pool-based expected model change framework, it may take a long time to evaluate the expected gradient over all possible labellings for all examples in a large unlabelled

set. In that case, it may be beneficial to query new labels in groups, which is referred to as *batch-mode* active learning.

The challenge of batch-mode active learning is selecting the optimal query set $\mathcal{Q}$. Naïvely choosing the B best instances according to a single-instance query strategy (sketched for pool-based sampling in Algorithm 4) can be inefficient because it ignores the overlap in information content among the chosen instances. In other words, an instance may be informative given the current labelled set, but may much less informative given the current labelled and other instances in the same batch. In such case, optimal construction of a batch may require searching through all possible subsets of the unlabelled set, which may be computationally infeasible.

A simple heuristic which has been proposed to mitigate the issue with information overlap is incorporation of a diversity measure into the query function. Greedy batch selection where a base query function is weighed by a diversity term is a popular strategy [67], with minimum distance to instances already in the batch as an example [72].

3.4 Active learning in image classification

In this section, we provide a brief overview of previous work related to active learning applied to image classification (and similar) tasks.

As mentioned before, training a well-performing image classifier typically requires large amounts of labelled images. On the other hand, unlabelled images are usually readily available. As a result, active learning, and specifically, the pool-based sampling scenario, is well-suited to image classification machine learning tasks.

One of the first attempts to use active learning for training of neural networks for image classification was made by Lang and Baum [60], who use membership query synthesis with human oracles to recognize handwritten digits, similar to the long-popular dataset MNIST [73].

Joshi et al. [74] explore batch-mode uncertainty sampling with the addition of an annotation cost term. They use class membership probability estimation for one-vs-all SVMs for multi-class image classification with pre-computed features on the Caltech-101 image dataset [75].

Gal et al. [76] apply the Monte-Carlo dropout Bayesian approximation proposed by Gal and Ghahramani [50] (described in Section 2.4) to active learning image classification on the MNIST digits dataset using a shallow CNN and image-based skin cancer binary classification using a deeper CNN. They compare multiple query strategies in the uncertainty sampling framework to a random baseline in batch-mode pool-based sampling scenario. They also compare their active learning approach to semi-supervised learning [77, 78].

Beluch et al. [79] compare uncertainty sampling using an ensemble of learners to MC dropout and a single network in batch-mode pool-based sampling scenario. They also experiment with a density-weighted method with the weighing by minimum distance to instances already in the batch, which is greedily constructed one instance at a time. They use Euclidean distance between feature vectors ouput by the last hidden layer of the network as a measure of similarity.

Sener and Savarese [80] focus specifically on batch-mode active learning in large-scale image classification, arguing that traditional strategies primarily used for single-instance queries are inefficient when used with greedy batch selection. On the other hand, selection of single instances is impractical due to the complexity of retraining a CNN after each acquisition in large-scale tasks. They define active learning as a problem of core-set selection, which aims to find a small subset of a larger dataset such that a model learned on this subset performs well on the entire dataset. The optimal query set is then computed by approximately minimizing a core-set loss. The core-set loss is a component in decomposition of the expected loss of the learner over the underlying data-generating distribution when trained on an initial labelled set combined with a queried set.

4. Uncertainty sampling active learning in Bayesian neural networks

4.1 Uncertainty estimation in neural networks

Proper estimates of uncertainty of neural networks prediction is important in many applications, where neural networks serve a critical role [81]. As a practical example, high confidence predictions could be used without further human intervention, while lower confidence predictions could be further assessed by a human expert.

As demonstrated by Guo et al. [82], the outputs of modern classification neural networks with high capacity tend to make highly confident predictions by assigning high probability to the most probable predicted class. However, the assigned probability often does not correspond to the true accuracy of predictions. In such case, we say that the predictions are poorly *calibrated*, specifically *overconfident*. Guo et al. experiment with various image classification neural networks, observing that with increasing model capacity, calibration tends to deteriorate. Additionally, batch normalization, which is used in some form in virtually all modern image classification neural networks, seems to further compound the problem of poor calibration.

Classification neural networks output a distribution carrying uncertainty information, even though it is often unreliable. In comparison, classical regression neural networks output a single predicted value (or a vector of values), which carries no uncertainty information. An extension has been proposed for regression [83], where an additional output head is added to the network predicting the variance of the output.

4.1.1 Uncertainty estimation in Bayesian neural networks

Compared to frequentist models, Bayesian neural networks model uncertainty over parameters, which translates into uncertainty over activations and thus predictions. In theory, a Bayesian neural network (or any Bayesian model) outputs a full distribution, which captures the uncertainty in the predictions. However, as discussed throughout Chapter 2, recovering the output distribution exactly is intractable in most practical cases. Therefore, analogously to obtaining predictions, we must resort to sampling. The procedure of uncertainty estimation is thus similar to prediction: T samples of parameters from the posterior are obtained, each used in a forward pass, yielding a prediction. The set of T predictions is then used to compute a suitable statistic measuring the uncertainty. To obtain predictions, the mean over the samples is used.

In regression tasks, a natural choice of a measure of uncertainty is the variance among the individual predictions [84]. In classification tasks, the situation is more complicated, since each sample from the output distribution is a (categorical) distribution itself. A simple solution is to average out one of the dimensions of

the distribution, yielding a distribution which can be used to quantify uncertainty.

An average over the individual predictions yields a single categorical output distribution, which can be used for uncertainty estimation by common uncertainty sampling acquisition functions used in active learning (reviewed in Section 3.2.1). A drawback of this approach is that taking into consideration only the average output of a Bayesian model ignores the uncertainty captured by the distribution over the outputs arising from the uncertainty over the model parameters.

The other possible approach is to take average weighted by predicted values within each sampled prediction. Uncertainty can then be estimated using the variance within the T sampled outputs. Let $p_c^{(t)} = p(y = c \mid \boldsymbol{x}; \boldsymbol{w}^{(t)})$ be the estimated probability of class $c \in C$ given input $\boldsymbol{x}$ and a particular sample of model parameters $\boldsymbol{w}^{(t)}$:

$$\sigma^2 = \frac{1}{|C|} \sum_{c \in C} \frac{1}{T} \sum_{i=1}^{T} \left(p_c^{(t)} - \frac{1}{T} \sum_{i=1}^{T} p_c^{(t)} \right)^2 .$$

This approach ignores the uncertainty captured by the individual sampled outputs, treating each value as a separate regression prediction. Gal et al. [76] experiment with this function in setting of active learning image classification, measuring only slight improvement over baseline uniform random acquisition, while the former approach shows a significant improvement over the random acquisition baseline.

4.2 Bayesian active learning by disagreement

Houlsby et al. [85] propose an extension of the maximum entropy query strategy for Bayesian models called *Bayesian active learning by disagreement (BALD)*. Instead of querying the points which maximize the entropy of the output categorical distribution, BALD acquires instances which maximize the mutual information between predictions and model parameters. Let $q(\boldsymbol{w} \mid \mathcal{L})$ be the posterior distribution over parameters $\boldsymbol{w}$ of a Bayesian model outputting a (categorical) distribution y conditioned on the input $\boldsymbol{x}$ and parameters $\boldsymbol{w}$ trained on a labelled set $\mathcal{L}$. BALD queries training instances according to:

$$\begin{aligned} \boldsymbol{x}^*_{\text{BALD}} &= \arg\max I(y; \boldsymbol{w} \mid \boldsymbol{x}, \mathcal{L}) \\ &= H(\mathbb{E}_{\boldsymbol{w} \sim q(\boldsymbol{w}|\mathcal{L})} [y \mid \boldsymbol{x}; \boldsymbol{w}]) - \mathbb{E}_{\boldsymbol{w} \sim q(\boldsymbol{w}|\mathcal{L})} [H(y \mid \boldsymbol{x}; \boldsymbol{w})] . \end{aligned}$$

The first term in the last equality is the entropy of the expected output of the model, which is the same as in the maximum entropy framework. The second term is the expected entropy of the output. BALD thus combines uncertainty in the averaged output with the uncertainty over the space of possible outputs.

Since the expectation over the model posterior is assumed to be intractable, we resort to approximation by Monte-Carlo sampling. Let $y^{(t)}$ be the categorical distribution given by individual values of $p_c^{(t)}$ for each class. The BALD score is

approximated by T MC samples as

$$I(y; \boldsymbol{w} \mid \boldsymbol{x}, \mathcal{L}) \approx H\left(\frac{1}{T}\sum_{t=1}^{T} y^{(t)}\right) - \frac{1}{T}\sum_{t=1}^{T} H(y^{(t)})$$
$$= -\sum_{c\in C}\left[\left(\frac{1}{T}\sum_{t=1}^{T} p_c^{(t)}\right)\log\left(\frac{1}{T}\sum_{t=1}^{T} p_c^{(t)}\right) - \frac{1}{T}\sum_{t=1}^{T} p_c^{(t)}\log p_c^{(t)}\right].$$

The BALD query strategy can also be used with an ensemble of point-estimate models, using outputs of the individual members of the ensemble instead of samples from a posterior of a Bayesian model [79]. The BALD query strategy can thus be viewed as belonging to the query-by-committee framework. This is a result of the observation that a parametric Bayesian model can be seen as an infinitely large ensemble [29], with every particular sample of parameters representing an individual member of the ensemble.

4.2.1 BatchBALD

Kirsch et al. [86] propose and extension of the BALD query strategy to explicitly consider dependencies within samples selected in a batch in batch-mode active learning (Section 3.3). They note that BALD overestimates the mutual information between model a parameters and a batch of instances by using the sum of individual mutual information values, which counts the information overlap among the samples multiple times.

The authors present a method of theoretically finding the exact optimal batch of acquired samples, which is intractable due to the need to enumerate all possible subsets of unlabelled set. An approximate greedy algorithm is proposed, which the authors prove to be a $(1 - 1/e)$-approximation. They show that the BALD score for a given set is always greater than or equal to the BatchBALD score and BatchBALD is exactly equivalent to BALD when acquiring batches of size 1.

Experiments are performed by the authors on two common image classification benchmark datasets, demonstrating that BatchBALD outperforms ordinary BALD in setting of batch-mode active learning. In these experimentes, the advantage of BatchBALD is only slight when using smaller acquisition batch sizes (e.g., 10) and increases as the size of the acquired batch increases.

5. Experiments

The main goal of the experiments is to show whether Bayesian inference improves the efficiency of active learning and if so which approximate Bayesian inference method yields the highest improvement. As secondary goals, we compare approximate Bayesian inference methods to a classical neural network with frequentist inference and compare training and prediction times of the Bayesian network and the the classical network.

5.1 Experiment design

5.1.1 Acquisition functions

We experiment with all uncertainty sampling acquisition functions mentioned in Section 3.2.1, namely *least confident*, *margin sampling* and *maximum entropy*. Additionally, we use the Bayesian-specific uncertainty sampling function *BALD* (described Section 4.2.1). These acquisition functions are compared with a baseline, *random sampling*, where acquired images are sampled uniformly from the unlabelled set in each iteration.

5.1.2 Approximate Bayesian inference methods

We compare 3 different types of neural networks with approximate Bayesian inference:

- **Monte Carlo dropout (MCDO)** Traditional binary dropout with sampling of dropout masks during prediction, which is described in Section 2.4.
- **Variational Gaussian Dropout (VGDO)**, which is described in Section 2.4.1. We use the variant where a separate dropout rate is learned per neuron.
- **Bayes by Backprop (BBB)**, which was described in Section 2.3.2. We use a simple independent Gaussian prior and compute the KL divergence term using the closed-form solution.

5.1.3 Data

We perform our experiments on two commonly-used dataset used as benchmarks for image classification – *MNIST digits* [73] and *CIFAR-10* [87].

Table 5.1: Summary of the two datasets used in the experiments.

Dataset	Dimensions	Channels	Classes	Train size	Test size
MNIST	28×28	1	10	60 k	10 k
CIFAR-10	32×32	3	10	50 k	10 k

Figure 5.1: Sample of one image from each class in the MNIST dataset.

MNIST digits

MNIST digits is a dataset consisting of greyscale images of handwritten digits. Each of the 10 classes correspond to a digit in a given image. Each image has a dimension of 28×28 pixels. The training set contains 60,000 images and the test set contains 10,000 images. Both the training and the test sets are balanced, i.e. they contain an equal number of images from each class. Figure 5.1 shows a sample of images from the dataset.

Using the full training dataset, classification on MNIST digits is a relatively easy task nowadays. Even simple convolutional networks can achieve an error rate below 0.5% [88], misclassifying only images which are not clearly distinguishable even for a human. In this regard, classification with the full MNIST digits dataset available for training could be considered "solved". However, the dataset is relevant in active learning research, perhaps because learning from a small subset of the data still poses a challenge.

CIFAR-10

CIFAR-10 contains small 32×32 images from 10 mutually exclusive classes. Each pixel in an image has red, blue and green channels. Figure 5.2 shows one image from each of the classes. The training set consists of 5,000 images from each class for a total of 50,000 images and the training set contains 1,000 images from each class for a total of 10,000 images.

Classification on the CIFAR-10 dataset is a much more difficult task and an active area of research, requiring much more complex architectures to achieve state-of-the-art results. Recent CNN models can achieve an error rate as low as about 1% without using additional data in training [89] and even below 1% with pre-training on a larger dataset [90].

5.1.4 Active learning

We use batch-mode pool-based active learning. The training and acquisition procedure is identical for both datasets.

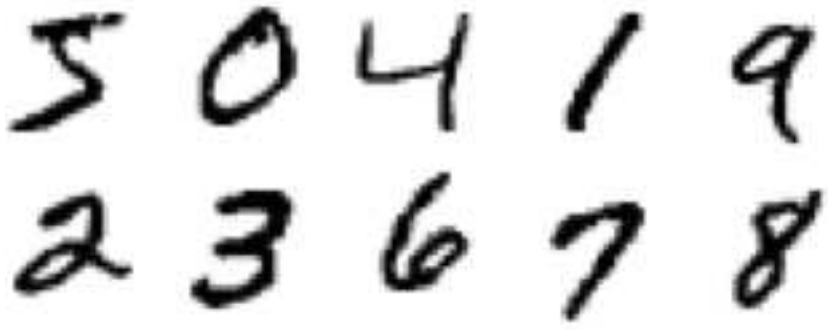


Figure 5.2: One image from each class in CIFAR-10. Top row from left to right: ship, deer, truck, dog, horse. Bottom row: airplane, automobile, cat, bird, frog.

Initialization

The active learning process starts with a class-balanced small seed labelled set and a small class-balanced validation set, both chosen as mutually disjoint subsets of the original training set. The rest of the original training dataset is used as the unlabelled pool.

Active learning loop

Each iteration of the active learning loop consists of three steps:

1. In the **training** step, the model is trained on the current labelled training set until its validation performance converges.

2. In the **evaluation** step, the trained model is evaluated on the test set. The test set is used only to evaluate metrics, i.e. it does not influence the trained models in any way.

3. The **acquisition** step involves evaluating the acquisition function on each instance of the current unlabelled set and selecting a small batch of the most informative instances. The selected instances are assigned labels from the original dataset and moved from the unlabelled pool to the training set.

The iteration of the active learning loop is repeated until a budget is exhausted. We consider the budget to equal to a chosen number of acquired instances.

5.2 Experiment Setup

To compare the performance of the models, we use use the accuracy metric evaluated on the full test set provided with each dataset. We are able to only use accuracy and omit some of the other commonly-used classification metrics such as precision, recall, F1-score, etc., because the test sets are class-balanced in both cases.

5.2.1 Models

Because the two datasets are different especially in terms of complexity, we choose a different CNN architecture for each dataset.

We conceptually divide our models into a feature extractor part, which is fully convolutional in both cases, and a classifier part, which uses fully-connected layers. The feature extractor part is deterministic (non-Bayesian), i.e. it always gives the same output for a given input outside of training. The classifier is Bayesian and we take multiple samples from posterior to make predictions and evaluate uncertainty.

We use no data augmentation. The only pre-processing we perform is normalization to a mean of 0 and a standard deviation of 1 across each channel in the training set, applying the same transformation to the test set, i.e. scaling and shifting by values computed on the training set.

All models are optimized using the SGD-based Adam adaptive optimizer [28] (described in Section 11).

Hyperparameter optimization

In active learning, we train the model at a different training set size after each iteration. The optimal set of hyperparameters could thus be different after each acquisition mainly due to a different amount of regularization required. However, hyperparameter optimization is very computationally demanding and is therefore not realistic to perform it after every acquisition. It is more realistic to perform the hyperparameter optimization at pre-determined steps in training set size. However, this approach still considerably increases the required computation time, especially when each experiment is run with multiple acquisition functions and multiple times.

To save computation time, we perform search for the optimal set of hyperparameters once for each combination of dataset and approximate Bayesian inference method and at various training set sizes. Pre-searching the parameters reduces the time for hyperparameter search 25-fold in our case (5 acquisition functions times 5 runs of each experiment).

Hyperparameter optimization is performed at doubling training set sizes: 50, 100, 200, 400, 800, 1600, 3200 and 6400 instances. Training and validation datasets for hyperparameter search are randomly chosen as a class-balanced sample of the original training set. The size of the validation set used in the optimization is 25% of the training set size with a minimum of 100 instances.

To optimize the models in terms of correctness of predictions as well as good uncertainty calibration, Brier score [91] evaluated on the validation dataset was chosen as the metric according to which the best set of hyperparameters is selected. Brier score is equivalent to the mean squared error between the output probability vectors and the one-hot vector corresponding to the label. Denoting the output vector for the i-th instance as $\hat{\boldsymbol{y}}(\boldsymbol{x}_i)$ and the one-hot label vector as $\boldsymbol{y}_i$, brier score on the set $\mathbb{X} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$ is defined as

$$S_{\text{Brier}} = \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{y_i} - \hat{\boldsymbol{y}}(\boldsymbol{x}_i)\|_2^2 .$$

Table 5.2: Summary of models used for classification on the respective datasets. The number of weights excludes any uncertainty parameters, e.g. learned dropout rates in Variational Gaussian dropout.

Dataset	Weights	Conv Layers	FC layers
MNIST	0.51 M	2	2
CIFAR-10	5.03 M	13	2

The search is performed by Bayesian optimization using a Tree of Parzen estimators [92] surrogate model. Each set run of hyperparameter optimization uses 500 trials, where each model is trained until convergence on the validation, after which the Brier score is evaluated on the validation set. The following hyperparameters are optimized for the individual approximate Bayesian inference methods.

- **Monte Carlo dropout**: Dropout rates (between 0.05 and 0.5).
- **Variational Gaussian Dropout**: Initial dropout rates (between 0.1 and 1) and KL divergence term weight (between 5×10^{-3} and 5). Although the VGDO dropout rates are learnable parameters, we observe a benefit in initializing them to suitable values, perhaps due to a slower convergence rate compared to the weights.
- **Bayes by Backprop**: Prior standard deviations (between 0.1 and 3) and KL divergence term weight (between 5×10^{-3} and 5).

Additionally, we optimize the learning rate (between 5×10^{-4} and 5×10^{-2}) and mini-batch size (one of 16, 32, 64 or 128) for each model.

Next, we describe the individual models we use for each dataset. Table 5.2 summarizes the number of parameters, number of convolutional layers and number of fully-connected layers used in each model.

MNIST digits

For our MNIST digits model, we follow Gal et al. [76], who use a simple model from a Keras reference implementation[1].

Gal et al. slightly modify the architecture from the reference implementation – we use their modified version and modify it further by adding batch normalization (BN, described in Section 1.4.3) after every hidden weight layer to speed up training.

The feature extractor part of our model uses two convolutional layers, each with kernel size 4×4 and 32 filters, each followed by BN and ReLU activation. The convolutions are followed by a 2×2 max-pooling layer and a flattening layer. The classifier consists of one hidden fully-connected layer with 128 units, BN and ReLU activation, leading to the fully-connected output layer with softmax activation.

In case of binary dropout, we follow in Gal et al. [76] by adding binary dropout before the two fully-connected classifier layers. Whereas Gal et al. use

[1] `https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py`

empirically chosen dropout rates, we perform hyperparameter search to set the individual dropout rates as mentioned. In case of variational Gaussian dropout, we use 2 VGDO layers, one before each fully-connected layer. In case of Bayes by Backprop, the two fully-connected classifier layers are BBB layers.

CIFAR-10

For classification on the CIFAR-10 dataset, we use an architecture called SimpleNet [93]. We choose this architecture because of its relative simplicity, both in terms of implementation and computational demands, while achieving good classification performance.

The feature extractor part of the architecture consists of 13 convolutional layers organized in 6 blocks. Each block of convolutions is followed by a 2×2 max-pooling layer and dropout with probability 0.1, except the last layer convolutional, after which global max pooling follows. Each convolutional layer uses a stride of 1 and is followed by batch normalization and ReLU activation. The output uses softmax activation.

The blocks of convolutional are as follows:

1. The first block of convolutional layer starts with one layer of 64 3×3 filters, which is followed by 3 layers, each with 128 filters and 3×3 kernel size.

2. The second block consists of 3 convolutional layers, each with 3×3 filters. The first two layers in the second block use 128 convolutional filters and the third one uses 256 filters.

3. The third block consists of two convolutional layers with 3×3 kernels and 256 filters each.

4. The fourth block consists of a single 3×3 convolutional layer with 512 filters.

5. The fifth block consists of two 1×1 convolutions, with the first one using 2048 filters and the second one using 256 filters.

6. The final block consists of a single 3×3 convolution with 256 filters in the original architecture, which is made to also suit other image classification dataset. However, since we use a dataset with an input size of 32×32, the downsampling from the previous layers results in size of input feature maps to this layer being 1×1. Therefore, we reduce the kernel size to 1×1 to reduce the number of parameters and computations.

The classifier in the original architecture consists of just the output layer, however we modify it by adding a hidden fully-connected layer with 256 hidden units followed by BN and ReLU activation. As mentioned before, we use a deterministic feature extractor and thus we only apply the dropout layers between the convolutional layers during training time.

Similarly to the MNIST model, in case of BBB, we use BBB in the last two hidden layers. In case of VGDO, we put a VGDO layer before each of the two hidden layers, and similarly in case of binary dropout.

Our implementation is based on an unofficial open-source implementation[2].

[2] `https://github.com/Coderx7/SimpleNet_Pytorch`

5.2.2 Active learning setup

Initial training set and validation set

For MNIST, the initial seed labelled set is a class-balanced set of 2 instances per class, i.e., 20 training instances. The validation set consists of 10 instances per class, i.e. 100 instances. The training and validation sizes follow Gal et al. [76]. Both the initial training set and the validation are chosen as subsets of the training set of the original dataset.

For CIFAR, we start with a larger seed set because we observe that with the very small sets, our model is unable to reliably converge to a validation performance better than random guessing. Therefore, we use 10 training instances from each class, i.e. 100 samples and 25 instances per class as the validation set, which gives a validation set size of 250.

Training

Each active learning iteration starts by training the model on the current training labelled set until validation loss converges. We consider the loss to have converged if it has not decreased from the best result seen so far (in the current training iteration) in 5 successive epochs by a factor of at least $1 + \epsilon$, where we set $\epsilon = 5 \times 10^{-5}$.

The training in a given iteration starts with the set of model parameters used in the previous iteration and continues for between 15 and 20 epochs. The set of parameters achieving the lowest loss in the current training iteration is kept and used in further steps. The state of the adaptive optimizer is reset before the training starts in a given iteration. Retraining from scratch (after resetting of parameters) until convergence for a number of epochs without an upper bound is performed only if at least one of the following conditions is satisfied:

- training has not converged within the 20 epochs,
- converged validation performance has not improved compared to the previous iteration,
- hyperparameters were changed since the last iteration,
- the size of the current training set has increased by 10% or more since the model was previously retrained from scratch.

The motivation for starting with parameters used in the previous iteration is faster convergence when only a small amount of data is added to the training set, which helps reduce the high computational demands of the performed experiments.

Hyperparameters are chosen in each iteration from the pre-optimized sets according to the lowest ratio between the size of the current training set and a given training set size used in hyperparameter optimization.

Evaluation

Next is the evaluation step, where metrics are computed on the full test set (10,000 instances in both datasets). Predictions are obtained by averaging softmax model outputs over 40 MC samples. We use a number of samples which is

slightly higher than the numbers used in other similar experiments: Gal et al. [76] use 20 samples, Beluch et al. [79] use 25 samples. The number of samples is a compromise between extracting as much of the performance potential of the models as possible and keeping the computation time reasonable. Using more samples increases the time required to perform the experiments. However, the increase is not too drastic since a large fraction of the time is spent on repeatedly retraining the models, which is largely unaffected by the number of samples used in evaluation, except for evaluating the performance on the relatively small validation set after each epoch.

Acquisition

In the acquisition step, the acquisition function is evaluated for each instance in the current unlabelled set using 40 MC samples. Similarly to evaluation, we use a relatively high number of samples because the penalty is quite low. The 10 most informative instances according to the acquisition function are greedily selected and added to the labelled set.

Stopping criterion

The active learning loop stops when the number of instances in the labelled training set reaches 5000. This number is limited by the time the experiments require to perform and availability of hardware.

5.3 Results

In this section, we present and discuss the results of our experiments.

5.3.1 Bayesian active learning classification on MNIST digits

Table 5.3: Average number of labelled instances needed to achieve a given classification accuracy on the MNIST test set. Bold indicates the best-performing acquisition function with a given approximate Bayesian method at a given threshold. Underlining indicates the best result at a given threshold among all Bayesian approximations and acquisition functions.

Bayes	Accuracy Function	80%	85%	90%	95%	98%	99%
MCDO	Random	89	130	210	464	2931	N/A
	Least conf.	90	131	196	363	890	1961
	Margin	**80**	**100**	**166**	**288**	**834**	**1904**
	Entropy	122	164	266	432	946	2038
	BALD	102	146	204	388	927	2460
VGDO	Random	91	139	273	493	3033	N/A
	Least conf.	96	124	204	334	792	**1970**
	Margin	**82**	**98**	**158**	**292**	**775**	2108
	Entropy	103	173	274	413	880	2140
	BALD	92	137	246	367	839	2138
BBB	Random	94	136	284	696	3740	N/A
	Least conf.	102	120	266	442	1080	**2300**
	Margin	**80**	**92**	**158**	**360**	**972**	2438
	Entropy	116	180	288	486	1154	2356
	BALD	115	178	287	528	1141	2458

Table 5.3 summarizes the number of labelled instances needed to achieve chosen classification accuracy threshold for all combinations of Bayesian approximations and acquisition functions. Note that in the last column, the "N/A" values mean that the given threshold was not achieved with up to 5000 labelled examples, at which point the experiments were stopped. Figure 5.4 compares all acquisition functions using each of the approximate Bayesian methods. Figure 5.3 compares the approximate Bayesian methods using three of the examined acquisition functions: baseline random sampling, margin sampling and BALD. All results are averaged over 5 runs.

From these results, we can make the following observations:

- Margin sampling slightly outperforms all other acquisition functions at lower to medium labelled set sizes. The advantage of margin sampling is the largest at the 90% accuracy threshold. At higher labelled set sizes, the performance is mostly equalized between the acquisition functions.

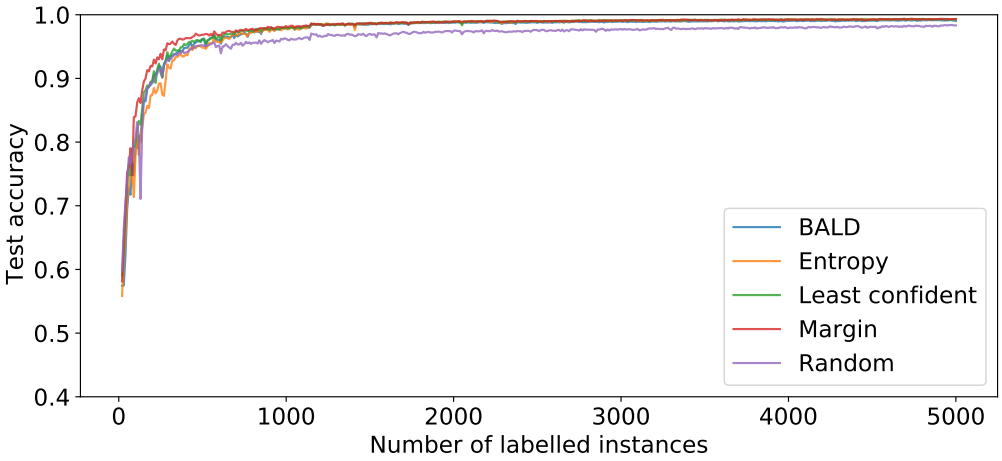
- At lower labelled set sizes, the advantage of active learning compared to random acquisition is small. All acquisition functions except Margin are initially slightly outperformed by random sampling. A possible explanation is that the first acquisitions by uncertainty sampling acquire many of the outliers in the data.

- As the number of acquired instances increases, all active learning acquisition function outperform random sampling. An explanation of this observation is that once the learner has seen more data, it is able to form a better representation of the data, which allows it to make better decisions about which instances might be the most informative.

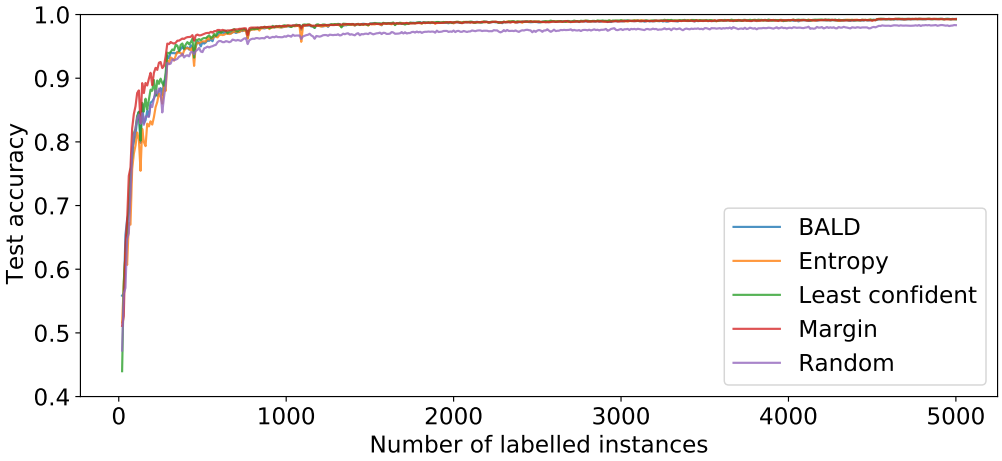
- Monte Carlo dropout and Variational Gaussian dropout perform generally similarly to each other in terms of the number of instances to reach given thresholds, with some exceptions. One such exception is the one region between about 200 to 400 training instances, where in Figure 5.3b we observe a dip in performance of VGDO. The problem likely originates in poorly-chosen hyperparameters for VGDO. Since the hyperparameter search is stochastic, such occurrence is difficult to prevent.

- Bayes by Backprop performs well at smaller training set sizes, requiring the fewest examples to achieve 80%, 85% and 90% accuracy with the best-performing function, which in all three cases is the Margin function. However, as more training examples are acquired, Bayes by Backprop starts lagging behind the other Bayesian approximations. Since BBB lags behind in random acquisition performance similarly to active learning performance, the problem seems to be generally poorer classification performance of BBB, not poorly-made acquisitions by BBB.

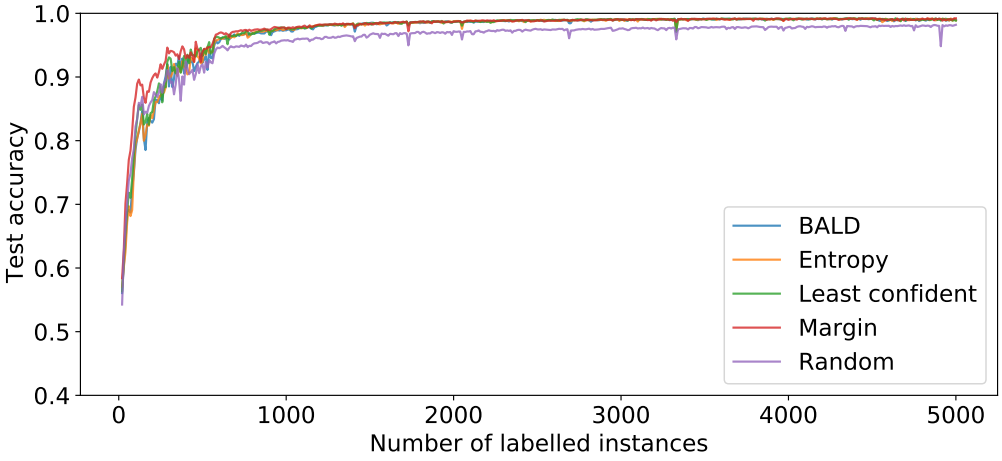
- In Figures 5.3 and 5.4, we can see occasional dips in the model performance, which we observed to occur mostly when the training set size modulo the batch size is small (minimum is 2), where one of the batches is small, which negatively affects the convergence.



(a) Monte Carlo Dropout

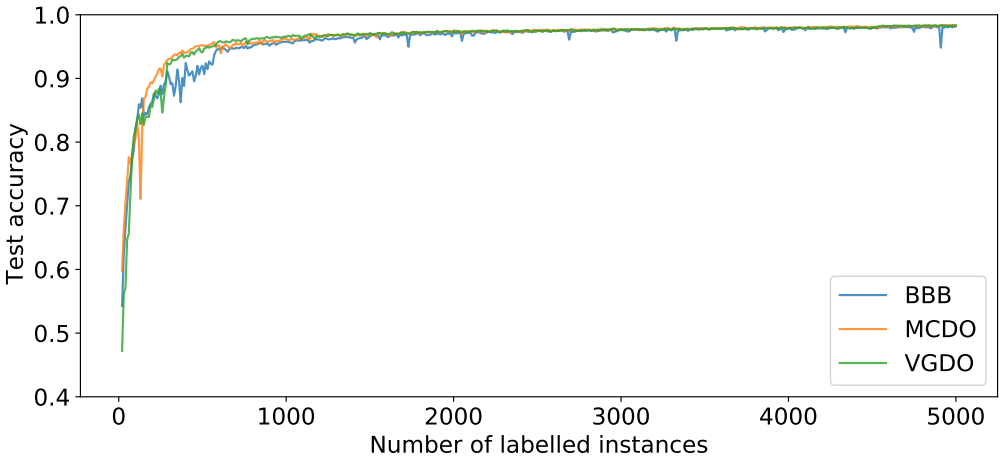


(b) Variational Gaussian Dropout

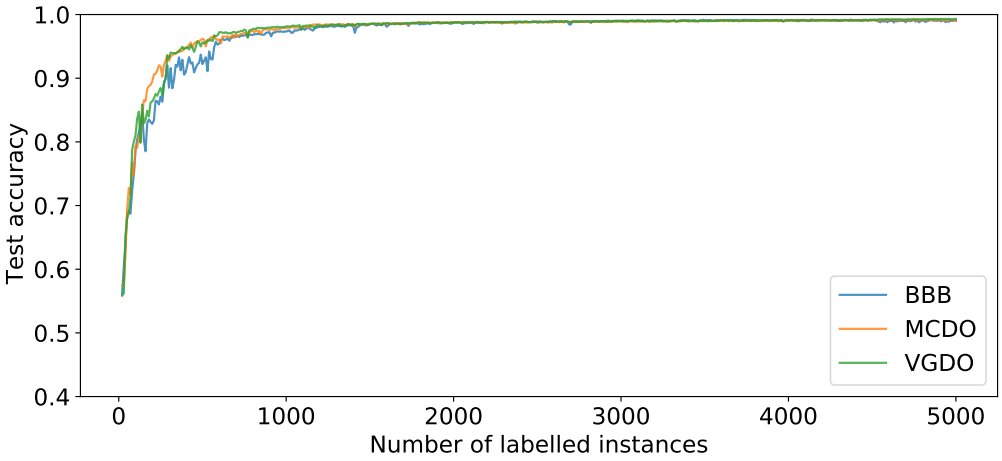


(c) Bayes by Backprop

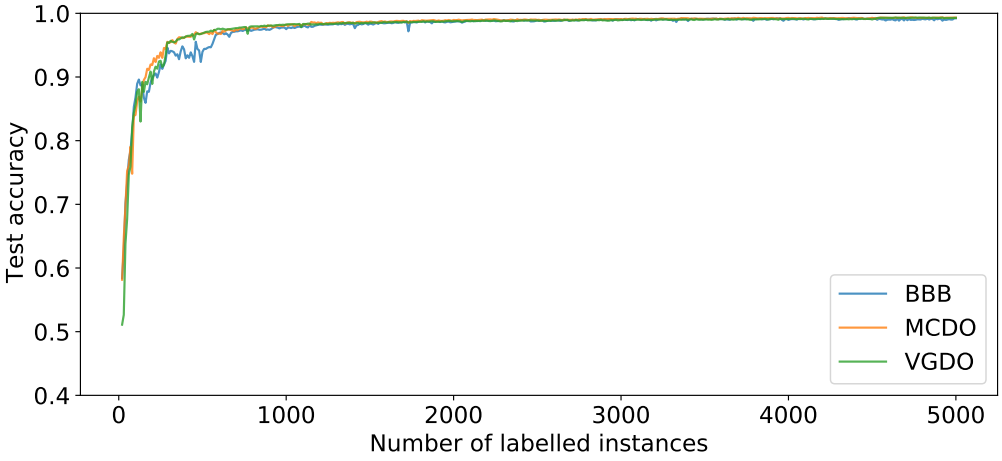
Figure 5.3: Comparison of active learning classification accuracies after each active learning iteration across all the examined acquisition functions on MNIST digits for each of the individual approximate Bayesian inference methods.



(a) Random



(b) BALD



(c) Margin

Figure 5.4: Comparison of active learning classification accuracies after each active learning iteration across all the examined approximate Bayesian methods on MNIST digits using acquisition functions Random, Margin and BALD.

5.3.2 Bayesian active learning classification on CIFAR-10

Table 5.4 summarizes the average number of labelled instances needed to achieve chosen classification accuracy threshold for all combinations of Bayesian approximations and acquisition functions used in experiments on the CIFAR-10 dataset. Figure 5.5 shows a comparison of classification accuracies after each acquisition across all the examined acquisition functions (including the random baseline) on the CIFAR-10 dataset for each individual approximate Bayesian inference method. Figure 5.6 shows a comparison of classification accuracies after each acquisition across all the approximate Bayesian methods using three acquisition functions: baseline Random sampling, Margin and BALD. The results are averaged over 5 runs.

From these results, we make the following observations:

- None of the examined acquisition functions convincingly outperforms the random sampling baseline.

- It seems that active learning using MCDO with BALD or Margin acquisition and VGDO with BALD acquisition start to slightly outperform random sampling at higher training set sizes. On the other hand, in BBB, BALD is on two functions which were not able to achieve the 67% accuracy threshold in all 5 runs.

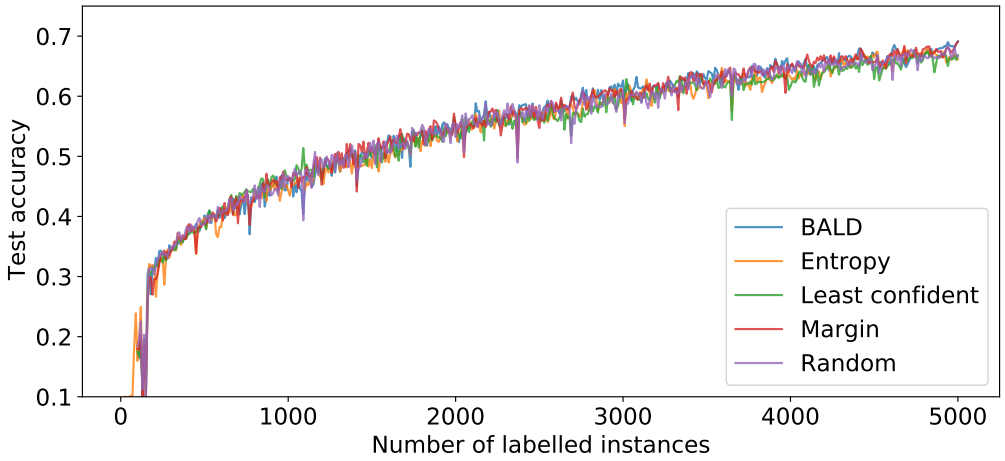
  This hypothesis is consistent with the behavior seen in MNIST – the advantage of active learning compared to random sampling is much greater as more data is seen and the learner is thus able to form a better internal representation of the data. However, it is difficult to conclude whether this is a spurious occurrence without performing more runs of the experiments and and/or running the experiments for iterations.

- Bayes by Backprop performs generally worse compared to Monte Carlo Dropout and Variational Gaussian Dropout in terms of the number of acquisitions needed to reach the given accuracy thresholds. The performance of MCDO and VGDO is similar to each other.

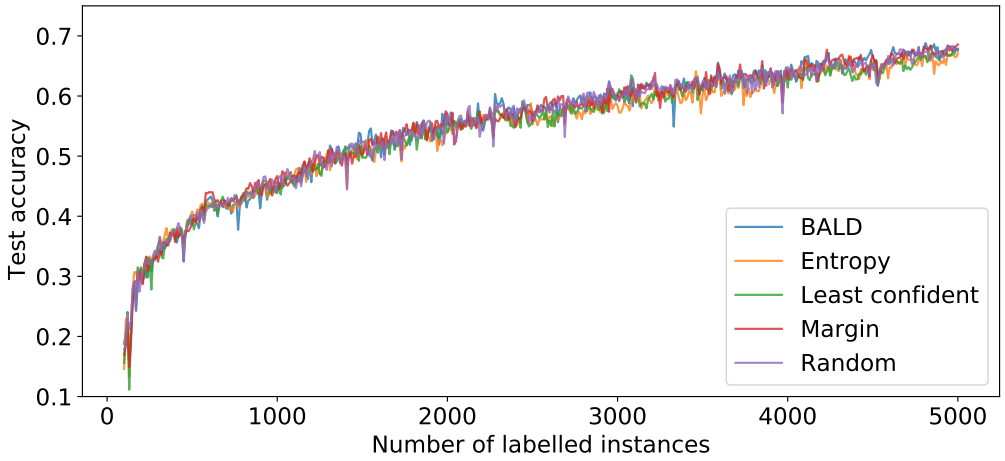
- The results on CIFAR-10 are noisier than the results on MNIST. The problem with convergence when one of the batches is small is also more evident.

Table 5.4: Average number of labelled instances needed to achieve a given classification accuracy on the CIFAR-10 test set. Bold face indicates the best-performing acquisition function with a given approximate Bayesian method at a given threshold. Underlining indicates the best result at a given threshold among all Bayesian approximations and acquisition functions.

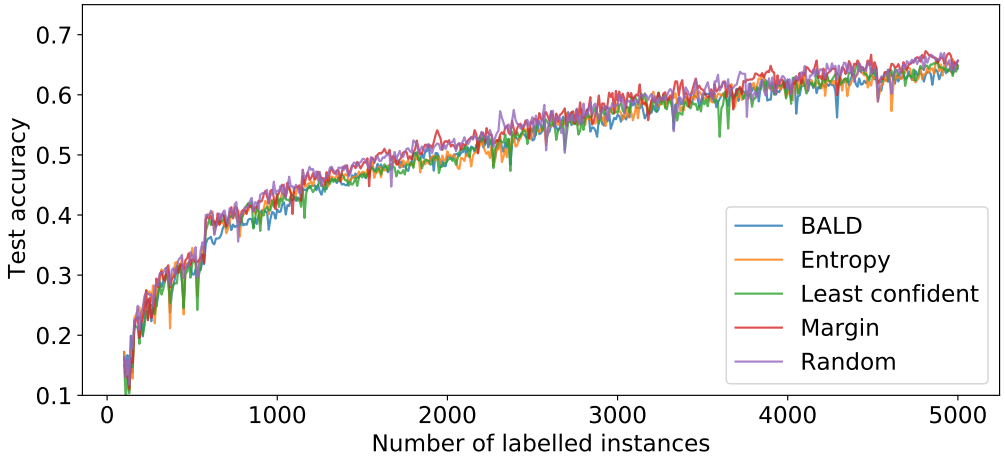
Bayes	Accuracy Function	40%	50%	60%	67%
MCDO	Random	**468**	1158	2392	4184
	Least conf.	504	1234	2354	4274
	Margin	498	**1086**	**2308**	3858
	Entropy	523	1158	2615	4235
	BALD	490	1210	2462	**3834**
VGDO	Random	484	1196	2328	4320
	Least conf.	472	1210	2490	4442
	Margin	482	**1074**	2360	4222
	Entropy	**448**	1232	2390	4252
	BALD	500	1216	**2242**	**4020**
BBB	Random	602	1512	2764	4393
	Least conf.	646	1710	2912	N/A
	Margin	612	**1508**	**2692**	**4222**
	Entropy	**594**	1678	2978	4768
	BALD	760	1680	3072	N/A



(a) Monte Carlo Dropout

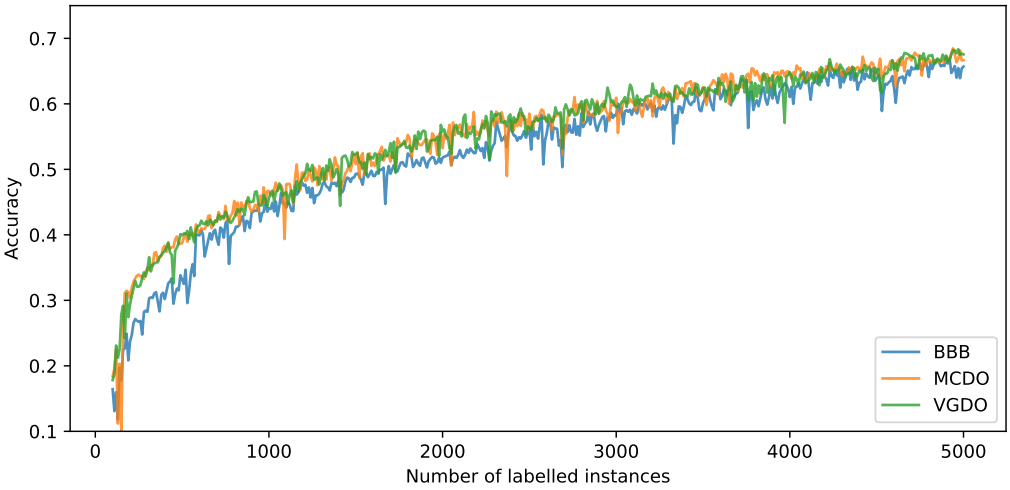


(b) Variational Gaussian Dropout

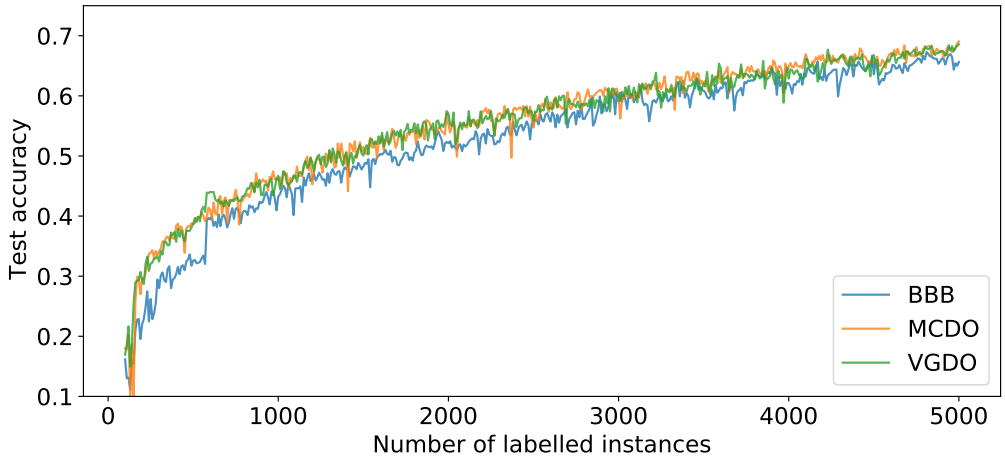


(c) Bayes by Backprop

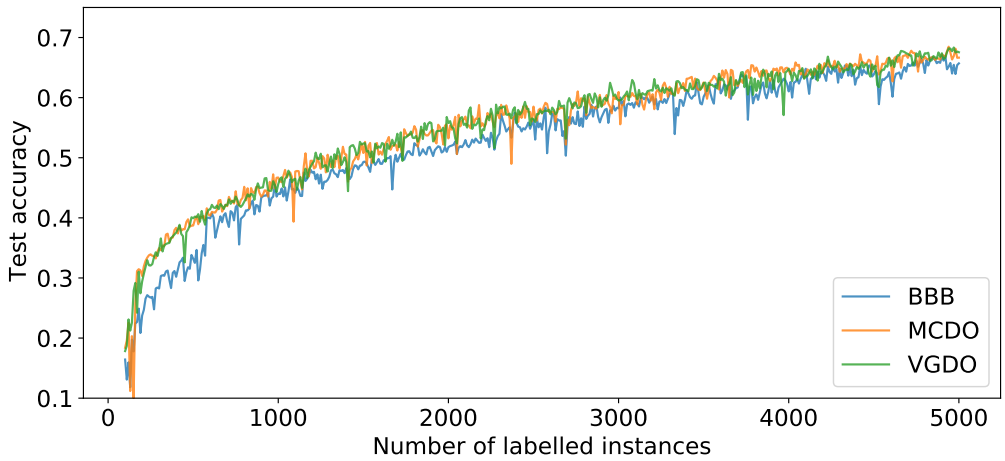
Figure 5.5: Comparison of acquisition functions in Bayesian active learning. Each plot shows the accuracy at all acquired training set sizes for one approximate Bayesian inference method and all examined acquisition functions.



(a) Random



(b) Margin



(c) BALD

Figure 5.6: Comparison of approximate Bayesian inference methods in Bayesian active learning. Each plot shows shows the accuracy at every acquired training set size for all approximate Bayesian inference using a single acquisition function: random baseline, BALD or Margin.

5.3.3 Comparison to non-Bayesian methods

In this section, we compare Monte Carlo dropout to a non-Bayesian neural network with the same architecture and hyperparameters as the MC dropout network. The goal is to show whether the improvement in performance using active learning compared to the random baseline can be attributed to the Bayesian nature of the models.

The non-Bayesian neural network makes standard dropout predictions, which means that dropout is disabled when making predictions. We compare the model using all the acquisition functions used in the experiments with Bayesian neural networks including random sampling and excluding BALD, which is specific to Bayesian models. Except for the different model, the experiment setup is identical to the the setup used in the experiments with Bayesian neural networks.

MNIST digits

Table 5.5 summarizes the average of numbers of instances required to reach given accuracy thresholds on the MNIST test set, comparing the non-Bayesian model with the Monte Carlo dropout model. From these results, we can make the following observations:

- Uncertainty sampling active learning significantly outperforms baseline random sampling on the MNIST digits dataset even when using a non-Bayesian model.
- The non-Bayesian standard dropout model performs no worse than the Monte Carlo dropout model. It even slightly outperforms Monte Carlo dropout, especially as more data is seen.
- The comparison between the acquisition functions in the non-Bayesian model is very similar to the MC dropout model: Margin performs the best, closely followed by Least confident and Entropy. The performance of all the acquisition functions (except random) equalizes as the models approach perfect classification.

Table 5.5: Comparison of average number of labelled instances needed to achieve a given accuracy on the MNIST digits test set between a non-Bayesian neural network (Classical) and a neural network using MC dropout

Type	Accuracy Function	80%	85%	90%	95%	98%	99%
Classical	Random	90	132	190	456	2572	N/A
	Least conf.	94	126	182	326	812	1724
	Margin	**72**	**98**	**144**	**286**	**776**	**1672**
	Entropy	106	146	226	422	880	1820
MC dropout	Random	89	130	210	464	2931	N/A
	Least conf.	90	131	196	363	890	1961
	Margin	**80**	**100**	**166**	**288**	**834**	**1904**
	Entropy	122	164	266	432	946	2038

CIFAR-10

Table 5.5 summarizes the average of numbers of instances required to reach given accuracy thresholds on the CIFAR-10 test set, comparing the non-Bayesian model with the Monte Carlo dropout model.

Similarly to MNIST digits, we can see that the non-Bayesian model performs approximately equally to the Bayesian MC dropout model.

Table 5.6: Comparison of average number of labelled instances needed to achieve a given classification accuracy on the CIFAR-10 test set between a non-Bayesian model (Classical) and a model using MC dropout.

Type	Accuracy Function	40%	50%	60%	67%
Classical	Random	**404**	1110	2270	4262
	Least conf.	478	1184	2422	4100
	Margin	420	**1100**	**2264**	**3862**
	Entropy	530	1212	2484	4154
MC Dropout	Random	**468**	1158	2392	4184
	Least conf.	504	1234	2354	4274
	Margin	498	**1086**	**2308**	**3858**
	Entropy	523	1158	2615	4235

In conclusion, we observe no clear advantage in using Monte Carlo dropout compared to standard dropout prediction in uncertainty sampling active learning. Since Monte Carlo dropout performs similarly to variational Gaussian dropout and better than Bayes by Backprop, we conclude that our experiments show no advantage in using Bayesian neural networks compared to classical neural networks in setting of uncertainty sampling active learning.

Uncertainty calibration

To investigate why the Bayesian neural networks do not seem to perform better than the classical neural networks, we investigate the uncertainty calibration of the individual models. We explore the uncertainty given by the score used in the least confident strategy and the error, which is the sum of predicted probabilities of all classes, except the most probable one, i.e. one minus the probability of the most probable class.

The error rate is evaluated on the test set and is a complement to the accuracy. The uncertainty is evaluated on the unlabelled part of the training set. Even though these are two different sets, they both represent a large sample of the underlying data-generating distributions. However, the unlabelled training set becomes more and more biased towards "easier" instances as the acquisition process continues. Since all the results carry the same bias, we believe they still provide a reasonable comparison. An unbiased estimate could be obtained by evaluating the uncertainty on the test set, however this was not measured in the original experiments and re-running the experiments to obtain these measurements was unfortunately not possible.

The results are shown in Figure 5.7 for MNIST digits and in Figure 5.8 or CIFAR-10. Each point in the scatter plots is obtained using the model at one

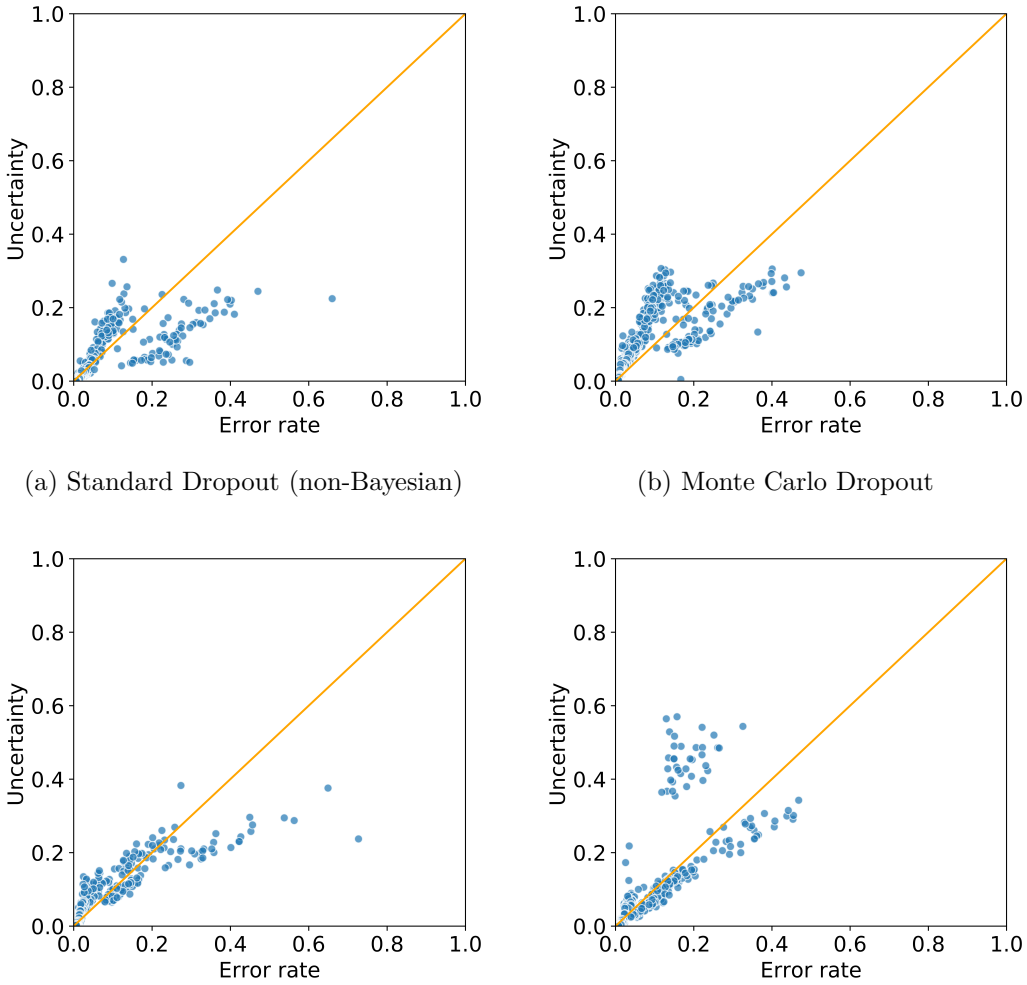
iteration of the active learning loop. More specifically, each point corresponds to the average uncertainty on the unlabelled part of the training on one axis and the true error rate evaluated on the test set on the other axis. Points from all 5 experiment runs are used.

The plots also show a line representing the identity function, which is optimal – the closer the points lie to the line, the better the calibration. Points above the line correspond to models which make under-confident predictions on average and points below the correspond to models which make over-confident predictions on average.

We can see that on MNIST, the predictions are relatively well-calibrated and that the predictions of the variational dropout model show the best calibration, with the non-Bayesian model closely behind. On the other hand, the MC dropout model shows the worst calibration. Note that the predictions of the standard dropout and the Monte Carlo dropout models are *under-confident* more often than over-confident, which is atypical in neural networks.

On CIFAR-10, all the types of models are similarly over-confident in their predictions.

Table 5.7 compares the expected calibration error among all the Bayesian models and the non-Bayesian models on both datasets. The expected calibration error [94] is the mean absolute error between the uncertainties and the true error rates, averaged over all points.



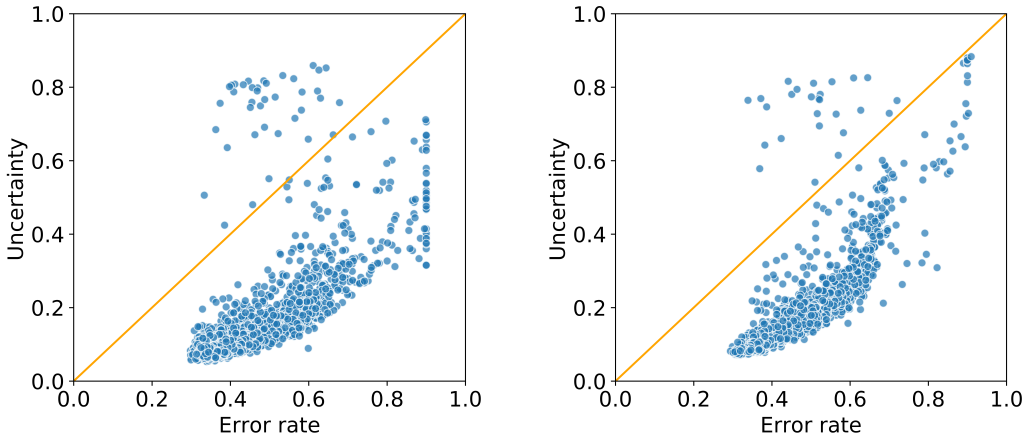
(a) Standard Dropout (non-Bayesian)

(b) Monte Carlo Dropout

(c) Variational Gaussian Dropout

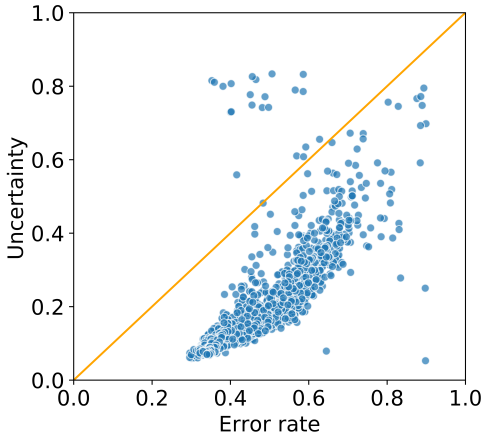
(d) Bayes by Backprop

Figure 5.7: Calibration plots of each model type on the MNIST digits dataset. Each point corresponds to one iteration of the active learning loop The orange line shows the identity function, which corresponds to perfect calibration.

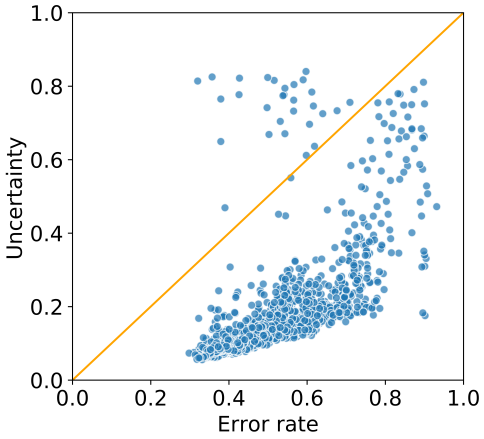


(a) Standard Dropout (non-Bayesian)

(b) Monte Carlo Dropout



(c) Variational Gaussian Dropout



(d) Bayes by Backprop

Figure 5.8: Calibration plots of each model type on the CIFAR-10 dataset. Each point corresponds to one iteration of the active learning loop The orange line shows the identity function, which corresponds to perfect calibration.

Table 5.7: Expected calibration errors of all model types on both datasets.

	Classical	MCDO	VGDO	BBB
MNIST	0.0120	0.0151	0.0117	0.0133
CIFAR-10	0.317	0.285	0.283	0.343

5.3.4 Training and prediction time

Table 5.8 summarizes the training and prediction times per batch for each approximate Bayesian inference method and the standard dropout model measured on both MNIST and CIFAR-10. The results are averaged over 5000 batches. The measurements were carried out on a server with an Intel Xeon E5-2650v4 GHz CPU clocked at 2.2 GHz, 256 GB DDR4 RAM, an Nvidia Tesla V100 16 GB GPU using CUDA version 9.0 and Ubuntu 16.04 operating system.

We can see that the increase in training time of the Bayesian models is not dramatic compared to the frequentist model. The increase is more pronounced in MNIST, where a larger fraction the of weights are subject to approximate Bayesian inference. During prediction, the increase in time of the Bayesian models over the classical models is much more pronounced. The increase in prediction time of Monte Carlo dropout compared to standard dropout is only approximately 4-fold in CIFAR-10 and 16-fold in MNIST despite taking 40 samples. The reason is that the feature extractors we used are not Bayesian and thus a single forward pass without Monte Carlo sampling is used in the feature extractor. We can also see that all the Bayesian models work well with large batches. The increase in training time per batch between batch size 128 and 32 is around 10 to 15%, similarly to the non-Bayesian models.

Also note that in case of Bayes by Backprop, the prediction time for MNIST is actually higher than the prediction time for CIFAR-10. This is because in the MNIST model, we have more parameters in the last two Bayesian fully-connected layers compared to the CIFAR-10 model. This is because the CIFAR-10 model uses global pooling at the end of the feature extractor, while the MNIST model flattens the spatial dimensions, which are 11×11 at the end of the feature extractor, which means that the classifier receives $11 * 11 * 32$ features, compared to 256 input features received by the CIFAR-10 classifier. The MNIST Bayes by Backprop model has almost 497 thousand uncertainty parameters compared to the CIFAR-10 Bayes by Backprop model, which has approximately 68 thousand uncertainty parameters.

Table 5.8: Means and standard deviations of model training prediction times per batch over 5000 batches. All values are in milliseconds. The prediction times are measured using 40 forward passes for the Bayesian models and a single forward pass for the classical non-Bayesian model, same as in the other experiments. The training times are measured at batch sizes 32 and 128 and the prediction times are measured at a batch size of 32.

Dataset	**Type**	**Train @ 32**	**Train @ 128**	**Pred @ 32**
MNIST	Classical	3.95 ± 0.42	4.40 ± 0.49	0.75 ± 0.03
	MCDO	4.05 ± 0.47	4.63 ± 0.53	12.2 ± 0.45
	VGDO	6.43 ± 0.76	7.15 ± 0.61	22.8 ± 0.93
	BBB	7.21 ± 0.94	8.02 ± 0.89	35.7 ± 3.86
CIFAR-10	Classical	18.3 ± 1.89	20.8 ± 1.78	3.03 ± 0.26
	MCDO	18.6 ± 1.97	21.3 ± 2.06	13.4 ± 0.53
	VGDO	22.1 ± 3.00	24.8 ± 2.38	25.4 ± 1.10
	BBB	20.7 ± 2.50	22.7 ± 2.33	31.4 ± 1.12

Conclusion

The goal of this thesis was to survey neural networks and methods for approximate Bayesian inference in neural networks for image classification and evaluate these methods in setting of uncertainty sampling active learning.

In the thesis, we reviewed neural networks and neural networks for image classification, three approximate Bayesian inference methods and some of their specifics when used in convolutional networks. Next, we reviewed active learning and uncertainty sampling active learning using Bayesian neural networks. Finally, we performed experiments where we compared all three reviewed approximate Bayesian inference methods using four different active learning acquisition functions and a random sampling baseline. We compared the results of the Bayesian models to non-Bayesian models. The following conclusions can be made from the results of the experiments:

- Active learning on image classification shows considerable improvement compared to baseline random sampling on MNIST digits, where supervised classification is a relatively easy task. On CIFAR-10, which is a harder dataset, active learning performs similarly to random sampling.
- We have found no improvement using any of the examined approximate Bayesian inference methods compared to a classical non-Bayesian neural network. On the MNIST digits dataset, we found the non-Bayesian model to perform slightly better than all the Bayesian models. On CIFAR-10, Bayesian and non-Bayesian networks perform similarly.

Future work

The main natural continuation of our work would be performing more similar experiments, which we we were unable to do due to very high computational demands. Possibilities for further experiments include:

- more runs of each experiment to obtain more reliable results,
- experimenting on additional datasets – both image classification datasets and other types of datasets, such as sequence labelling, regression or other computer vision tasks,
- running the experiments for more iterations of acquisition, especially for more complex datasets, such as CIFAR-10 in our case,
- using other acquisition functions, for example density-weighed methods or other batch-mode acquisition functions,
- evaluating the possible benefits of using approximate Bayesian inference in the whole neural network as opposed to just a few layers at the end of the network at the end, as was done in our experiments.

Bibliography

[1] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[3] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[4] J.S. Cramer. The origins of logistic regression. *Tinbergen Institute, Tinbergen Institute Discussion Papers*, 01 2002.

[5] Leo Breiman, Joseph H Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. 1983.

[6] Tin Kam Ho. Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, page 278, USA, 1995. IEEE Computer Society.

[7] Jerome H. Friedman. Stochastic gradient boosting. *Comput. Stat. Data Anal.*, 38(4):367–378, February 2002.

[8] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8:679 – 698, 12 1986.

[9] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[10] Hans Feichtinger and Georg Zimmermann. *Gabor Analysis and Algorithms*, pages 123–170. 01 1998.

[11] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[12] Frank F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[13] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[15] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[16] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, page 9–50, Berlin, Heidelberg, 1998. Springer-Verlag.

[17] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[19] Ronald A Fisher. On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 222(594-604):309–368, 1922.

[20] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

[21] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[22] Xiaoyu Li and Francesco Orabona. On the convergence of stochastic gradient descent with adaptive stepsizes. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *Proceedings of Machine Learning Research*, volume 89 of *Proceedings of Machine Learning Research*, pages 983–992. PMLR, 16–18 Apr 2019.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[24] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999.

[25] Robert A. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1:295–307, 1988.

[26] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[27] Matthew D. Zeiler. Adadelta: An adaptive learning rate method. *ArXiv*, abs/1212.5701, 2012.

[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[29] 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.

[30] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[31] Diederik P. Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. 2015.

[32] Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep pyramidal residual networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6307–6315, 2016.

[33] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[35] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[36] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[37] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network, 2013.

[38] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[39] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 05 2019.

[40] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[41] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

[42] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

[43] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, chapter 10. Springer-Verlag, Berlin, Heidelberg, 2006.

[44] Geoffrey E. Hinton and Drew van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, COLT '93, page 5–13, New York, NY, USA, 1993. Association for Computing Machinery.

[45] Alex Graves. Practical variational inference for neural networks. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc., 2011.

[46] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1613–1622. JMLR.org, 2015.

[47] Kumar Shridhar, Felix Laumann, Adrian Llopart Maurin, Martin Olsen, and Marcus Liwicki. Bayesian convolutional neural networks with variational inference. 2018.

[48] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[49] J. . Durrieu, J. . Thiran, and F. Kelly. Lower and upper bounds for approximation of the kullback-leibler divergence between gaussian mixture models. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4833–4836, 2012.

[50] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1050–1059. JMLR.org, 2016.

[51] Sida I. Wang and Christopher D. Manning. Fast dropout training. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page II–118–II–126. JMLR.org, 2013.

[52] Dmitry Molchanov, Arsenii Ashukha, and Dmitry P. Vetrov. Variational dropout sparsifies deep neural networks. In *ICML*, 2017.

[53] Yarin Gal and Zoubin Ghahramani. Bayesian convolutional neural networks with Bernoulli approximate variational inference. In *4th International Conference on Learning Representations (ICLR) workshop track*, 2016.

[54] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In Edwin R. Hancock Richard C. Wilson and William A. P. Smith, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press, September 2016.

[55] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[56] Meire Fortunato, Charles Blundell, and Oriol Vinyals. Bayesian recurrent neural networks. *CoRR*, abs/1704.02798, 2017.

[57] Felix Laumann, Kumar Shridhar, and Adrian Llopart Maurin. Bayesian convolutional neural networks. *CoRR*, abs/1806.05978, 2018.

[58] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. A comprehensive guide to bayesian convolutional neural network with variational inference. *CoRR*, abs/1901.02731, 2019.

[59] Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.

[60] Kenneth Lang and Eric Baum. Query learning can work poorly when a human oracle is used, 1992.

[61] Les E. Atlas, David A. Cohn, and Richard E. Ladner. Training connectionist networks with queries and selective sampling. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 566–573. Morgan-Kaufmann, 1990.

[62] David D. Lewis and William A. Gale. A sequential algorithm for training text classifiers. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '94, page 3–12, Berlin, Heidelberg, 1994. Springer-Verlag.

[63] Cha Zhang and Tsuhan Chen. An active learning framework for content-based information retrieval. *Trans. Multi.*, 4(2):260–268, June 2002.

[64] Simon Tong and Edward Chang. Support vector machine active learning for image retrieval. In *Proceedings of the Ninth ACM International Conference on Multimedia*, MULTIMEDIA '01, page 107–118, New York, NY, USA, 2001. Association for Computing Machinery.

[65] X. Li and Y. Guo. Adaptive active learning for image classification. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 859–866, 2013.

[66] A. J. Joshi, F. Porikli, and N. Papanikolopoulos. Multi-class active learning for image classification. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2372–2379, 2009.

[67] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[68] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 287–294, New York, NY, USA, 1992. Association for Computing Machinery.

[69] Burr Settles, Mark Craven, and Soumya Ray. Multiple-instance active learning. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1289–1296. Curran Associates, Inc., 2008.

[70] Burr Settles and Mark Craven. An analysis of active learning strategies for sequence labeling tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, page 1070–1079, USA, 2008. Association for Computational Linguistics.

[71] Hieu T. Nguyen and Arnold Smeulders. Active learning using pre-clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 79, New York, NY, USA, 2004. Association for Computing Machinery.

[72] Klaus Brinker. Incorporating diversity in active learning with support vector machines. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 59–66. AAAI Press, 2003.

[73] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[74] Ajay Joshi, Fatih Porikli, and Nikolaos Papanikolopoulos. Multi-class active learning for image classification. pages 2372–2379, 06 2009.

[75] Li Fei-Fei, Rob Fergus, and Pietro Perona. One-shot learning of object categories. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(4):594–611, April 2006.

[76] Yarin Gal, Riashat Islam, and Zoubin Ghahramani. Deep bayesian active learning with image data. *CoRR*, abs/1703.02910, 2017.

[77] Olivier Chapelle, Bernhard Schlkopf, and Alexander Zien. *Semi-Supervised Learning*. The MIT Press, 1st edition, 2010.

[78] Antti Rasmus, Mathias Berglund, Mikko Honkala, Harri Valpola, and Tapani Raiko. Semi-supervised learning with ladder networks. In *Advances in neural information processing systems*, pages 3546–3554, 2015.

[79] W. H. Beluch, T. Genewein, A. Nurnberger, and J. M. Kohler. The power of ensembles for active learning in image classification. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9368–9377, 2018.

[80] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach, 2017.

[81] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

[82] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1321–1330. JMLR.org, 2017.

[83] D. A. Nix and A. S. Weigend. Estimating the mean and variance of the target probability distribution. In *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, volume 1, pages 55–60 vol.1, 1994.

[84] Felix Laumann, Kumar Shridhar, and Adrian Llopart Maurin. Bayesian convolutional neural networks. *CoRR*, abs/1806.05978, 2018.

[85] Neil Houlsby, Ferenc Huszar, Zoubin Ghahramani, and Máté Lengyel. Bayesian active learning for classification and preference learning. *CoRR*, abs/1112.5745, 2011.

[86] Andreas Kirsch, Joost van Amersfoort, and Yarin Gal. Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning. *CoRR*, abs/1906.08158, 2019.

[87] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[88] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 2*, ICDAR '03, page 958, USA, 2003. IEEE Computer Society.

[89] Tal Ridnik, Hussam Lawen, Asaf Noy, and Itamar Friedman. Tresnet: High performance gpu-dedicated architecture. *arXiv preprint arXiv:2003.13630*, 2020.

[90] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning, 2019.

[91] GLENN W. BRIER. Verification of forecasts expressed in terms of probability. *Monthly Weather Review*, 78(1):1–3, 1950.

[92] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 2546–2554, Red Hook, NY, USA, 2011. Curran Associates Inc.

[93] Seyyed Hossein HasanPour, Mohammad Rouhani, Mohsen Fayyaz, and Mohammad Sabokrou. Lets keep it simple, using simple architectures to outperform deeper and more complex architectures. *arXiv preprint arXiv:1608.06037*, 2016.

[94] Mahdi Pakdaman Naeini, Gregory F Cooper, and Milos Hauskrecht. Obtaining well calibrated probabilities using bayesian binning. In *AAAI*, page 2901–2907, 2015.

List of Figures

List of Tables

A. Attachments

A.1 Implementation

A.1.1 Used technologies

The software used to perform the experiments is implemented in Python (version 3.8.3). The following Python libraries were used:

- PyTorch[1] (1.5.1) is used to build, train and evaluate the neural networks and run the required computations using GPU acceleration.
- Torchvision[2] (0.6.0) contains extension of PyTorch specialized for computer vision. It is used to load the datasets used in the experiments.
- NumPy[3] (1.19.0) is used to process multi-dimensional arrays.
- Pandas[4] (1.0.5) is used to store, load and process results of the experiments.
- hyperopt[5] (0.2.4) is used to perform Bayesian hyperparameter optimization.
- Jupyter Notebook[6] (6.1.5) is used to explore the results of the experiments.
- Seaborn[7] (0.10.1) and Matplotlib[8] (3.2.2) are used to create plots of the results.

A.1.2 Usage

In this section, we describe how to run the experiments. The instructions below are

Installation

The project uses Poetry for Python dependency management. Installation instructions for Poetry can be found at `https://python-poetry.org/docs/`. The required dependencies are specified in the file *pyproject.toml* in the root project directory. The required dependencies can also be manually installed using other tools, such as *Conda* or *Pip*. To install the required dependencies using Poetry, run

```
poetry install
```

[1] `https://pytorch.org/`
[2] `https://pytorch.org/docs/stable/torchvision/index.html`
[3] `https://numpy.org/`
[4] `https://pandas.pydata.org/`
[5] `https://github.com/hyperopt/hyperopt`
[6] `https://jupyter.org/`
[7] `https://seaborn.pydata.org/`
[8] `https://matplotlib.org/`

in the project root. This will create a Poetry Python environment and install all the required dependencies in it. To run a command within the created environment, use

```
poetry run <command>
```

For example, to run Python within the Poetry environment, run

```
poetry run python
```

Alternatively, it possible to issue the following command:

```
poetry shell
```

Running this command spawns a shell with the poetry environment activated. In this shell, the prefix `poetry run` can be omitted when running commands required to be run within the project environment.

Running the experiments

The experiments are run using the script *main_active.py* located in the project root directory. The script *hyperparams.py* located in the root folder is used to perform hyperparameter search. To see available options for the scripts, run

```
poetry run python <script> --help
```

Hyperparameter optimization can be run for example using the following command:

```
poetry run python hyperparams.py --dataset mnist
                                 --net mnistnet
                                 --bayes vgdo
                                 --n_train 20
                                 --n_val 10
```

The following is an example of a command which can be used to run the active learning experiments:

```
poetry run python main_active.py --dataset cifar
                                 --net simplenet
                                 --bayes dropout
                                 --function margin
                                 --n_initial 10
                                 --n_val 25
                                 --max_labelled 5000
```

The experiment measuring training and prediction time can be run using the following command:

```
poetry run python main_time.py --dataset mnist
                               --net mnistnet
                               --bayes bbb
                               --bs_train 32 128
                               --bs_pred 32
                               --n_batches 5000
```

A.1.3 Project structure

In this section, we describe the structure of the project. The project is organized in four packages. Additionaly, the directory *notebooks* contains notebooks used to explore the results of the experiments. To view the notebooks, use the following command to start a Jupyter notebook server and open a Jupyter notebook client in your browser

```
poetry run jupyter notebook
```

Next, we briefly describe each of the four packages the project consists of.

Active learning

The package *active_learning* contains an implementation of pool-based sampling active learning and the query functions used in our experiments.

Modules

The package *modules* contains PyTorch modules, which implement the non-standard layers used in our experiments. It contains three (Python) modules:

- The module *bbb* contains Bayes by Backprop layers without the local reparameterization trick.
- The module *bbb_lrt* contains Bayes by Backprop layers with the local reparameterization trick.
- The module *vgdo* contains implements Variational Gaussian dropout.

Models

The package *models* contains the models used in our experiments. It contains four sub-packages:

- The package *bbb* contains models using Bayes by Backprop (using the local reparameterization trick)
- The package *determ* contains the non-Bayesian standard dropout models.
- The package *dropout* contains Monte Carlo dropout models.
- The package *vgdo* contains model using Variational Gaussian dropout.

Utils

The package *utils* contains various utilities for training models. The notable modules in this package are:

- The module *train* contains wrappers for training Bayesian PyTorch models and implements the stopping convergence criteria.
- The module *elbo* implements the Evidence Lower Bound loss used in variational inference.

- The module *hyperparam_provider* loads and provides model hyperparameters for use during training.
- The module *logging* supports saving of experiment results while the experiments are running.