Active Learning for Bayesian Neural Networks with Gaussian Processes

Lukas Erlenbach

Born 01.08.1994 in Nastätten

October 28, 2020

Master's Thesis Mathematics

Advisor: Prof. Dr. Jochen Garcke

Second Advisor: Prof. Dr. Martin Rumpf

Institut für Numerische Simulation

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

Notation

Symbol	Meaning
w.r.t.	with respect to
i.i.d	independent and identically distributed
$\mathcal{X} \subset \mathbb{R}^n$	input space
$\mathcal{T} \subset \mathbb{R}^d$	target space, often $d = 1$
$\mathcal{D} = (X, Y)$	training data with input X and labels Y
$f_{true}(.)$	true underlying function of a learning problem
$\hat{f}(.)$	approximation function or model within the solution process
$\mathcal{L}$	loss function
X_{pool}	pool for an active learning problem
$A(x\|\mathcal{D}, \hat{f})$	acquisition function in the greedy active learning framework
$\sim, \propto$	distributed according to, proportional to
$\mathbb{E}\left[x\right], \mathbb{E}_{q(x)}\left[x\right]$	expectation of x, expectation of x, when $x \sim q(x)$
$\mathcal{N}(\mu, \sigma^2)$	normal or Gaussian distribution with mean μ and variance σ^2
$\mathcal{GP}\left(m(.), k(.,.)\right)$	Gaussian Process
L	number of hidden layers of a neural network
w_l	width of layer $l \in \{1, \ldots, L\}$ of a neural network
W^l, b^l	weight matrices and biases of a neural network
$\phi(.)$	non-linearity applied in a neural network
$x \in \mathcal{X}$	input point
$x^l(x) \in w_l$	computed value at intermediate layer
θ	parameters of a (Bayesian) neural network
$f_{nn}(x, \theta)$	forward pass of input x through a neural network with parameters θ
$p(\theta)$	prior for the network parameters
$p(\theta\|\mathcal{D})$	a posteriori distribution of the parameters
$p(Y\|X, \theta)$	likelihood of training data given parameters θ
$p(Y\|X)$	marginal likelihood of the training data
η	variational parameters
$q(\theta\|\eta)$	variational distribution of the network parameters
$\mathrm{KL}\left[p\|\|q\right]$	Kullback-Leibler divergence of distributions p and q
$\mathcal{F}(\mathcal{D}, \eta)$	evidence lower bound

Acknowledgments

I want to express gratitude towards my advisor Prof. Dr. Jochen Garcke for suggesting the topic of this thesis and for providing valuable feedback and guidance over the last year. Furthermore, I want to thank Prof. Dr. Martin Rumpf for being the second advisor. Thanks to Meike, Janis, Mark, Lauritz, Marc and Joachim for proofreading the earlier versions of this work. Furthermore, I am grateful for finding friends within the EFG which made my last six years of studying mathematics in Bonn possible and special. Finally, I thank my family and Melanie for their ongoing support and encouragement.

Chapter 1

Introduction

More often than not, machine learning models require large amounts of labeled data for a successful training process. This is not a problem, if training instances are already available in sufficient numbers or if additional data can easily be obtained. There are, however, also applications where data acquisition is more difficult and where the available data becomes a valuable good which determines and limits the performance of employed models. Typical examples of this problem include image-based cancer diagnosis and speech recognition but also engineering or modeling-related tasks. Often, *unlabeled* data (e.g. images) is readily available but human experts or costly computations are required to get the corresponding *label* (e.g. [cancer] or [no cancer]). Machine learning models, such as neural networks, are highly optimized to extract as much information as possible from the data presented to them. If the performance is unsatisfactory for the particular application, the only way to improve it is to increase the number of training instances.

The purpose of *Active Learning* techniques is to guide this data selection process called *sampling*. In contrast to a *Passive Learner* which only observes the presented instances, an *Active Learner*, usually a pair of a machine learning model and a sampling policy, also interacts with *unlabeled* data. Based on the trained model and the already seen labeled data, the sampling policy decides which instances should be labeled and added to the training data to improve the performance of the model. Compared to generating additional labels for randomly chosen points, a successful active learning framework can exponentially reduce the number of required labels. This has a tremendous impact on the before-mentioned applications: thousands of working hours of medical professionals and other human experts could be saved without compromising any quality of the machine learning models.

The main motivation and guideline for this work is given by Tsymbalov et al. [Tsy+19]. They established a pool-based active learning strategy which is tailored towards neural networks and regression problems. Although neural networks have arguably become the most popular machine learning model, active learning for neural networks and regression problems is currently not very well developed. This highlights the importance of the authors work.

Before the *Active Learning Problem* is introduced more formally in chapter 2, we motivate the topic using an example for active learning on a simple 1d-regression task in the remainder of this chapter. The strategy developed in [Tsy+19] originated from the connection between neural networks and Gaussian processes which we shortly present in chapter 3. In chapter 4 we will extend the neural network to a *probabilistic* model with the introduction of Bayesian neural networks and show how they can be trained with the *Bayes by Backprop* algorithm from [Blu+15]. The application to active learning follows in chapter 5. After connecting neural networks and Gaussian processes (based on [Lee+17]) the sampling strategy from [Tsy+19] is presented and extended with a more general updating procedure. Finally, we test the presented approach extensively in chapter 6 before concluding and providing an outlook in chapter 7.

To illustrate the active learning problem, consider the following toy example of a noisy function which has a minor slope and a sine oscillation in the center as shown in figure 1.1. In the pool-based active learning setting there are two datasets. The **training data** consists of *labeled* points (black dots) which are used for model training. The **pool data** is initially *unlabeled* (gray dots). In the toy example the training data consists of ten randomly selected instances and 1,000 pool points. After training on the initial training data, active learning operates in iterations: additional points from the pool are selected to get labeled and added to the training data. The model is then retrained on the increased dataset. Choosing (or *sampling*) points from the pool that contribute positively to the training process of the model is the main challenge in the active learning problem.

In the following we will compare two sampling strategies and see that a smart strategy can improve the trained model significantly compared to a random selection of points.

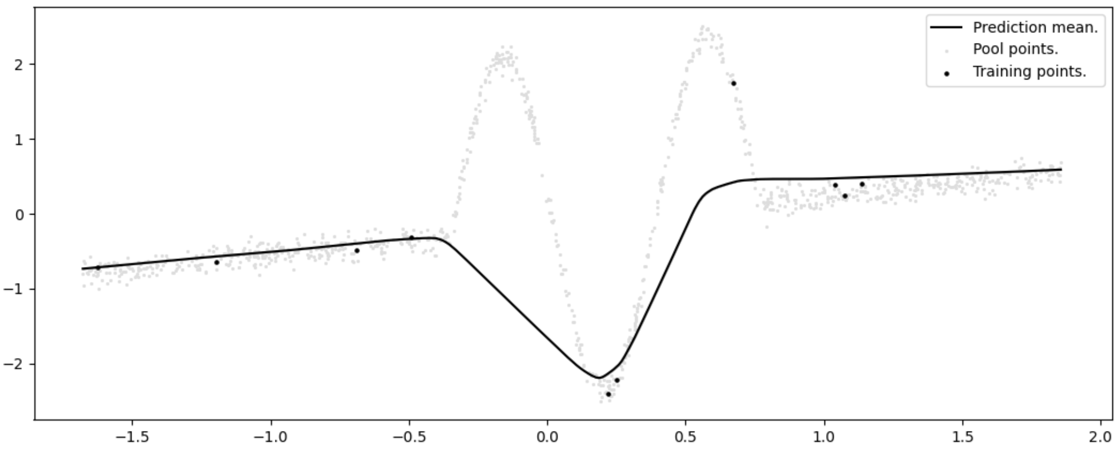

Figure 1.1: Initial training points in black, pool points in gray. Neural network prediction after the first training is shown as the black line.

As model we implement a simple feed-forward neural network in TensorFlow ([Mar+15]) which has two dense layers of 100 neurons each and train it on the ten initial training points. The predictions from this network can be seen in figure 1.1. In this toy active learning framework we allow the selection of five unlabeled instances in each active learning iteration. These points get labeled and then added to the training instances before the network training is continued. As this is only a toy example, we can easily provide the labels for the pool data but keep in mind that the labeling in an application would be done by humans or computer simulations and the corresponding costs could be high.

Because the algorithm does not know the labels of the points in the pool, it is not clear which instances should be added to improve the network. As a baseline sampling strategy we select five points at random. After the addition of these points the network is re-trained on the now 15 labeled instances. The predictions of the resulting network can be seen in the upper plot of figure 1.2: The instances which were selected in the **first** active learning iteration are circled and the points chosen for the **second** iteration are marked with a cross. Observe that the predictions are close to the labels of the training points but in general the improvement after adding five points is small.

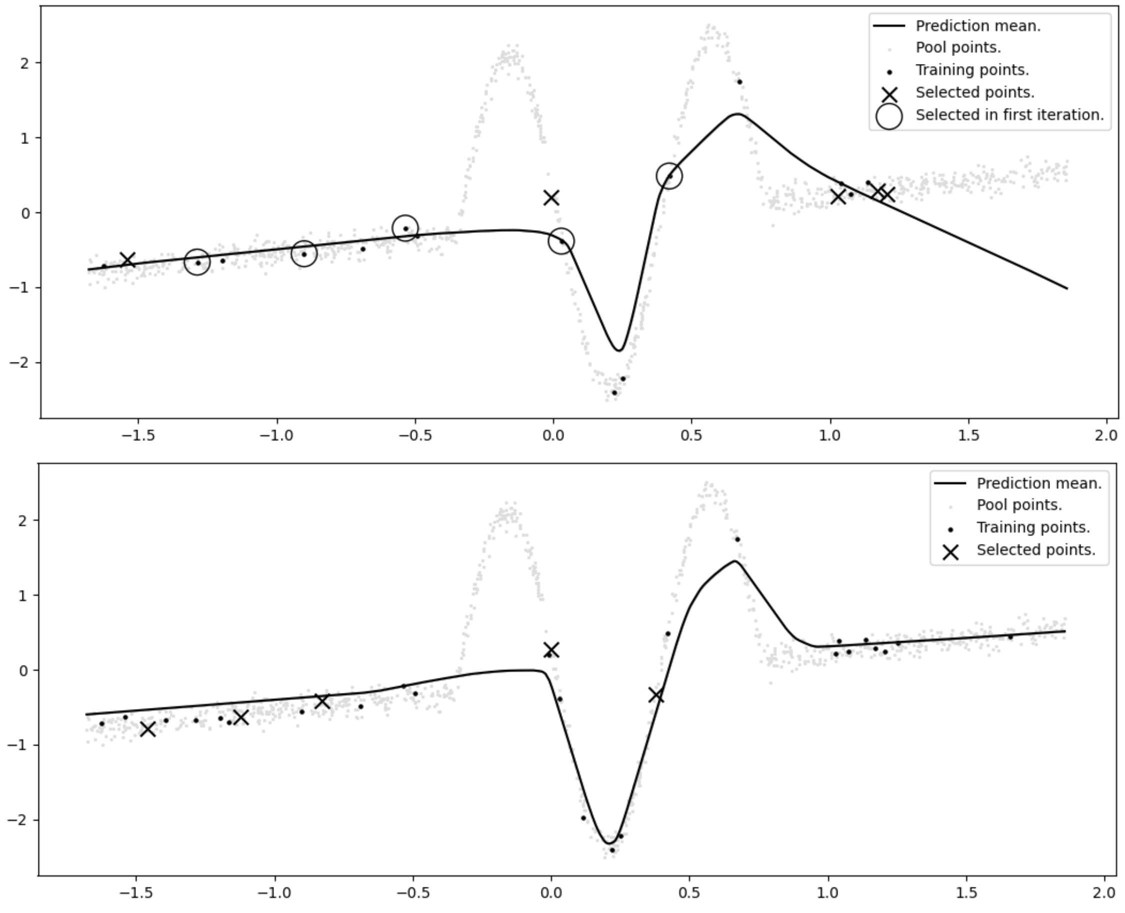

Figure 1.2: Network prediction after training on 15 randomly selected points (upper plot) and on 25 points (lower plot). Points selected in the first active learning iteration are circled, points selected in iterations two and four are marked with a cross.

We perform two more active learning iterations, each selecting five more points at random, add them to the training set and retrain the network. After a total of three iterations the training data consists of 25 labeled points. The predictions of the network can be seen in the lower plot of figure 1.2. Because of bad luck, almost no points from the two peaks of the underlying function were selected and labeled. This exactly constitutes the problem of selection additional data points randomly. Without knowing about these peaks the network can not predict them well and, as a result, the network approximation of the function is far from optimal.

To showcase how a *successful* active learning technique can improve upon the random selection, we start again from the same initial network as shown in figure 1.1. More precisely, we use the GPA Sampler from section 5.3 to choose points from the pool. This sampling method relies on uncertainty estimations of the network output to select instances having a particularly high variance, while also ensuring that the points are well spread.

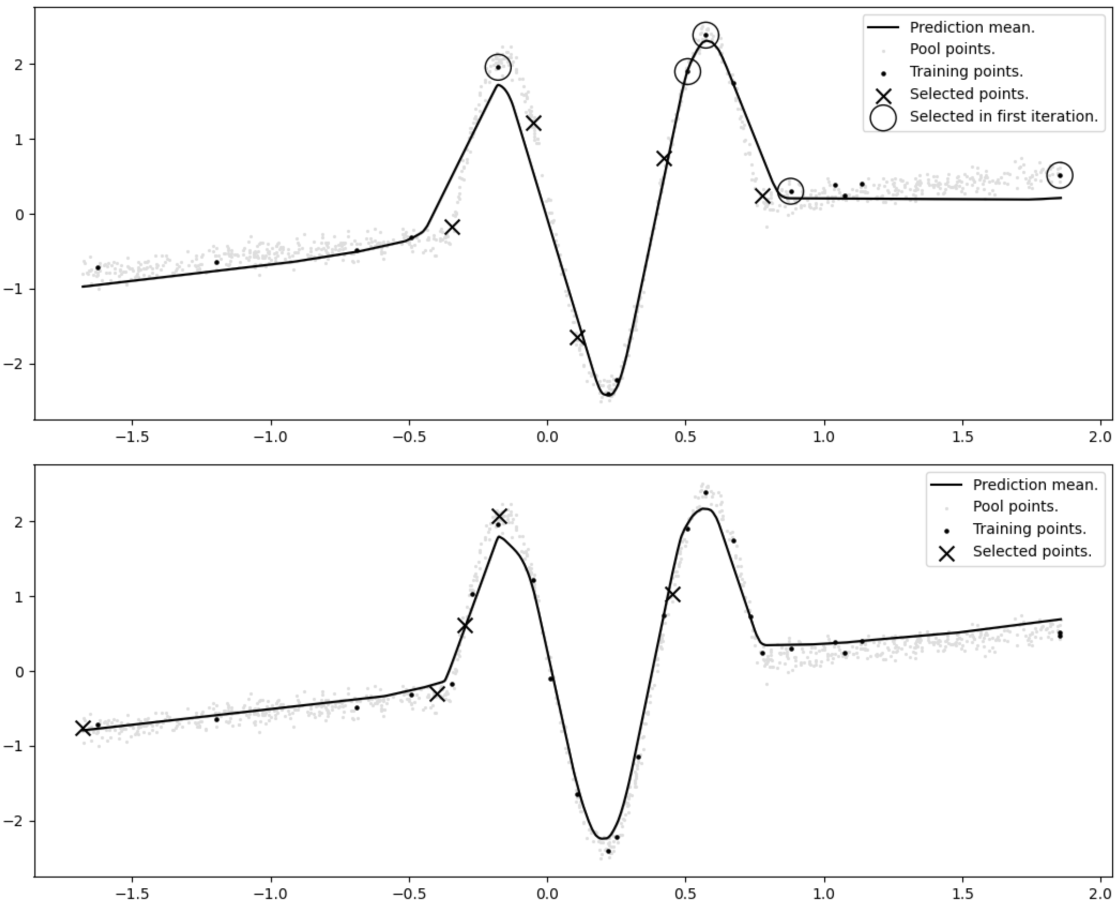

Figure 1.3: Network prediction after training on 15 (upper plot) and on 25 actively selected points (lower plot). Points selected in the first active learning iteration are circled, points selected for the **next** training are marked with a cross.

The five points selected by the GPA Sampler are circled in the upper plot of figure 1.3. It can be seen that the majority of chosen points in the *first* iteration is located in the central region and that re-training the network on 15 points clearly improved the approximation quality of the network. Additionally, the points selected in the *second* active learning iteration (crosses in the upper plot) are still situated in the center. This indicates that the GPA Sampler still predicts a high network uncertainty in that area and therefore decides for more points from there.

Looking at the lower plot of figure 1.3 we see that after three active learning iterations and 15 selected points (25 training points total), the network fits the underlying function reasonably well and certainly better than its counterpart with the randomly selected points in figure 1.2. We are not restricted to just looking at the network predictions but can also calculate their respective loss on the remaining pool instances: After training of 25 points and a total of 10,000 training epochs, the first network with randomly selected points has an RMSE of 0.66 while the network with points chosen by the GPA Sampler has an RMSE of 0.22; a significant improvement.

For a final note on the toy example, we show both networks side by side after five active learning iterations in figure 1.4. The RMSE of the actively trained network is now at about 0.17, whereas we observe no improvement at all in the case of random selection. Of course, this toy example is far from doing complex prediction tasks such as cancer types in skin images but it illustrates well the underlying principles and the possible advantages of using an active learning approach over passive learning. We will now continue in the next chapter with a more formal definition of the *Active Learning Problem*.

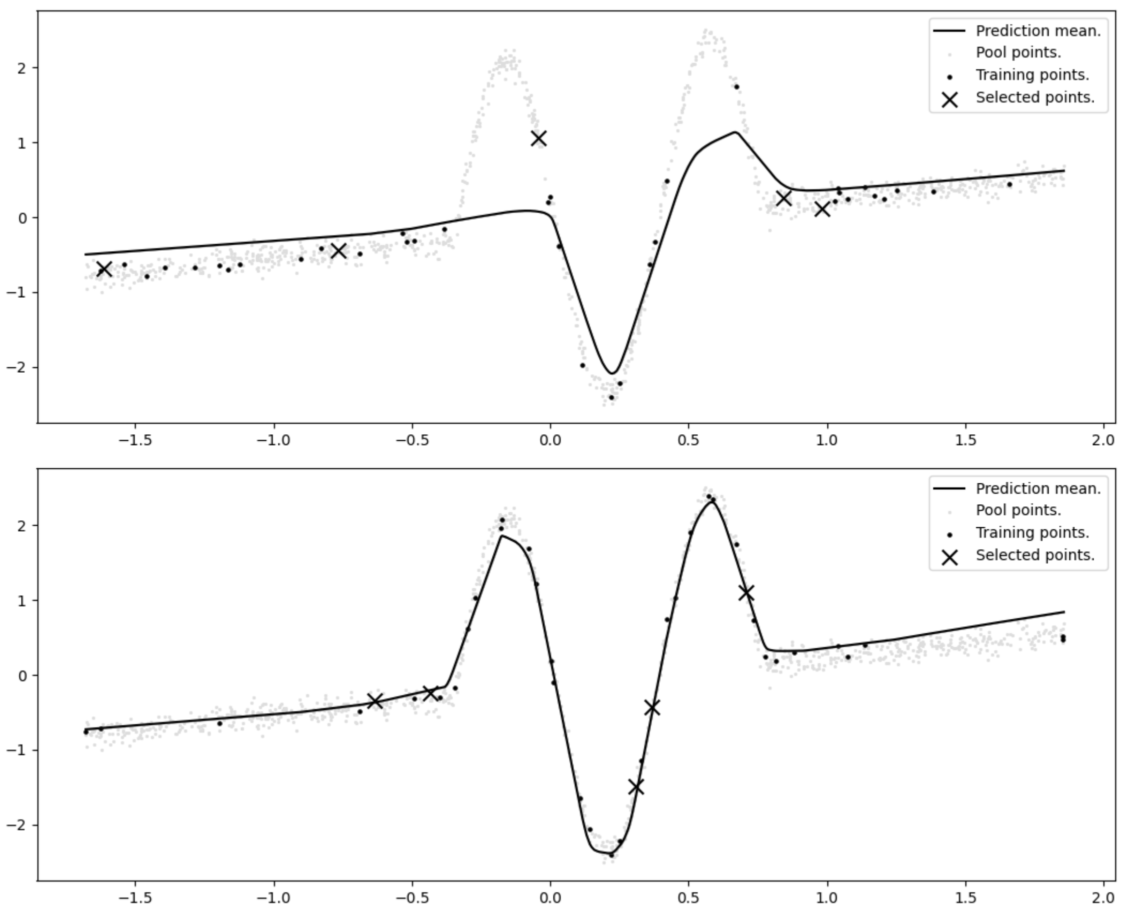

Figure 1.4: Both networks after five active learning iterations, i.e. with a total of 35 training points and 15,000 epochs trained total. Network with randomly selected points on the top, network with points chosen by the GPA Sampler at the bottom.

Chapter 2

Active Learning

In this chapter we will now formally introduce the *Active Learning Problem* which we want to tackle in the upcoming chapters. In general, active learning refers to algorithms which proactively make decisions on additional data acquisition for training purposes. This is useful in supervised learning scenarios when additional training instances can be obtained at runtime but also when the available data is too big to be used as a whole and the training process has to take place on a subset.

We will first be presenting the *Supervised Learning Problem* and then extend it to a broad definition of an *Active Learning Problem* (2.1). The most common solution approach via greedy point selection will be introduced in a solution framework (2.1) before we give an overview on literature for active learning problems (2.3).

2.1 From Supervised to Active Learning

Supervised Learning is - broadly speaking - the machine learning task of inferring a function from labeled data that generalizes well to new unseen data. This is usually achieved by some kind of training process which determines the parameters of a suitable model and ideally, the resulting model can be used to predict the labels of previously unseen input data reasonably well. It is called *supervised* because the access to already annotated instances gives the learning algorithm feedback on how consistent the current models predictions are with the true labels.

Formally, we are given a **labeled dataset** $\mathcal{D} = (X, Y) = \{(x_i, y_i)\}_{i=1}^{|\mathcal{D}|}$ which consists of input-output pairs, where the input stems from an **input space** $\mathcal{X}$ and the label from a **target space** $\mathcal{T}$. Typically, the input space $\mathcal{X}$ is a subset of some $\mathbb{R}^n$ and the target space $\mathcal{T}$ is the real line of dimension one, so that the input-output pairs $(x, y) \in \mathcal{D}$ consist of an input- or feature vector x and an output value y which we also call label. Furthermore, there is an unknown distribution q over $\mathcal{X}$ and an also unknown **underlying function** $f_{true} : \mathcal{X} \to \mathcal{T}$. We assume that the input vectors of the samples (x, y) in $\mathcal{D}$ are drawn independently according to this hidden distribution and that $y = f_{true}(x)$. (**Note:** This formulation assumes that the given dataset is noise-free. In most cases, it is more realistic to add some kind of noise model to the underlying function, e.g. $y = f_{true}(x) + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma^2)$ like at the end of section 3.2).

Our aim is to analyze the training data $\mathcal{D}$ in order to **infer a function** $\hat{f}: \mathcal{X} \to \mathcal{T}$ which can be used to predict labels in $\mathcal{T}$ for unlabeled data from $\mathcal{X}$. In many cases, the function is inferred indirectly as part of a model that is optimized. In all cases, we want the predicted labels to be "reasonable" in the sense that they should agree well with the underlying function f_{true}. As we do not know f_{true}, we need a measure telling us how far away we expect other functions to be from the true function. This measure is called a **loss function** and is formally described as a function $\mathcal{L}: \mathcal{T} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$. Since the only available information about f_{true} are the training instances, $\mathcal{L}$ will be evaluated on them. For example one could take the squared difference between the prediction and the known label or the negative log-likelihood for a probabilistic model. We summarize:

Supervised Learning Problem

Input: a dataset $\mathcal{D} = (X, Y)$, a loss function $\mathcal{L}: \mathcal{T} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$

Task: find a function $\hat{f}: \mathcal{X} \to \mathcal{T}$ that minimizes the expected loss

$$\mathbb{E}_{q(x)}\left[\mathcal{L}(\hat{f}(x), f_{true}(x))\right]$$

The expected loss can usually not be computed directly because q is unknown and labels are given only for a subset of input points. Therefore, in practice this is often circumvented by splitting the dataset $\mathcal{D}$ into a training and a validation set $\mathcal{D} = \mathcal{D}_{train} \dot{\cup} \mathcal{D}_{val}$. While the training is performed on $\mathcal{D}_{train}$, we approximate the expected loss by computing the loss on the validation set:

$$\mathbb{E}_{q(x)}\left[\mathcal{L}(\hat{f}(x), f_{true}(x))\right] \approx \frac{1}{|\mathcal{D}_{val}|} \sum_{(x,y) \in \mathcal{D}_{val}} \mathcal{L}(\hat{f}(x), y) \tag{2.1}$$

The ability to solve a specific Supervised Learning Problem depends on many factors, for example the access to computational power and supplementary information about the problem domain. The most central and limiting resource, however, is often the training data. It needs to be of good quality and quantity to allow supervised learning algorithms to perform well. The optimal training data should be noise-free and well distributed across the input space but it should not contain too many instances, as this would make learning difficult or at least very time consuming. More often than not, the available data is less than ideal and *active learning* can help in at least two very common scenarios. Firstly, when high quality data can be obtained but at a high price and secondly, when the amount of data is too big.

Consider the example of image classification. Here, training data is obtained by humans or human experts who give labels to the images. In most applications this labeling is of high quality because classifying an image is easy for humans and for more difficult applications, like cancer diagnosis, there are experts who spent years training their classification skills. Yet, a tremendous amount of annotated images is required when training modern neural networks for image classification. The most recent version of Google's Inception-network (v4, [Sze+16]) has about 43 million trainable parameters and was trained on 1.2 million hand labeled images from [Rus+14]. But also for smaller applications it can be valuable to develop a smart strategy for deciding which images are actually required to improve the models performance. After a model is trained on an initial set of images, an active learning algorithm could pick unlabeled images which only then would get labeled by humans. This can reduce the total amount of required labels by a huge margin because only images which actually improve the performance of the trained model are selected. Other examples which fall into the same category of human annotation are speech recognition but also physics or engineering when real life experiments have to be supervised. The same scenario applies when labeling corresponds to the conduction of expensive computer simulations.

Active learning also provides help in a second scenario which is becoming more and more common with the development of cheap and digital storage. If suitable data can be obtained from the internet, the training instances quickly get so numerous that the training of a model becomes computationally demanding or even unbearable. The user then has to decide which parts of the available data she uses for the inference but it is rarely clear how to improve over a randomly selected subset. In an active learning framework we would first train a supervised learning model on a small subset of the available data. Afterwards, the active learning algorithm would proactively decide which additional data is needed to continue the learning process and therefore entail big speed-up opportunities.

The promise of active learning methods is to minimize the number of additionally requested labels while maximizing the approximation quality. Formally, the **Active Learning Problem** is an extension of the Supervised Learning Problem mentioned above. We should note that various formulations for active learning are known (refer to section 2.3 for an overview) but we will only consider the pool-based variant: Besides the labeled training data $\mathcal{D}$ another set of input points from $\mathcal{X}$ is given which we call the **pool** X_{pool}. Candidates from X_{pool} can be selected and are evaluated by a blackbox algorithm $\blacksquare$ (this could be a human, simulation software, a database, ..). The task remains the same as in the supervised learning problem but now the additional evaluations can and shall be utilized to minimize the loss function even further. We summarize:

Active Learning Problem

Input: a dataset $\mathcal{D} = (X, Y)$, a loss function $\mathcal{L} : \mathcal{T} \times \mathcal{T} \to \mathbb{R}_{\geq 0}$, a pool X_{pool}, a blackbox $\blacksquare$ which for $x \in X_{pool}$ computes a $y \in \mathcal{T}$

Task: find a function $\hat{f} : \mathcal{X} \to \mathcal{T}$ that minimizes the expected loss

$$\mathbb{E}_{q(x)} \left[\mathcal{L}(\hat{f}(x), f_{true}(x)) \right]$$

2.2 A Solution Framework for Active Learning

The most common solution framework for the active learning problem is via greedy point selection. A supervised learning model is improved over multiple active learning iterations, each consisting of **first**, classical training on the labeled data and **second**, the query of new labels for data points from the pool (just as in the toy example from the introduction 1). The decision which points from the pool should be labeled is made with help of an **acquisition function** $A(.|\mathcal{D}, \hat{f}) : X_{pool} \to \mathbb{R}$. This function incorporates the information from the current training data $\mathcal{D}$ and current approximation function $\hat{f}$ and expresses beliefs on how beneficial the selection of a point would be. The point in the pool maximizing A is selected for labeling:

$$x_{new} = \arg\max_{x \in X_{pool}} A(x|\mathcal{D}, \hat{f}). \tag{2.2}$$

In many scenarios it would be inefficient to only sample a single point in each iteration, therefore, a fixed number of points with the highest A-values is labeled. The pseudocode of the framework is given in algorithm 1.

Algorithm 1: Greedy Active Learning Framework

```
Data: an Active Learning Problem ($\mathcal{D}, \mathcal{L}, X_{pool}, \blacksquare$), a trainable model, a
      number of points to select $N$
Result: a trained supervised learning model
1 while termination criteria not satisfied do
2     train model on $\mathcal{D}$ according to $\mathcal{L}$;
3     $X_{new} \leftarrow \arg\max_{X_* \subset X_{pool}, |X_*|=N} \sum_{x \in X_*} A(x|\mathcal{D}, \texttt{model})$;
4     $\mathcal{D} \leftarrow \mathcal{D} \cup \{(x, \blacksquare(x))\}_{x \in X_{new}}$;
5     $X_{pool} \leftarrow X_{pool} \setminus X_{new}$;
6 end
7 return: model
```

To really put this framework into practice, a few more details have to looked at more closely. What is an appropriate number of points per iteration? What should be the termination criteria in line 1? And how exactly should the training in line 2 be done? One of the more difficult questions involves the details of line 3 where the framework selects N many points from X_{pool} with highest A values. We call line 3 the (active learning) **sampling policy** according to which we select the new datapoints and call an algorithm performing this selection a **sampler**. The central goal of this thesis it to replace the greedy sampling policy from line 3 with a more sophisticated sub-routine. We finish this chapter with an overview on active learning research.

2.3 Literature Review

As already noted, the active learning problem arises naturally in a wide range of machine learning applications and has therefore attained a decent amount of attention from researchers and users. This thesis mainly discusses a specific niche case, namely a fixed class of models (neural networks) and only regression problems, therefore, this section is dedicated to giving a broader overview over the scientific work that has been conducted on the topic and touches on the ideas and approaches which are used to tackle active learning problems.

Most applications where active learning is used are in fact *not* regression problems but classification problems. This might be caused by the fact that classification data can often be annotated more easily by humans or human experts, while it is more difficult for persons to accurately guess values (think for example about classifying an image of an animal versus predicting the price of a house). Also, a big percentage of applications involves classification of image data which perfectly fits into the active learning problem, as large amounts of training data often need to be generated by humans. With regards to models in general, those which provide more information - like probabilistic models - are more commonly used because they allow for smarter active learning strategies.

We roughly follow the outline of the most recent surveys on active learning [Set09] from 2010 and [Agg+14] from 2014 which give a more complete overview and introduction to the various approaches. Theoretical results are surveyed by [BU16] but only classification problems are covered. Literature on active learning for neural networks specifically will be covered in the next chapter in section 4.2. We will now shortly present different scenarios, general sampling strategies, theoretical results and related research topics.

2.3.1 Problem Formulations

Depending on the specific application the unlabeled instances can be available in different forms which we will briefly review here. We chose to only consider the pool-based variant because it is most common and the other scenarios can often be modified to apply pool-based results.

Query-based: In this scenario the learner is completely free to query any unlabeled instance from the input space. For example, in the context of image learning this would mean that the learner could generate an image and ask a human to obtain the label, i.e. to classify it. There are also variants which only allow the learner to ask for membership to a certain class or for the equivalence of instances. The basic formulation was introduced in [Ang88] and extensively studied in [Ang90] and [Ang04]. Recent research developed polynomial time algorithms for several formal languages [AAF18] and automata [AAF20]. More practical oriented approaches include self-paced learning [TH19] where the learner chooses easier examples in the beginning of the learning process. A recent survey features different query strategies [KG20]. Query-based active learning gives the most power to the active learner, as the choice of unlabeled instances is completely free and offers therefore the greatest potential. A major drawback of this setting is that the requested instances might be hard for humans to classify. For example, if a network generates an image it might not make any sense to a human at all. An example of this problem is reported by Land and Baum in [LB92]. They tried to use active learning for the classification of handwritten letters and the active learner asked the humans to classify images which were completely unrecognizable as letters. This formulation can be reduced to the pool-based version by discretization of the input space. Stream-based active learning problems can sometimes be naively reduced to a pool-based problems by collecting a set of instances from the stream which is then used as pool.

Stream-based: As the name suggest, here, the unlabeled instances are drawn from the underlying distribution and the learner has to decide if they are worth labeling one at a time. This variant is therefore applicable if the unlabeled instances arise in stream-form from the problem setting, or if they are cheap to obtain. This approach is also called *selective sampling* and its formulation was characterized by [ACL90] and [CAL94]. Solution approaches include biased random decisions and fixed thresholds which are used to decide if an instance should be labeled or not. It has been used for applications in the financial industry [Sma+14], for document classification [BBB13] and in combination with reinforcement learning [WCC19]. Theoretical research on stream-based active learning is not as well developed as for the pool or query-based versions. However, [FK16] developed a general framework to convert pool-based algorithms to stream-based version while preserving some theoretical guarantees. An argument for this version - besides certain applications which require it - is that the unlabeled data does not have to be stored and the resulting algorithms therefore require less memory. Yet, it is more difficult to make good labeling decision one by one in contrast to evaluating the complete input space or pool.

Pool-based: This is the most common of the three presented approaches because in many applications a large amount of unlabeled data can be obtained basically for free. Consider again image classification where millions of images are available online or really any data which is stored or available online, such as user data in social networks or audio data used for speech recognition. For a broader spectrum of applications we refer to section 2.3 of [Set09]. The general approach was established by [LG94] where they reduced the amount of required training data for a text classifier by as much as a factor of 500. Following this, most of the research conducted on active learning focused on this setting. We will not go into detail here, but continue with approaches for sampling strategies which have all been used for the pool-based setting. We also refer to section 4.2 for the most recent advances in the context of pool-based active learning and neural networks.

2.3.2 Sampling Strategies

The key question in all active learning scenarios is how to select samples which most benefit the learning procedure of the model. There are various intuitive ideas on which data should be selected and corresponding strategies have been developed and employed. Some of them try to request samples which directly improve the model or represent the underlying population well. The first two approaches argue the point of view that the selected samples should contain as much information as possible. All the strategies boil down to the definition of a corresponding acquisition function which can be used in a framework similar to algorithm 1. In general, a good acquisition function has to balance between different objectives and the choice must depend on the application and desired outcome.

Highest Uncertainty: It is intuitive to label instances for which the current model is uncertain about the correct output. Providing the label for such a sample will at least reduce the uncertainty of that particular instance and - hopefully - also in general. For probabilistic models, uncertainty estimation is often straightforward, yet, for non-probabilistic model more sophistication is needed (for example [TK02] developed a strategy for support vector machines). A model independent approach is to train multiple models and to estimate uncertainty by their disagreement as proposed in [SOS92]. *Query-by-committee* can be used either with the same model and different parameters (as suggested in [DE95] and already used with neural networks in [Fre+97]) or with completely different models (a small number of models can already suffice [SCR08]).
Depending on the problem, different acquisition functions for ranking the unlabeled instances are used. In **classification** many models output a probability distribution over the possible classes. Therefore, it is straightforward to compute the probability of the most likely class and rank the pool according to *least confidence*, i.e. 1 minus the probability of the most likely class. The most popular choice is the *Shannon-entropy* ([Sha48]) which not only considers the most likely class, but also the remaining distribution. For query-by-committee different acquisition functions like the *vote-entropy* ([DE95]) or the *KL-divergence* ([KL51]) are used. For **regression** tasks, the output variance is a precise measure of uncertainty. In the Bayesian setting which we will introduce in chapter 4, the posterior variance of the trained neural network will be available at no extra cost.

Expected Model Change: This was introduced by [SCR08] for active learning of multi instances. The idea is that the newly obtained instances should have a big impact on the current model, i.e. cause a big change to its parameters. The change of the model is measured by the length of the gradient w.r.t to model parameters and therefore requires the gradients to be accessible. Considering an unlabeled instance, its possible labels and respective likelihoods the expected model change can be estimated. This method has performed well in empirical studies (refer section 3.3 in [Set09]) but needs proper rescaling if label classes are unbalanced and can be computationally expensive.

Expected Error or Variance Reduction One can also try to directly improve the objective function of the corresponding supervised learning problem. The important distinction to the uncertainty approach is that not the state of the model *before* the choice of samples is considered but the state *after* new samples are added and the model was retrained. Depending on the loss function the expected loss after the addition of a sample can be estimated (as proposed in [RM01]). This approach represents very precisely what we intend to achieve: selecting samples which lead to a good generalization of the model which we typically measure with the loss function. The crux of the idea lies in the fact that we do not know the labels of the unlabeled instances beforehand and therefore have to guess how the addition of an instance will reduce the error. This can for example be done by retraining the model with each possible label and weighting the results with the probability of really encountering that label, however this is computationally expensive and not applicable if the real labeling procedure is not extraordinary expensive.

In [GG07] it was suggested to be optimistic and only consider the label class which minimizes the expected loss. Other approaches which exploit specific model classes and loss functions have also been successful (e.g.[Mos+07]). Another line of research does not deal directly with the objective function but rather aims at reducing the expected uncertainty, i.e. the variance after the addition of the new samples. This is computationally cheaper (for some models the expected variance can even be expressed in a closed form [Mac92]) and one can imagine that reducing the variance of the prediction should also reduce the errors made by the model. In chapter 5 we follow a similar idea and approximate a neural network with a Gaussian process because the posterior variance of the latter can be calculated in a closed form. We will also estimate the variance of the model after the addition of samples but in this case only for the selection of more samples without the need of retraining the network.

Density-based and Hybrid. Focusing on the individual instances and choosing samples which maximize an acquisition function can run into problems by requesting labels for outliers which are unrepresentative for the underlying distribution. This can be avoided by weighting the primal acquisition function with a second function which measures the *representiveness* of an instance. The second function can be realized by different similarity measures like distances or via clustering; we refer again to [SC08] for more details. And finally, all of the presented approaches can be combined to form all sorts of hybrid strategies.

2.3.3 Related Topics

Different well known Machine Learning problems are similar in spirit as they also require a good learner to proactively make use of unlabeled data. More recently, there have been attempts to use ideas from active learning research and apply them to related areas (and vice versa) such as active semi-supervised learning [RKG18], active transfer learning [Den+19] and active reinforcement learning [EVD08], [FLC17], [WCC19].

In **Semi-Supervised Learning** not only the labeled dataset is provided but additional, unlabeled data is given. Additional labels cannot be obtained, but the the unlabeled data can be used to draw conclusions about the underlying data distribution in the input space. For example identifying different cluster within the data can make predictions a lot easier. The approach presented in section 5.3 will also make use of the unlabeled data to sample more diverse points.

In **Transfer Learning** the task is similar to supervised learning, yet additionally labeled data from another somehow related dataset is given. It has been used successfully with neural networks in distinct applications. Again, a successful learner has to draw conclusions from data which can not be used in the traditional way as it might be unlabeled and even stem from a different domain.

Reinforcement Learning deals with the problem of learning how to move an agent through a "world" via actions to reach states which provide certain rewards.

In the beginning, the learner neither knows in which states it can expect a positive reward, nor how to get there. Similar as in active learning, a good learning strategy therefore has to make smart decisions on how to explore the space without being sure about the outcomes of its actions.

2.3.4 Theoretical Results

To exemplify the enormous potential that lies in active learning we will start with a commonly used example ([Set09], [Agh+19]) before reviewing the rigorous theory which has been developed for active learning.

Assume a 1-d example of an unknown binary threshold function $f_\alpha(x);\ \alpha \in [0,1]$ which is 1 if $x > \alpha$ and 0 otherwise and that we want to learn this function up to an error of $\epsilon > 0$. If we draw random samples $x^* \in [0,1]$, we have to evaluate $\mathcal{O}(1/\epsilon)$ times on average (e.g. by hitting both the intervals $[\alpha - \epsilon/2]$ and $[\alpha + \epsilon/2]$) to get to an error of ϵ. Regard the following active learning strategy: Consider the points $\{0, 1\epsilon, 2\epsilon, \ldots, 1\}$. We know that the point before α evaluates to 0 and the point after α to 1, therefore, we can conduct a binary search to find the transition point in only $\mathcal{O}\left(\log(1/\epsilon)\right)$, an exponential reduction of evaluations! Of course this is only a simple toy example but the great promise of active learning (and declared goal of theoretical research) is to achieve similar reductions also in multi-dimensional, multi-class, noise-corrupted and more realistic settings.

In general, theoretical guarantees for active learning algorithms are given within the *probably approximately correct* (PAC) learning model [Val84] which requires the learner to achieve an error of at most ϵ with a high probability. Results typically include v, the *Vapnik-Chervonenkis (VC)-dimension* [VC71] as a factor which measures the complexity of the model space and the dimension of the input space d.

After a negative result [ER90] which showed that membership queries are essentially useless for a large set of model classes, an exponential reduction (from the standard $\mathcal{O}(d/\epsilon)$ to $\mathcal{O}(\log(d/\epsilon))$) was proven in [Fre+97] using a committee of networks under the assumption that the underlying function is drawn according to a known distribution. For learning half-spaces with networks the same reduction of queries was shown by [DKM05] without the need for multiple models but with an improved updating rule. For the pool-based version lower and upper bounds were provided in [Das05] covering several worst cases which can require $\Omega(1/\epsilon)$. But later it was shown in [BHV10] that these worst cases can almost always be avoided.

The beforementioned results assume that a perfect labeling can be realized with the specified model class. As this often is an unrealistic assumption (model classes are very small compared to the space of all possible function and do only rarely include noise), the PAC learning theory can be extended to the **agnostic** case which no longer requires a maximum error of ϵ to be achieved but an error that is at most ϵ worse than the best possible error within the model class. A first algorithm with exponential gains in some settings was presented in [BBL09] and later improved and extended to more scenarios in [DHM08] where they reduced the problem to special kind of supervised learning. More recent results [ABL18] also consider bigger noise rates of $\Omega(\epsilon)$.

For **regression**, the active learning theory is less well developed and convergence rates are only known for specific problem, data, model and noise settings. For constant noise, convergence rates for two function classes could be proven in [WNC06]. In [Cha+15] active learning for maximum likelihood estimation was considered. It was general enough to include regression problems but could only prove a constant factor improvement over passive learning. No rates, however a connection between the *fisher-information-ratio* (FIR) and the expected variance of a log-likelihood was shown in [Sou+17]. For a least-square loss function [CP19] gave a framework which requires a near minimal number of samples. For instance-dependent noise [CJN17] used a two-stage approach: first they derived an oracle with guarantees based on the true, unknown underlying statistics and in a second step they approximated this oracle in an active learning scheme.

Chapter 3

Gaussian Processes

In this chapter we will introduce Gaussian processes (GP), a machine learning model which is commonly used for regression tasks and popular in active learning applications and research because of its probabilistic nature. Gaussian processes do not scale well with larger sizes of data, therefore we will not train them as a model in this work but rather approximate a GP to smartly select additional points in an active learning framework which trains a neural network.

In the first section we will to define the notion of a *Gaussian process* and in the second section we review the properties which will be utilized later. We assume that the reader is familiar with the fundamentals of probability theory like distributions of random variables and probability density functions. For an introduction to this field we refer to [BB11] and follow the notation from the book *Gaussian Processes for Machine Learning* by Rasmussen and Williams ([RW06]).

3.1 Gaussian Processes - Definition

A **Gaussian** or **normal** distribution refers to a set of continuous probability distributions for real-valued random variables which have a particular form of density function.

Definition 3.1.1. *A random variable x is said to follow a (univariate)* ***Gaussian-distribution*** *if its probability density function takes the form:*

$$f(x|\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{3.1}$$

When a random variable x follows a Gaussian distribution with parameters μ and σ^2, we write $x \sim \mathcal{N}(\mu, \sigma^2)$ or $\mathcal{N}(x; \mu, \sigma^2)$ for short. The parameters μ and σ^2 are the expectation and the variance of the distribution and determine it completely. The importance of this particular distribution stems from the fact that it naturally arises as the limit of the mean of samples even if the samples are not drawn from a normal distribution themselves. In probability theory this can be formalized as the well known **Central Limit Theorem**.

Theorem 3.1.2 (Central Limit Theorem (CLT))**.** *Let $\{X_1, \dots, X_k, \dots\}$ be independent and identically distributed (i.i.d.) random variables with mean μ and variance σ^2. Then the normalized sample mean $\bar{X}_k := \sqrt{k}\left(\frac{X_1+\cdots+X_k}{k} - \mu\right)$ converges in distribution to $\mathcal{N}(0, \sigma^2)$ as k approaches infinity.*

We can also generalize the notion of a Gaussian-distribution easily to the multivariate case where the random variables are vectors.

Definition 3.1.3. *A multivariate random variable $X \in \mathbb{R}^n$ is called* ***multivariate Gaussian distributed*** *if every linear combination of its components $\sum_{i=1}^{m} a_i X_i, a \in \mathbb{R}^n$ has an univariate Gaussian distribution. For such a random vector we write $X \sim \mathcal{N}_n(\mu, \Sigma)$ where μ is a vector in $\mathbb{R}^n$ and Σ a matrix in $\mathbb{R}^{n,n}$.*

We will still call μ the mean of the distribution $\mathcal{N}_n(\mu, \Sigma)$, while Σ is its covariance matrix which contains the variances on the diagonal and the covariances off the diagonal. Note that μ and Σ still determine the distribution completely. Unsurprisingly, the CLT can also be generalized to the multidimensional case.

Theorem 3.1.4 (Multivariate Central Limit Theorem (MCLT))**.** *Let $\{X_1, \dots, X_k, \dots\}$ be independent and identically distributed random vectors in $\mathbb{R}^n$ with common mean vector $\mathbb{E}[X_1] = \mu$ and covariance matrix Σ. Let the normalized sample mean be calculated component wise, $\bar{X}_k := \frac{1}{\sqrt{k}} \left(\sum_{i=1}^{k} X_i - \mu \right)$. Then $\bar{X}_k$ converges in distribution to $\mathcal{N}_n(0, \Sigma)$ as k approaches infinity.*

Abstractly speaking, a random vector $X = (X_1, \dots, X_n)$ of length n is the collection of n random variables which are indexed by the set $I = \{1, \dots, n\}$. From this vantage point, the extension to infinite random vectors is easily realized by taking a collection of random variables $\{X_i\}_{i \in I}$ together with the index set $I = \{1, \dots\} = \mathbb{N}$. The idea of a stochastic process takes this generalization one step even further, namely to arbitrary index sets.

Definition 3.1.5. *A real-valued* ***stochastic process*** *f is a collection of real-valued random variables, indexed by some set I. It can be written as:*

$$\{f(i) : i \in I\}. \tag{3.2}$$

Historically, the index set I was often interpreted as time, therefore one might think of $f(i)$ as a random variable that is observed at a certain point in time. For our purposes however, we are not interested in processes which evolve over time, but over the input space of the considered problem. In other words, we will use the input space $\mathcal{X}$ as index set such that the process gives a random variable for each input point $\{f(x) : x \in \mathcal{X}\}$. This is motivated by the following, explained in more detail in [RW06].

When talking about supervised learning in chapter 2 we wanted to use a given training dataset to infer a function $\hat{f}$ which can be used to make predictions for so far unseen input values. We can only verify the performance of $\hat{f}$ at the datapoints for which we know the correct labels. But not every function which is consistent with the training data is equally likely to be an useful approximation. For example, the function which predicts only the known labels and is zero otherwise is also perfectly consistent with the training data and completely useless. Therefore, one might prefer functions which are expected to be a good approximation of the true underlying function, for example functions which are smoother in some sense.

Such assumptions are often realized by restricting the space of possible functions $\hat{f}$ to a certain class of functions, e.g. combinations of basis functions. This has the advantage of directly reducing the tremendous complexity of the search space (the space of all functions from input to output space). However, this approach comes with the burden of finding an appropriate class of functions: If chosen too small the underlying function can not be modeled well, and choosing it too large may lead to overfitting.

To avoid this pitfall we will use a stochastic process indexed by the input space, which gives a distribution over *all* possible functions. We want to assign higher probabilities to functions which fit the training data and prior assumptions. Lower probabilities or even a probability of zero should be assigned to those not fitting well. Inferring a "good" stochastic process from the training data might seem computationally difficult because the space is possible functions is very large. But luckily it turns out that with the notion of *Gaussian processes* the necessary operations can be done on a finite amount of points. We finish this section with the formal definition of a Gaussian process and will continue in the next section with the practical usage of them.

Definition 3.1.6 (Gaussian Process)**.** *A stochastic process $\{f(x) : x \in \mathcal{X}\}$ is called **Gaussian** if and only if for every finite set of indices $x_1, \ldots, x_k \in \mathcal{X}$, the random vector $(f(x_1), \ldots, f(x_k))$ is multivariate Gaussian distributed.*

A Gaussian process f is completely determined by its first two moments, i.e. the mean and the covariance function which is also called the *kernel*.

$$\begin{aligned} m(x) &:= \mathbb{E}\left[f(x)\right] && (3.3) \\ k(x, x') &:= \mathbb{E}\left[(f(x) - \mathbb{E}\left[f(x)\right])(f(x') - \mathbb{E}\left[f(x')\right])\right] && (3.4) \\ &= \mathbb{E}\left[(f(x) - m(x))(f(x') - m(x'))\right] \end{aligned}$$

Therefore, we will use the notation

$$f(x) \sim \mathcal{GP}\left(m, k\right) \tag{3.5}$$

and also assume $m(x) = 0$ for simplicity from here on as the case $m(x) \neq 0$ can be derived from the centered case.

Note that for a given covariance function $k(.)$, the existence of the Gaussian process $\mathcal{GP}\left(0, k\right)$ is guaranteed by Kolmogorov, since the entries of the covariance matrix are defined by $k(.)$ and therefore automatically consistent.

3.2 Gaussian Processes - Predictions

Given a Gaussian process $\mathcal{GP}\left(0, k\right)$, there are perspectives one can consider and which turn out to be mathematically equivalent. First, we can imagine that for an input value $x \in \mathcal{X}$ the Gaussian process at that index $f(x)$ is a distribution over the possible outputs in the output space. (This is already helpful because a distribution contains more information than a point estimate.) Moreover, one could go the other way around, set the input values aside for a moment, and first sample a *function* $\hat{f}$ which can then be evaluated deterministically for input points. From this perspective, the Gaussian process actually specifies a distribution over the function which goes from input to output space.

For applications we typically are interested in a function which evaluates only a finite subset of the input space, say $X_* \subseteq \mathcal{X}, |X_*| < \infty$. In this case we can list all the covariances of X_* in a matrix $K(X_*, X_*)$ and define a Gaussian process for this finite set which is also called the **prior** distribution.

$$\begin{aligned} K(X_*, X_*) &:= [k(x_i, x_j)]_{x_i, x_j \in X_*} \\ f(X_*) &\sim \mathcal{GP}\left(0, K(X_*, X_*)\right). \end{aligned} \tag{3.6}$$

Now $f(X_*)$ is a vector which is Gaussian distributed and depends on the covariance function $k(.)$ and X_*. This vector contains one random variable for each $x \in X_*$ and we could already sample them to make predictions which however would be of little use, as there is no information about the training data contained so far. Of course the choice of the particular $k(.)$ is also essential and usually needs to be optimized as well but we will see later that this does not apply to the way we want to employ Gaussian processes.

Assume now we have training data $\mathcal{D} = (X, Y)$ and new input points X_* which we want to make predictions for. Assume also that we have decided for a particular covariance function $k(.)$. According to $k(.)$, we can write down the joint distribution of known labels Y and the unknown labels $f(X_*)$ in block vectors and matrices.

$$\begin{bmatrix} Y \\ f(X_*) \end{bmatrix} \sim \mathcal{GP}\left(0, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{3.7}$$

But the really interesting distribution appears when we *condition* $f(X_*)$ on the known values (X, Y)

$$\begin{aligned} f(X_*)|(X,Y) \sim \mathcal{GP}(&K(X_*,X)K(X,X)^{-1}Y, \\ &K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*)). \end{aligned} \tag{3.8}$$

We call this the **posteriori** distribution and its calculation only needs the evaluation of the covariance function $k(.)$ and matrix operations. The functions drawn from the posterior distribution will still depend on the kernel, yet they will be consistent with the training data. Again taking the other perspective of obtaining distributions per input: if (x, y) is a training instance then $f(x)$ will be the point estimate y and if $x^* \in X^*$ is not known then $f(x^*)$ will be a normal distribution. Ideally, $f(x^*)$ will be closer to a point distribution if the output can clearly be inferred from the training data and be flatter with a bigger variance if the correct output seems unclear. The posterior can still be utilized to make deterministic predictions using the maximum likelihood principle. Nevertheless, by predicting distributions, much more expressiveness is obtained compared to non-probabilistic models.

Generally, it is not realistic to assume noise free data as we have done so far. To go one step further we derive a similar formulation under the assumption of noise which effects the observed values. With additive independent identically distributed Gaussian noise with variance σ^2 the covariance matrix of the training instances changes on the diagonal (and only there because of the independence of the noise):

$$\begin{bmatrix} y \\ f(X_*) \end{bmatrix} \sim \mathcal{GP}\left(0, \begin{bmatrix} K(X,X)+\sigma^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \tag{3.9}$$

and the posterior becomes

$$\begin{aligned} f(X_*)|X,y \sim \mathcal{GP}(&K(X_*,X)\left[K(X,X)+\sigma^2 I\right]^{-1} y, \\ &K(X_*,X_*) - K(X_*,X)\left[K(X,X)+\sigma^2 I\right]^{-1} K(X,X_*)). \end{aligned} \tag{3.10}$$

As previously mentioned we will not discuss possible kernel choices and parameter optimization which are needed to use this formulation in the context of machine learning problems. Instead we note a major disadvantage of Gaussian processes which is the runtime of $\mathcal{O}(|\mathcal{D}|^3)$ for training data $\mathcal{D}$. This stems from the inversion of the non-sparse covariance matrix $K(X,X)$ and is the reason why this approach quickly becomes unbearable for large datasets. There are approaches to speed up the process (often by approximating the covariance matrix like in [MRE16] or [GAS20]), however we will use a completely different model (neural networks) which is known to scale well with big data. We will introduce this model in chapter 4 and then see how it is connected to Gaussian processes in chapter 5.

Chapter 4

Bayesian Neural Networks

In this chapter we will introduce the powerful and well known class of machine learning models called *neural networks*. Inspired by the neurons in the human brain they also consist of small information storage units called *neurons* or *perceptrons* which interchange information with each other. Over the last decade neural networks have proven to be exceptionally powerful tools, advancing the success of machine learning in previous unexpected areas and magnitudes. They have been utilized to tackle issues from classification to unsupervised learning, almost every problem in the world of machine learning. We will utilize them in the most classic fashion which is as function approximation for a regression task.

Neural networks are known to scale well with large data sizes because they can use powerful graphics processing units (GPUs) to accelerate the training process. Research in the last years has developed dozens of tools and extensions to optimize prediction qualities and powerful open-source libraries like TensorFlow ([Mar+15]) and PyTorch ([Pas+19]) enable users to employ neural network models with relative ease. Yet the strong computational powers of neural networks do not come without certain drawbacks. One of the biggest challenges when applying neural networks is the lack of insight on the decision making process of a network. In other words, one might be fairly certain that a trained neural network archives good or even very results with regards to the defined errors and losses, however the network does not express in any way how to get there. In contrast, other classical supervised learning models do come with a natural interpretation: For example in a decision tree the user can clearly follow the paths of decisions taken and comprehend how and with which reasoning the model is arriving at a label. This enables the user to gain a deeper understanding about the problem on one hand, and on the other - as the user can judge the complete decision making process - also provides refined control and intervention options.

One way of getting more useful information out of a neural network is using them in a Bayesian setting as **Bayesian neural network (BNN)**. Similarly to the Gaussian processes from chapter 3, we are not interested in predicting only a single output but a distribution over possible outputs which expresses the beliefs of the network on how likely they are to be the correct label. With the distribution at hand we still have the possibility to predict a single value (e.g. the most likely value) yet it also provides access to other statistical tools. This allows a better assessment of the output, as the distribution describes the certain or uncertainty of the network. To exemplify this, imagine two possible extreme cases: If the network predicts a distribution which has almost no variance, we can be certain that the maximum of this distribution is the correct label. However, the network could predict a distribution with *the same* maximum but with a much higher high variance and maybe even other local maxima. In the second case we would not trust the network as much as in the first case but if the network only predicts the maximum, as we would not be able to distinguish the two cases. This highlights the advantage of predicting a distribution over a single value.

In the context of active learning the probabilistic information that the BNN model proves particularly useful. Knowing where the network is comfortable in its predictions and where it is uncertain we can decide which additional points are needed to improve its performance. At the end of this chapter we will use this observation to present an intuitive sampling policy for a Bayesian neural network which is used as a baseline for the upcoming chapter.

We will proceed in this chapter as follows: After defining the most of the necessary notation along the introduction of the *classical* neural network in section 4.1 we will give a short overview on literature about active learning with neural networks in section 4.2 . Next, in section 4.3, we present how neural networks can be combined with Bayesian inference. Finally, we will see how Bayesian neural networks can be trained efficiently in section 4.4 and finish the chapter by showing how their structure can be utilized to build a variance-based sampler for the active learning framework. We adopt the notation for neural networks from [Lee+17].

4.1 Feed-Forward Neural Networks

In the most basic version a neural network for inputs from $\mathbb{R}^n$ and outputs from $\mathbb{R}^d$ consist of $L \in \mathbb{N}$ hidden **layers** of neurons. We denote the number of neurons in layer $l \in \{1, \dots, L\}$ with w_l and call it the **width** of layer l. For simplicity reasons we also denote $w_0 = n$. For each layer there is a **weight matrix** $W^l \in \mathbb{R}^{w_{l-1} \times w_l}$ and a **bias** $b^l \in \mathbb{R}^{w_l}$ and additional ones for the output layer $W^{L+1} \in \mathbb{R}^{w_L \times d}$, $b^{L+1} \in \mathbb{R}^d$. For the sake of notation we denote the collection of all trainable parameters of a neural network, i.e. the weight matrices and biases with a single variable

$$\theta := \{W^l, b^l\}_{l=1,\dots,L+1}. \tag{4.1}$$

The input $x \in \mathbb{R}^n$ is passed to the first layer by multiplying it with a weight matrix W^1 and adding a bias b^1. A non-linearity or **activation function** ϕ is applied point-wise before the values are passed to the next layer where they serve as the input for the next layer of neurons. After the input is fed through all layers, the values at layer L are multiplied with the final matrix and the final bias is added to form the **output** $f_{nn}(x;\theta) \in \mathbb{R}^d$. Noticeably, for fixed parameters θ the network model encodes the approximation function $f_{nn}(.\,;\theta)$. We denote the **intermediate values** of the network at layer l with $x^l(x;\theta)$. Defining the intermediate values as functions of the input will make it easier to speak about values of the same network for different inputs. If the particular parameters of the network are not important, we will also drop the θ and simply write $f_{nn}(x)$ and $x^l(x)$. Putting this together, a forward pass through the network can be computed as

$$\begin{aligned} x^0(x) &:= x \\ x^l(x) &= \phi(b^{l-1} + W^{l-1}x^{l-1}(x)), \text{ for } l \in \{1, \dots, L\} \\ f_{nn}(x) &= b^{L+1} + W^{L+1}x^L(x). \end{aligned} \tag{4.2}$$

The weight matrices W^l and biases b^l are the degrees of freedom of the network and have to be determined from the training data . The training utilizes the concept of *stochastic gradient decent* as follows. Assume a supervised learning problem $(\mathcal{D}, \mathcal{L})$ as in section 2.1 is given as well a neural network with initial parameters θ. The parameters are updated iteratively by estimating the gradient of the loss function

$$\theta' = \theta - \nu \sum_{(x,y)\in\mathcal{B}} \nabla_\theta \mathcal{L}(f_{nn}(x;\theta), y) \tag{4.3}$$

where $\mathcal{B} \subset \mathcal{D}$ is a random subset of the training data and called a (mini-)**batch** and $\nu \leq 1$ is a small constant also referred to as the **learning rate**. One iteration, or **epoch**, typically consists of splitting the complete dataset into batches and applying the respective update gradients. In the context of neural networks this can be done very efficiently by *backpropagation* where the gradients are first calculated w.r.t to the last layer and then propagated backwards through the network using the chain rule. As this leads to simple but numerous matrix operations, GPUs and parallel computing are successfully employed to speed up the learning process.

This is of course only a very fundamental description of a neural network. Significant improvements can be made by employing extension (like dropout [Sri+14]) and choosing the correct hyper-parameters which include parameter initialization strategy, network architecture, learning rate, batch size, activation function and more. We will specify those when we employ the networks in the experiments in chapter 6 but for now we want to focus on active learning and give a short literature survey on active learning with neural networks.

4.2 Literature Review: Active Learning for Neural Networks

As we have already seen in section 2.3, active learning has been widely studied in machine learning for many years. Nevertheless, the advances of deep learning and neural networks in the recent years require the adaptation of old methods or the design of new algorithms which are tailored towards neural network models. Although neural networks were already considered in early active learning research like [ACL90], we want to focus on the rather recent advances which were developed for modern large scale applications.

One of the most popular and commonly discussed active learning use cases is image classification. Since neural networks and in particular an architecture called *convolutional neural netorks* (CNN) are also known for successes in image related tasks, it only makes sense to try and combine both approaches. Image data is high dimensional and hence deep learning models require big amounts of labeled data to be trained successfully. The initial labeling usually has to be done by humans which is expensive and not easily scalable. Active learning therefore promises to reduce the amount of data that has to labeled by hand to train networks which afterwards can support humans. The well known MNIST ([LC10]) and CIFAR-10 ([Kri12]) datasets are usually considered as a benchmark.

A first mayor contribution was introduced by [Hou+11] in 2011. Originally not intended for neural networks, they derived *Bayesian Active Leaning by Disagreement* (short BALD) for Gaussian Processes. BALD follows the paradigm of greedy point selection as described in section 2.2 and defines an acquisition function which consists of the *current* and the *posteriori expectation* of the entropy of the model prediction. In that sense it is a hybrid of the *uncertainty* and *variance reduction* sampling strategies discussed in section 2.3.2. It was successfully used on image data with Bayesian convolutional neural network by [Gal16] compared with other acquisition functions in [GIG17] and since then is often used as a baseline for further research.

As BALD was originally designed to sample singletons from a pool, it is not optimized to sample multiple points at once (a problem that we will also encounter by the end of this chapter). The intuitive approach of choosing the top k points with the highest BALD values fails because this typically returns many points which similar characteristics and therefore little information. Last year an improved version called *BatchBALD* was published by Kirsch et al. in [KAG19] which addresses the problem by defining an acquisition function not for single points $x \in \mathcal{X}$ but for subsets $X \subset \mathcal{X}$. Since there is an exponential amount of subsets to chose from, they also showed how to efficiently find an approximately optimal set. The same problem was addressed by [SS17] where they formulated the active learning problem as a set cover problem which they approximate using combinatorial optimization. A two step solution approach was proposed by [Zhd19]: First, a bigger set of points with high acquisition values gets selected and then clustered with K-means to finally obtain divers points.

Earlier work was done in [GE17] where they showed on the MNIST and CIFAR-10/100 datasets that active learning indeed reduces the amount of required labels significantly compared to passive learning and random sampling. Ensemble-based approaches came from [Bel+18] and [PF18]. They trained groups of identical networks which were initialized with different parameters and then used in a *query-by-committee* framework (see section 2.3.2). Although this proved successful, training multiple neural networks instead of a single model is computationally quite expensive. The work [YK19] outlines another idea which only requires two networks to be trained. They trained one network to predict the target function and an additional one to predict the loss function. The second network was then used to assess, if the first network is likely to produce wrong predictions and can therefore suggest which points in the pool should be selected.

Outside the classical image classification active learning for neural networks has been used in various settings. Reinforcement learning in the stream-based active learning setting (see section 2.3.1) was discussed by [WF17]. Here, an action-value function is learned and decides at runtime which images should be labeled. Motivated by the task of pedestrian detection for autonomous vehicles, [Agh+19] proposed a framework for active learning for detection networks. [Wan+19] applied active learning for global recurrent neural networks to classify ECG beats. Graphically structured data was considered by [Wu+19]. Recurrent neural networks were discussed in [BT19]. Recently, active learning was also used for generative neural networks in [CG20] where they train a network to identify promising candidates in material design.

Active learning for *regression* neural networks has not gained much attention so far. This might be due to the fact that the variance of networks in a Bayesian setting is naturally available and an appealing choice for an acquisition function. Nevertheless, there have been some recent advances like the motivational paper for this work [Tsy+19] and their earlier work [TPS18]. Hafner et al. [Haf+20] examined how priors of Bayesian neural networks effect the uncertainty estimates of out of distribution data. They developed *noise contrastive priors* which prevent networks from overfitting outside of the training distribution and verified their results in an active learning setting. Another application in the context of PDEs comes from [NGK19a] and will be presented in more detail in chapter 6. The remainder of this chapter is motivated by the ideas presented in [TPS18] and develops an intuitive sampling strategy for the regression case based on Bayesian neural network.

4.3 Bayesian Neural Networks

When training a classical neural network we aim to use the available training data to calculate a set of parameters $\hat{\theta}$ which corresponds in some sense to a *good* approximation of the underlying function. Given a fixed input x^* we can use a fixed set of parameters $\hat{\theta}$ to compute an output for that point via a forward pass through the network $f_{nn}(x^*; \hat{\theta})$.

In the Bayesian setting, we are not interested in calculating point estimates but rather distributions over possible output values. To this end, we will use the training data $\mathcal{D} = (X, Y)$ to estimate a distribution $p(\theta|\mathcal{D})$ over the possible parameters θ which is called the **posterior** (the next section is dedicated to finding this distribution). Since every fixed set of parameters θ corresponds to a neural network and therefore to a function $f_{nn}(., \theta) : \mathcal{X} \to \mathbb{R}$, the posterior is also a distribution over evaluation functions. In the classical setting one would try to find a set of parameters θ^{MLE} which maximize the likelihood of the training data $\theta^{MLE} = \arg\max_\theta p(Y|X, \theta)$ where $p(Y|X, \theta)$ is the probability of observing Y given input X and parameters θ. Note that this value depends on the noise assumption: If we for example assume that there is no noise at all this probability would be 1 if the predictions $f_{nn}(X; \theta)$ match Y exactly and 0 otherwise and for an independent additive normal noise with variance γ, $p(Y|X, \theta)$ would be $\prod_{(y_i, x_i) \in \mathcal{D}} \mathcal{N}(y_i; f_{nn}(x_i, \theta), \gamma)$. Typically, the likelihood can be calculated rather easily, because we only needs to compare the deterministic forward pass $f_{nn}(X; \theta)$ with the observed labels Y.

Assuming now that we have the posterior distribution $p(\theta|\mathcal{D})$ already at hand, we can answer queries for a new input x^* by taking the expectation over the posterior of the parameters. The probability of a label y^* in the output space then becomes

$$p(y^*|x^*) = \mathbb{E}_{p(\theta|\mathcal{D})}\left[p(y^*|x^*, \theta)\right]. \tag{4.4}$$

For classification tasks this distribution would assign a probability to each label. For regression tasks a continuous distribution is obtained. One can imagine this as if every possible neural network configuration would vote on the output for x^* and the votes were weighted by the likelihood of the respective network (the more likely a particular network configuration is, the more its output is taken into account). This distribution offers deeper insight compared to the single value we get in the classical setting. But we can also derive a deterministic function from the distribution $p(y^*|x^*)$ by taking the average (which makes particularly sense when we know that the average is also the maximum):

$$\hat{f}(x_*) = \mathbb{E}_{p(y^*|x^*)}\left[y^*\right] = \int \left(\int p(y^*|x^*, \theta) p(\theta|\mathcal{D}) d\theta \right) y^* dy^* \tag{4.5}$$

Typically, the posterior distribution cannot be calculated in a closed form. But if we assume that we can sample from it we can approximate the mean (and also its variance, as we will see in section 4.5) with say S many Monte Carlo Samples $\theta^i \sim p(\theta|\mathcal{D})$

$$\hat{f}(x_*) \approx \frac{1}{S} \sum_{i=1}^{S} f_{nn}(x^*, \theta^i). \tag{4.6}$$

Finally, the Bayesian setting also allows to define a **prior** for the parameters $p(\theta)$ which is independent of the training data. Usually, the prior can be used to incorporate knowledge - for example from domain experts or earlier experiments - about the parameters but for neural networks where the parameters do not have a clear interpretation, it is rather a regularization term which prevents the network from overfitting. If a prior is used the point estimate in the classical setting is called maximum a postiori estimator and takes the form $\theta^{MAP} = \arg\max_\theta \log p(Y|X, \theta) + \log p(\theta)$. Placing a Gaussian prior is most common which corresponds to L2 regularization. In the next section we will discuss how to derive the posterior from the training data.

4.4 Training Bayesian Neural Networks

After having seen how a Bayesian neural network (BNN) works *if* we have the posterior available, this section is devoted to the training process of BNNs which consists of deriving the posterior $p(\theta|\mathcal{D})$ from the training data.

If we again denote the set of training points as $\mathcal{D} = (X, Y)$, we can infer properties of the posterior using Bayes theorem:

$$p(\theta|\mathcal{D}) = \frac{p(Y|X, \theta)p(\theta)}{p(Y|X)}. \tag{4.7}$$

The **marginal likelihood** $p(Y|X) = \int p(Y|X,\theta)p(\theta)d\theta$ is independent of θ and a normalization term which we do not have to consider yet. Therefore, we can simplify to

$$p(\theta|\mathcal{D}) = \frac{p(Y|X,\theta)p(\theta)}{p(Y|X)} \propto p(Y|X,\theta)p(\theta). \tag{4.8}$$

As already seen, the likelihood of the training data $p(Y|X,\theta)$ can be computed by a forward pass through the network and the prior $p(\theta)$ is specified by the user. Accordingly, $p(Y|X,\theta)p(\theta)$ can easily be calculated but only for *fixed* parameters θ. This means for *given* parameters we can calculate their relative (simplification above) posterior probability, yet computing the *complete* distribution $p(\theta|\mathcal{D})$ is more difficult. To exemplify this consider again the case of independent additive normal noise ($p(Y|X,\theta) = \prod_{(y_i,x_i)\in\mathcal{D}} \mathcal{N}(y_i; f_{nn}(x_i,\theta),\gamma)$) and assume a centered Gaussian as prior $p(\theta) = \mathcal{N}(\theta; 0, \alpha I), \alpha > 0$. Putting them together, the posterior becomes

$$p(\theta|\mathcal{D}) \;\propto\; \mathcal{N}(\theta; 0, \alpha I) \prod_{(y_i,x_i)\in\mathcal{D}} \mathcal{N}(y_i; f_{nn}(x_i,\theta),\gamma). \tag{4.9}$$

Assuming such distributions for prior and noise is already a simplification, yet we can not calculate the posterior analytically because of the complicated dependence of $f_{nn}(x,\theta)$ on θ. There are two fundamentally different approaches to deal with this problem: The first approach exploits the fact that we can calculate the posterior for fixed parameters. Markov Chain Monte Carlo (MCMC) algorithms construct a series of samples by moving in the sample space in small steps. By exploiting the fact that the real posterior can be calculated for fixed values they ensure that the derived samples behave similar to samples from the real posterior when the number of samples is sufficiently large. However, as BNNs are non-linear hierarchical model with thousands of parameters those techniques can only be applied to relatively small networks.

The second approach is to approximate the posterior with a simpler distribution as is was proposed by Blundell et al. in 2015. In [Blu+15] they develop a training scheme for BNNs called *Bayes by Backprop* which contains elements from the stochastic gradient decent and backpropagation used for classical neural networks. In comparing the two approaches consider the work [G M+18] which we will get to know in more detail in the next chapter. They were only able to use a BNN with three layers of 50 neurons because - as the authors state: *"For larger networks the MCMC approach would prohibitively slow"*. In contrast, [Blu+15] already trained networks with up to 1,200 neurons in two layers which have more than ten times the amount of parameters.

4.4.1 Bayes by Backprop

The key idea of Bayes by Backprop is to do a *Variational Inference* on the parameter distribution which gets optimized in a backpropagation like fashion. First presented in [Blu+15], it is based on ideas developed in [KW13] and was added to TensorFlow Probability last year [Tra+19].

Variational inference means that we approximate the real posterior $p(\theta|\mathcal{D})$ by a simpler - so called *variational* - distribution $q(\theta|\eta)$ which is controlled by parameters

η. For example, if q is chosen to be a Gaussian, η would include the mean and the standard deviation. The goal will be to determine a set of variational parameters η^*, such that q is a good approximation of the real posterior ($q(\theta|\eta^*) \approx p(\theta|\mathcal{D})$)) because then we can use q instead of the real posterior for estimating mean, variance and other statistics of the network output.

To derive the optimal parameters η^*, the difference between the variational distribution and the real posterior is defined and minimized. The difference between two distributions can be measured by the *Kullback–Leibler [KL] divergence* ([KL51]) which for the two continuous distributions $q(\theta|\eta), p(\theta|\mathcal{D})$ is defined as

$$\mathrm{KL}[q(\theta|\eta)||p(\theta|\mathcal{D})] := \int q(\theta|\eta)\log\left(\frac{q(\theta|\eta)}{p(\theta|\mathcal{D})}\right). \tag{4.10}$$

Then, the optimal parameter η^* becomes

$$\begin{aligned}
\eta^* &= \arg\min_\eta \mathrm{KL}[q(\theta|\eta)||p(\theta|\mathcal{D})] \\
&\overset{(1)}{=} \arg\min_\eta \int q(\theta|\eta)\log\left(\frac{q(\theta|\eta)}{p(\theta)p(Y|X,\theta)}\right) \\
&= \arg\min_\eta \int q(\theta|\eta)\left(\log\left(\frac{q(\theta|\eta)}{p(\theta)}\right) - \log\left(p(Y|X,\theta)\right)\right) \\
&= \arg\min_\eta \underbrace{\left(\mathrm{KL}[q(\theta|\eta)||p(\theta)] - \mathbb{E}_{q(\theta|\eta)}\left[\log p(Y|X,\theta)\right]\right)}_{=:\mathcal{F}(\mathcal{D},\eta)} \\
&= \arg\min_\eta \mathcal{F}(D,\eta)
\end{aligned} \tag{4.11}$$

where in (1) Bayes theorem (4.7) was used and the of η independent term $\log(P(\mathcal{D}))$ was dropped. Instead of minimizing the KL-divergence directly, the introduced shorthand $\mathcal{F}(\mathcal{D},\eta)$ will be minimized. It is a lower bound of $\mathrm{KL}[q(\theta|\eta)||p(\theta|\mathcal{D})]$ because the dropped term $\log P(\mathcal{D})$ is always positive. The term $\mathcal{F}(\mathcal{D},\eta)$ is also known evidence lower bound (ELBO) and consists of two terms. Minimizing the first term (the KL-divergence) brings the variational distribution q closer to the prior, while the second term (the expectation) incentivizes q to maximize the likelihood of the training data. In extreme cases the first term would be zero, if q would be chosen as the prior and minimizing only the second term yields the maximum likelihood point estimate for q.

Minimizing $\mathcal{F}(\mathcal{D},\eta)$ with respect to η directly is still computationally expensive or intractable, hence a stochastic gradient decent similar to the usual backpropagation is applied. This also has the benefit that no closed form of the costs in 4.11 are required. Instead, the exact costs can be estimated by rewriting $\mathcal{F}(\mathcal{D},\eta)$ as an expectation over q which we can approximate with say S many Monte Carlo samples $\theta^i \sim q(\theta|\eta)$ from the variational posterior

$$\begin{aligned}
f(\theta,\eta) &:= (\log q(\theta|\eta) - \log p(\theta) - \log p(Y|X,\theta)) \\
\mathcal{F}(\mathcal{D},\eta) &= \mathbb{E}_{q(\theta|\eta)}\left[f(\theta,\eta)\right] \approx \frac{1}{S}\sum_{i=1}^{S} f(\theta^i,\eta).
\end{aligned} \tag{4.12}$$

Recall that we want to find an η^* that *minimizes* $\mathcal{F}(\mathcal{D}, \eta)$. So assume we have some η and want to improve it. We can use the estimate from above to get an idea about the costs $\mathcal{F}(\mathcal{D}, \eta)$, yet to apply SGD we have to calculate or estimate the gradient of this costs with respect to the variational parameters $\nabla_\eta \mathcal{F}(\mathcal{D}, \eta)$.

If q **would not** depend in η, we **would** have $\nabla_\eta \mathbb{E}_{q(\theta)}\left[f(\theta, \eta)\right] = \mathbb{E}_{q(\theta)}\left[\nabla_\eta f(\theta, \eta)\right]$, which **would** allow us to do Monte Carlo sampling for the gradient as well. But $q(\theta|\eta)$ **does** depend on the optimization parameter η, after all that was the whole point of the variational distribution q. This problem is circumvented with a generalization of the *Gaussian reparameterization trick* (proposition 1 in [Blu+15]). We externalize the randomness of θ by sampling first from a distribution which is independent of η and then transform this sample into a sample of q. This transforms the derivation of this expectation to an expectation of derivatives which we can estimate more easily.

Assume we can transform $\theta = g(\eta, \epsilon)$ with $\epsilon \sim q(\epsilon)$ independent of η and $g(.,.)$ a differentiable deterministic function and furthermore that the marginal distribution coincides $q(\epsilon)d\epsilon = q(\theta|\eta)d\theta$. Then we see that we can move the gradient into the expectation which we can then approximate with Monte Carlo samples ϵ^i from the distribution $q(\epsilon)$:

$$\begin{aligned}\nabla_\eta \mathbb{E}_{q(\theta|\eta)}\left[f(\theta, \eta)\right] &= \nabla_\eta \int q(\theta|\eta) f(\theta, \eta) d\theta \\ &= \int q(\epsilon) \nabla_\eta f(g(\eta, \epsilon), \eta) d\epsilon \\ &= \mathbb{E}_{q(\epsilon)}\left[\nabla_\eta f(g(\eta, \epsilon), \eta)\right] \\ &\approx \frac{1}{S} \sum_{i=1}^{S} \nabla_\eta f(g(\eta, \epsilon^i), \eta).\end{aligned} \tag{4.13}$$

Note that $\nabla_\eta f(g(\eta, \epsilon^i), \eta)$ can be computed via the chain rule as long as g is differential with respect to η.

To make this more clear, consider the commonly used example of a variational posterior which is a diagonal Gaussian distribution $q(\theta|\eta) = \mathcal{N}(\theta; \mu, \sigma I)$. We parameterize the posterior with $\eta = (\mu, \rho)$ and $\sigma = \log(1 + \exp(\rho))$ (this ensures that σ is always positive). Now we can sample ϵ from a standard Gaussian and transform it to be a sample from the posterior

$$\begin{aligned}\epsilon &\sim \mathcal{N}(0, I) \\ \theta := g(\eta, \epsilon) &= \mu + \log(1 + \exp(\rho)) \circ \epsilon \\ \implies \ \theta &\sim q(\theta|\eta)\end{aligned} \tag{4.14}$$

where $\circ$ denotes pointwise multiplication. Using those as Monte Carlo sample, the gradients w.r.t. $\eta = (\mu, \rho)$ can easily be estimated as $g(\eta, \epsilon)$ is clearly differentiable w.r.t μ and ρ.

$$\nabla_\eta f(\theta = g(\eta, \epsilon), \eta) = \nabla_\theta f(\theta, \eta) \frac{d}{d\eta} g(\eta, \epsilon) + \nabla_{\eta'} f(\theta, \eta') \tag{4.15}$$

In conclusion, estimating $\nabla_\eta \mathbb{E}_{q(\theta|\eta)}\left[f(\theta,\eta)\right]$ with samples ϵ^i allows us to optimize η with respect to $\mathcal{F}(\mathcal{D},\eta)$ via gradient decent with a learning rate of $\nu > 0$

$$\begin{gathered}\eta' = \eta - \nu\frac{1}{S}\sum_{i=1}^{S}\nabla_\eta f(g(\eta,\epsilon^i),\eta),\\ \text{where } f(\theta,\eta) = \log q(\theta|\eta) - \log p(\theta) - \log p(Y|X,\theta).\end{gathered} \tag{4.16}$$

By repeating this updating procedure a - hopefully - close to optimum η^* can be found and therefore also a good variational distribution $q(\theta|\eta^*)$ which then can be used instead of the real posterior. Note that the optimization of η also optimizes the network to fit the data, as equation 4.16 includes in particular $-\nabla_\eta \log p(Y|X, g(\eta,\epsilon^i))$. It is also worth mentioning that the authors extended this idea to minibatches which are often used in practice to speed up the training process.

4.5 BNN and Active Learning

Knowing how to train or at least approximate Bayesian neural networks, we are now ready to apply them in the Active Learning framework from chapter 2. Recall that our intention for considering the neural network in a Bayesian setting was to derive a model which predicts distributions. The additional information contained in the output can be used for active learning in a very intuitive manner which was developed [TPS18] with the difference that they used classical neural network with dropout as a source of randomness while we directly consider a probabilistic model. It works as follows and considers the pool-based active learning version from chapter 2.

Recall that we have a pool of points available and want to decide which of them are most important to label. We start by feeding all the points in the pool through the trained BNN. If the network now predicts a distribution with a low variance for a point in the pool, this means that there is little uncertainty about the correct label and there is no reason the request the label for such a point because the network itself can already predict the label well. However, if the variance is very high, the network does not really know which label is correct. We expect that the addition of high variance points is beneficial to the training process, as knowing the label to this point should greatly reduce the uncertainty around it. Accordingly we follow the principle of *highest uncertainty* as presented in section 2.3.2.

Now assume we are given a pool X_{pool} and that we have a BNN with parameters θ trained on data $\mathcal{D} = (X, Y)$ and can sample from its posterior or a napproximation $p(\theta|\mathcal{D})$ (for example with the methods described in the last section). We want to replace line 3 in the active learning framework 1 which means we have to decide which points from the pool are selected for labeling. Again we can use S Monte Carlo samples $\theta^i \sim p(\theta|\mathcal{D})$ to estimate the mean m and variance σ^2 of the output for a new input $x^* \in X_{pool}$

$$\begin{aligned} m(x^*) &:= \mathbb{E}_{p(\theta|\mathcal{D})}\left[f_{nn}(x^*, \theta)\right] \approx \frac{1}{S}\sum_{i=1}^{S} f_{nn}(x^*, \theta^i) \\ \sigma^2(x^*) &:= \mathbb{E}_{p(\theta|\mathcal{D})}\left[f_{nn}(x^*, \theta) - m(x^*)\right] \\ &\approx \frac{1}{S-1}\sum_{i=1}^{S}\left(f_{nn}(x^*, \theta^i) - \frac{1}{S}\sum_{j=1}^{S} f_{nn}(x^*, \theta^j)\right). \end{aligned} \tag{4.17}$$

Note that $f_{nn}(x^*, \theta^i)$ is a simple forward pass through the network with fixed parameters θ^i, while $f_{nn}(x^*, \theta)$ is a random function which depends on the posterior distribution of the parameters. We estimate the variances for all $x^* \in X_{pool}$ and request the labels for those with the highest variances as shown in algorithm 2 to define a sampling strategy for BNNs which can be used to complete the active learning framework from chapter 2.

Algorithm 2: Variance Sampler for Bayesian Neural Networks

```
Data: number of samples $N$, pool $X_{pool}$, neural network model $f_{nn}$ and
      posterior $p(\theta|\mathcal{D})$, number of Monte Carlo samples $S$
Result: set of points $X_{sample} \subset X_{pool}$, with $|X_{sample}| = N$
1  for $i = 1, \dots, S$ do
2      sample $\theta^i \sim p(\theta|\mathcal{D})$;
3  end
4  for $x_j \in X_{pool}$ do
5      for $i = 1, \dots, S$ do
6          $y_{j,i} = f_{nn}(x_j, \theta^i)$;
7      end
8      calculate mean: $\bar{y}_j = \frac{1}{S}\sum_{i=1}^{S} y_{j,i}$;
9      calculate variance:
           $$\sigma_j^2 = \frac{1}{S-1}\sum_{i=1}^{S}(y_{j,i} - \bar{y}_j)^2$$
10 end
11 return: return $N$ points from $X_{pool}$ with highest variances $\sigma_j^2$
```

While this Variance Sampler is satisfying to a certain degree, its design runs into problems when multiple points are requested at once (this is often necessary to reduce runtime). As the variance of close points is typically pretty similar, it is likely, that the instances with the highest variances are instances from the same region of the input space. Therefore the sampler would return many points which actually contain very similar information, a waste of resources.

To circumvent this, one could either retrain the network after each addition of a point or introduce spacing heuristics which keep the sampled points apart. But retraining the network is computationally expensive and heuristics require precise hyperparameter optimization. A more elegant solution was found by [Tsy+19] and is presented in the next chapter: It is known that random neural networks converge in distribution to Gaussian processes (see section 3.1). This connection is used to approximate the trained BNN with a Gaussian process which also allows variance-based sampling while providing a quick updating procedure for sampling divers points.

Chapter 5

Connection between Neural Networks and Gaussian Processes

In this chapter a connection between neural network which we introduced in the last chapter and the Gaussian process from section 3.1 is established. Although the theory considered in the first two sections is derived for the limits of *randomly* distributed neural networks, it is known that already finite Bayesian neural networks behave similar to Gaussian processes (e.g. chapter 5 in [G M+18]). After building an intuition with the simple case of a single-layer network in section 5.1, we will proceed to general multi-layer networks in section 5.2 and show how the networks converge towards Gaussian processes. In section 5.3 the work of [Tsy+19] is followed to develop a sampler for the active learning setting which combines variance-based decision making with a quick updating procedure as suggested. Finally, in section 5.4 their work is improved by extending the suggested updating procedure to batches of points.

5.1 Single-Layer Neural Networks

The connection for single-layer neural networks was first discovered by [Nea96]. Yet we will use the notation of [Lee+17] here because they extend the proof also to multi-layer networks.

Consider a standard neural network with one hidden layer of width w and a single output. Recall the notation introduced in section 4.1: We denote the weights and biases before the hidden layer with W^1 and b^1 and the weights and biases after the hidden layer with W^2 and b^2. Given an input $x \in \mathcal{X} \subset \mathbb{R}^n$, we compute the forward pass via

$$f_{nn}(x) = b^2 + \sum_{i=1}^{w} W_i^2 x_i^1(x) \; ; \; x_i^1(x) = \phi(b_i^1 + \sum_{j=1}^{n} W_{ji}^1 x_j). \tag{5.1}$$

Now imagine we would not infer the parameters from the training data but draw them randomly and independently from distributions with zero mean and a variance of σ_W^2/w and σ_b^2 respectively. We note that drawing the weights and biases i.i.d implies that the entries of the vector $x^1(x)$ are also independent and identically distributed. Thus, $f_{nn}(x)$ is the sum of i.i.d. terms and applying the Central Limit Theorem 3.1.2 we conclude that the output $f_{nn}(x)$ will follow a Gaussian distribution for $w \to \infty$, i.e. when the width of the network becomes infinitely big. Moreover, as W^2 and b^2 are centered, the $f_{nn}(x)$ will also have zero mean.

If we fix a collection of inputs $\{x^{a_1}, \dots, x^{a_i}\}$, we can see that in the limit $w \to \infty$ the corresponding outputs $\{f_{nn}(x^{a_1}), \dots, f_{nn}(x^{a_i})\}$ follow a multivariate Gaussian distribution by the multivariate central limit theorem 3.1.4. This is precisely the requirement we made in the definition of a Gaussian Process (3.1.6) indexed by the input space $\{f(x) : x \in \mathcal{X}\}$. Therefore, the network actually converges to a centered Gaussian process with $w \to \infty$. Taking a closer look at equation 5.1, we can see that the defining covariance can be written as

$$\begin{aligned} k(x, x') &= \mathbb{E}_{x,x'}\left[f_{nn}(x) f_{nn}(x')\right] \\ &= \sigma_b^2 + \sigma_W^2 \mathbb{E}_{x,x'}\left[x_i^1(x) x_i^1(x')\right] \end{aligned} \tag{5.2}$$

and therefore conclude

$$f_{nn}(x) \xrightarrow{dist.} \mathcal{GP}(0, k) \text{ with } w \longrightarrow \infty. \tag{5.3}$$

5.2 Multi-Layer Neural Networks

The convergence result from the previous section is now extended to the case of multi-layer networks which can be done in a few different ways. The arguably strongest result was proven by [G M+18]. They consider a network whose layer widths are taken to infinity *simultaneously* (different rates are allowed). This poses a challenge because of the recursive structure of neural networks: If an intermediate layer has not converged (yet) and is therefore not a multivariate normal, its influence on the other layers make the analysis of their convergence more difficult. Under some weak assumptions they give a rigorous proof of the convergence in distribution and compare empirically Gaussian processes with small Bayesian neural networks which they infer using a MCMC approach (refer to the paragraph right before 4.4.1).

While the recursive structure poses a challenge when taking the layer widths to infinity simultaneously, it can also be exploited to give an intuitive proof of convergence for the case where the layer widths go to infinity one after another. This line of argument as proposed in [Lee+17] is now presented.

Again consider a neural network as described in section 4.1, this time with L layers and widths w_l which takes inputs of dimension n and returns a single value. The weights matrices and biases are denoted with W^l and b^l respectively and we assume them to be drawn i.i.d from distributions with zero mean and a variance of σ_W^2 / w_l and σ_b^2, respectively.

We proceed by induction in the sense that we take the layer widths to infinity successively ($w_1 \longrightarrow \infty, w_2 \longrightarrow \infty, \dots$). In the last section, we already established the basis for the induction and know that the values computed at the first hidden layer $x^1(x)$ are i.i.d. as well. Now, for the induction step, assume that at layer $l - 1$

the i.i.d. values $x^{l-1}(x)$ are already computed. The values at the next layer are computed via

$$x_i^l(x) = \phi(y_i^l(x)),\ y_i^l(x) := b_i^l + \sum_{j=1}^{w_{l-1}} W_{ij}^l x_j^{l-1}(x). \tag{5.4}$$

Again, $y_i(x)$ is the sum of i.i.d term random terms, thus we can apply the multivariate central limit theorem 3.1.4 to conclude that for any finite set of input values $\{x^{a_1}, \dots, x^{a_i}\}$ the values $\{y_i(x^{a_1}), \dots, y_i(x^{a_i})\}$ will be centered and jointly Gaussian distributed $y_i^l(x) \sim \mathcal{GP}(0, K^l)$ if $w_{l-1} \to \infty$. To complete the induction step, observe that under application of the activation function ϕ, the intermediate values remain i.i.d. Note that the non-linearity ϕ is not applied before the output layer and the final value is therefore Gaussian distributed.

The covariance K^l can be computed as in the previous section, leading to a recursive formula which only depends on ϕ and the covariance of the previous layer K^{l-1}:

$$\begin{aligned} K^l(x, x') &= \mathbb{E}_{x,x'}\left[y_i^l(x) y_i^l(x')\right] \\ &= \sigma_b^2 + \sigma_w^2 \mathbb{E}_{y^{l-1} \sim \mathcal{GP}(0, K^{l-1})}\left[x_i^l(x), x_i^l(x')\right] \\ &= \sigma_b^2 + \sigma_w^2 \mathbb{E}_{y^{l-1} \sim \mathcal{GP}(0, K^{l-1})}\left[\phi(y_i^l(x)), \phi(y_i^l(x'))\right]. \end{aligned} \tag{5.5}$$

It should be noted that this covariance can be computed analytically in some cases (e.g. in the case of a ReLU activation function [CS09]) and efficiently approximated in other cases, as described in section 2.5 of [Lee+17]. Research has been conducted to extract promising kernels from infinite-width networks for example in [HJ15] and [JGH18]. The latter derived the *Neural Tangent Kernel* (NTK) from the limit of neural networks which allowed them to consider the approximation function f_{nn} directly in the function space during training and recently. Furthermore, convolutional NTK has been developed in [Aro+19]. However, infinite wide random networks are not of interest for this work but rather the attempt to extract the kernels of finite Bayesian neural networks to construct a sampling policy in the next section.

5.3 GPA Sampler

Why is this connection now interesting for the active learning problem? It tells us that neural networks and Gaussian processes exhibit - at least in the limit - a similar behavior. In fact, the experiments in [G M+18] show that already finite Bayesian neural networks of relatively small size (three layers with 50 hidden units) behave very similar to Gaussian processes. This connection can be used in the active learning setting to utilize the advantages of both models while circumventing their downsides.

Recall that the variance-based sampler proposed in the last section (algorithm 2) was not satisfactory because it is not optimized for sampling multiple points at once and might therefore return many similar points which contain the same information. But as we employ neural network in the active learning framework we really do want to sample multiple points because retraining the network after every single acquired point would be computationally too expensive. Therefore, the trained BNN will be *approximated* with a Gaussian process which is only used in the active learning iteration for selecting points from the pool. The actual training will be done on the BNN which scales much better with bigger data than the Gaussian process while the Gaussian process approximation allows for an elegant way of sampling divers points. In the following, the idea from [Tsy+19] is presented with the slight modification of extending their updating rule from the posterior variance to the complete covariance matrix (equation 5.11).

Again, let us consider a Bayesian neural network model f_{nn} together with its posterior distribution $p(\theta|\mathcal{D})$. We have already seen in section 4.3 that for a fixed input x, the network output $f_{nn}(x,\theta)$ is a random function (depending on $\theta \sim p(\theta|\mathcal{D})$) and how to estimate its mean and variance (equation 4.17). We can also estimate its *covariance* for two input points x, x' with S Monte Carlo samples θ^i from $p(\theta|\mathcal{D})$ in a similar fashion. Note that we only need the abilities to sample from the posterior and to compute forward passes through the network with fixed parameters to compute an estimates for the mean $\hat{m}$ and the covariance $\hat{k}$. Accordingly, we can calculate them even if the label of a point is not known

$$\begin{aligned} m(x) &= \mathbb{E}_{p(\theta|\mathcal{D})}\left[f_{nn}(x,\theta)\right] \\ &\approx \frac{1}{S}\sum_{i=1}^{S} f_{nn}(x,\theta^i) =: \hat{m}(x) \\ k(x,x') &= \mathbb{E}_{p(\theta|\mathcal{D})}\left[\left(f_{nn}(x,\theta)-m(x)\right)\left(f_{nn}(x',\theta)-m(x')\right)\right] \\ &\approx \frac{1}{S-1}\sum_{i=1}^{S}\left(\left(f_{nn}(x,\theta^i)-\hat{m}(x)\right)\left(f_{nn}(x',\theta^i)-\hat{m}(x')\right)\right). \\ &=: \hat{k}(x,x'). \end{aligned} \tag{5.6}$$

If we consider $f_{nn}(x,\theta)$ as a random function indexed by the input point x, we see that it fulfills precisely the definition 3.1.5 of a stochastic process over the input space $\mathcal{X}$. Furthermore, as Gaussian processes are stochastic processes which are completely defined by their first two moments, this implies that we can approximate the stochastic process $f_{nn}(x,\theta)$ with a Gaussian process $g(x|f_{nn})$ which is defined by the estimates in equation 5.6:

$$g(x|f_{nn}) \sim \mathcal{GP}(\hat{m},\hat{k}). \tag{5.7}$$

So far this new process does not contain any more information than the network. Yet, as already stated, we can calculate $\hat{m}$ and $\hat{k}$ not only for the training instances but also for those in the pool. This way we can use our knowledge about Gaussian processes (see section 3.2) to calculate also the *posteriori* variance of $g(x|f_{nn})$ given the training data. To this end, let the set of training points X and the set of pool points X_{pool} be given. Estimating the covariances for all $x_1, x_2 \in X \cup X_{pool}$ according to equation 5.6 allows the computation of the posteriori variance for any $x \in X_{pool}$ using equation 3.8 as

$$\sigma^2_{post}(x|f_{nn},X) = \hat{k}(x,x) - \hat{\mathbf{k}}^{\mathbf{T}}(x)\hat{K}(X,X)^{-1}\hat{\mathbf{k}}(x) \tag{5.8}$$

where $\hat{K}(X, X)$ denotes the covariance matrix estimate for the training points and $\hat{\mathbf{k}}(x)$ is the vector containing the covariance estimates of x and the training points

$$\begin{aligned} \hat{K}(X, X) &= \left[\hat{k}(x_i, x_j)\right]_{x_i, x_j \in X} \\ \hat{\mathbf{k}}(x) &= \left(\hat{k}(x_1, x), \ldots, \hat{k}(x_{|X|}, x)\right)^T . \end{aligned} \tag{5.9}$$

The advantage of this formulation compared to using the plain variance of the network is that the posterior variance in equation 5.8 only depends on the covariance estimates and *not* on the labels. Therefore, it allows us to update the Gaussian process approximation even *before* we actually request the labels. Assume we decide to sample a point in the pool $x^* \in X_{pool}$, remove it from the pool $X'_{pool} = X_{pool} \setminus \{x^*\}$ and add it to the training set $X' = X \cup \{x^*\}$. When using the variance of the network we would now have to request the label for x^* and retrain the network on the new training data to get an updated variance. With the Gaussian process approximation and the covariance estimates $\hat{k}(x_i, x_j)$ already at hand we see that we can simply use equation 5.8 again with the updated pool and training data to get a new variance estimate which takes the sampled point x^* into account.

Apart from only taking a *single* point with the highest variance, also a *batch* of multiple points with high variances can be selected. This motivates the Batch Gaussian process approximation (BGPA) Sampler (algorithm 3) which computes a total number of N points from the pool to be labeled. Lines 1-10 describe the estimation of the covariances, as detailed in equation 5.6. Then, the algorithm continues to sample a batch of M points per iteration of the while loop (line 12) by calculating the posterior variance (line 13-15) and then updating the training and pool datasets (line 16-18).

To improve the numerical stability we can add a small unit matrix to $\hat{K}(X, X)$ in line 14, before inverting it. This corresponds to assuming a certain observed noise for the Gaussian process (refer equation 3.10). The posterior variance becomes

$$\sigma^2_{post}(x|f_{nn}, X) = \hat{k}(x, x) - \hat{\mathbf{k}}^{\mathbf{T}}(x) \left(\hat{K}(X, X) + \lambda I\right)^{-1} \hat{\mathbf{k}}(x), \tag{5.10}$$

where $\lambda > 0$ is a small parameter the influence of which we will investigate empirically in chapter 6.

Algorithm 3: Batch GPA Sampler for Bayesian Neural Networks

```
Data: number of samples $N$, number samples per iteration $M$, training
      points $X$, pool $X_{pool}$, neural network model $f_{nn}$ and posterior $p(\theta|\mathcal{D})$,
      number of Monte Carlo samples $S$, regularization parameter $\lambda$
Result: set of points $X_{samples} \subset X_{pool}$, with $|X_{samples}| = N$

1  define $\bar{X} := X \cup X_{pool}$;
2  for $i = 1, \dots, S$ do
3      sample $\theta^i \sim p(\theta|\mathcal{D})$;
4      for $x_j \in \bar{X}$ do
5          $y_{j,i} = f_{nn}(x_j, \theta^i)$ ;
6      end
7  end
8  for $x_a, x_b \in \bar{X}$ do
9      calculate covariances:
           $\hat{k}(x_a, x_b) = \frac{1}{S-1} \sum_{i=1}^{S} (y_{a,i} - \bar{y_a})(y_{b,i} - \bar{y_b}),$ with $\bar{y_j} = \frac{1}{S} \sum_{i=1}^{S} y_{j,i}$
10 end
11 $X' \longleftarrow X$;
12 while $|X' \setminus X| < N$ do
13     for $x_j \in X_{pool}$ do
14         calculate posterior variance:
               $\sigma^2_{post}(x_j|f_{nn}, X') = \hat{k}(x_j, x_j) - \hat{\mathbf{k}}^{\mathbf{T}}(x_j) \left(\hat{K}(X', X') + \lambda I\right)^{-1} \hat{\mathbf{k}}(x_j)$
15     end
16     $X_{new} \longleftarrow M$ points in $X_{pool}$ with highest posterior variance;
17     $X' \longleftarrow X' \cup X_{new}$;
18     $X_{pool} \longleftarrow X_{pool} \setminus X_{new}$;
19 end
20 return: $X_{samples} = X' \setminus X$
```

The major drawback of this sampling policy can be found in line 14. Because we want to sample points according to the updated posterior variance of the Gaussian process approximation, we have to invert $\hat{K}(X', X')$ in each iteration which contributes a runtime of $\mathcal{O}(|X|^3)$ per iteration with N/M iterations total. This can quickly become too much for large applications even if we compromise quality and decide to increase M and to sample a bigger number of points per iteration. To improve upon that, [Tsy+19] suggested a faster updating procedure. The notation is now changed to formulate the posterior estimate in vectors. This allows an updating procedure for the complete posterior covariance matrix of $g(x|f_{nn}, X)$ and not only for the variance, as shown in the paper. As a direct consequence the same updating procedure can be used multiple times in succession.

First, we reformulate equation 5.8 in terms of matrices. The addition of the regularization matrix λI is skipped from here on to simplify the notation. It is added again in the pseudocode to make the description complete.

$$\begin{aligned}\hat{K}(X_{pool}, X) &:= \left[\hat{k}(x_i, x_j)\right]_{x_i \in X_{pool}, x_j \in X} \\ \hat{K}(X_{pool}, X_{pool}) &:= \left[\hat{k}(x_i, x_j)\right]_{x_i, x_j \in X_{pool}} \\ \hat{K}(X_{pool}|f_{nn}, X) &= \hat{K}(X_{pool}, X_{pool}) - \hat{K}(X_{pool}, X)\hat{K}(X, X)^{-1}\hat{K}(X, X_{pool})\end{aligned} \tag{5.11}$$

Observe that the posterior variance the diagonal of $\hat{K}(X_{pool}|f_{nn}, X)$. Again, we are interested in calculating the updated version $\hat{K}(X'_{pool}|f_{nn}, X')$ after adding point x^*. As it turns out, there is a way of defining an updating procedure which we can apply to the old covariance matrix and transform it to the covariance matrix of for the updated training and pool sets.

$$\begin{aligned}\hat{K}(X'_{pool}|f_{nn}, X') &= \hat{K}(X'_{pool}, X'_{pool}) - \hat{K}(X'_{pool}, X')\hat{K}(X', X')^{-1}\hat{K}(X', X'_{pool}) \\ &\overset{(1)}{=} \hat{K}(X_{pool}|f_{nn}, X) - \frac{\hat{k}(X_{pool}, x^*)^T\hat{k}(X_{pool}, x^*)}{\hat{\sigma}^2(x^*|f_{nn}, X)}\end{aligned} \tag{5.12}$$

The proof of (1) is not given here, as it directly follows from the more general result in the next chapter. This update only requires the product of two vectors, rescaling of the resulting matrix and a subtraction from the old covariance matrix and is therefore computationally much cheaper than the recalculation. Note that the vector $\hat{k}(X_{pool}, x^*)$ is a row and $\hat{\sigma}^2(x^*|f_{nn}, X)$ a diagonal entry of the old covariance matrix and that the row and column of the new covariance matrix are precisely 0 after the update. Even though the $\hat{K}(X'_{pool}|f_{nn}, X')$ matrix calculated by the previous equation is of size $|X_{pool}| \times |X_{pool}|$, it does not contain any entries for the point x^* and is actually precisely the $(|X_{pool}| - 1) \times (|X_{pool}| - 1)$ size matrix one would expect.

In addition, this formulation reveals another interesting advantage of the method. After point x^* is selected and the new covariance matrix is calculated according to equation 5.12, points x' which contain information similar to x^* will have a low variance. This is the case because such a point x' has a **high** covariance $\hat{k}(x^*, x')$ with x^* and the updated posterior variance of x' is **reduced** by $\hat{k}(x^*, x')^2/\hat{\sigma}^2(x^*|f_{nn}, X)$. Therefore, the updating procedure ensures that selecting a single point reduces the incentive to sample similar points which in turn implies that the algorithm is likely to sample diverse points.

Putting this together, we can formulate a new sampling algorithm which we call *Fast* because it avoids the repeated inverse calculations. Up to line 8 this sampler actually does the same as the Batch GPA Sampler (algorithm 3). The main difference appears in line 13 where the fast updating procedure replaces the recalculation of the Gaussian process approximation from earlier.

```
Algorithm 4: Fast GPA Sampler for Bayesian Neural Networks
   Data: number of samples $N$, training points $X$, pool $X_{pool}$, neural network
         model $f_{nn}$ and posterior $p(\theta|\mathcal{D})$, number of Monte Carlo samples $S$,
         regularization parameter $\lambda$
   Result: set of points $X_{sample} \subset X_{pool}$, with $|X_{sample}| = N$

 1 define $\bar{X} := X \cup X_{pool}$;
 2 for $i = 1, \ldots, S$ do
 3     sample $\theta^i \sim p(\theta|\mathcal{D})$;
 4     $y_i = f_{nn}(\bar{X}, \theta^i)$
 5 end
 6 calculate covariance matrix:
       $$\hat{K}(\bar{X}, \bar{X}) = \frac{1}{S-1} \sum_{i=1}^{S} \left( (y_i - \bar{y})^T \cdot (y_i - \bar{y}) \right), \text{ with } \bar{y} = \frac{1}{S} \sum_{i=1}^{S} y_i$$
 7 calculate first posterior covariance matrix:
       $$\hat{K} = \hat{K}(X_{pool}, X_{pool}) - \hat{K}(X_{pool}, X) \left( \hat{K}(X, X) + \lambda I \right)^{-1} \hat{K}(X, X_{pool})$$
 8 $X_{samples} \longleftarrow \emptyset$;
 9 while $|X_{samples}| < N$ do
10     get posterior variance $\sigma^2_{post}(X_{pool}|f_{nn}, X \cup X_{samples}) = diagonal(\hat{K})$;
11     $x^* \longleftarrow \arg\max_{x \in X_{pool}} \sigma^2_{post}(x)$ ;
12     $X_{samples} \longleftarrow X_{samples} \cup \{x^*\}$;
13     update posterior covariance matrix:
           $$\hat{K} = \hat{K} - \frac{\hat{K}[:, x^*] \cdot \hat{K}[x^*, :]}{\left( \hat{K}[x^*, x^*] + \lambda \right)}$$
       $X_{pool} \longleftarrow X_{pool} \setminus \{x^*\}$;
14 end
15 return: $X_{samples}$
```

5.4 Fast Batch GPA Sampler

The presented fast updating procedure seems to give a big runtime advantage to algorithm 4 (Fast GPA Sampler) compared to the algorithm 3 (Batch GPA Sampler) which has to calculate the posterior variance of the Gaussian process approximation in every iteration. In practice it turns out that the ability to sample batches per iteration is often just as important because sampling M points per iteration reduces the runtime roughly by a factor of M. Therefore, both presented versions have their advantages and drawbacks. The Batch GPA Sampler can sample one batch per iteration, yet requires the costly calculation of $\hat{K}(X', X')^{-1}$ every iteration. In contrast, the Fast GPA Sampler avoids this inversion by a quick updating procedure but can only sample one point per iteration which result in slow runtimes as well.

In this section the updating formula 5.12 is generalized to the case where sample multiple points are sampled. This will lead to a sampling policy which combines the advantages of the two approaches presented in the last section. In particular, the combined policy will be able to sample batches per iteration and reduces the size of the inverted matrix significantly from the size of the training data to the size of the batch. As the calculation of the inverse is the most costly operation, this results in a big runtime advantage as will be shown empirically in section 6.3.

Again, assume that training data $\mathcal{D} = (X, Y)$ and a pool X_{pool} are given and that a set of points $X_* \subset X_{pool}$ has been requested based on the covariance estimates in equation 5.11. We are now interested in calculating the updated covariance matrix for $X'_{pool} = X_{pool} \setminus X_*$ and $X' = X \cup X_*$ which is $\hat{K}(X'_{pool}|f_{nn}, X')$. To make the calculations more clear, the covariances are written in block matrix notation:

$$\begin{pmatrix} \hat{K}(X'_{pool}, X'_{pool}) & \hat{K}(X'_{pool}, X_*) & \hat{K}(X'_{pool}, X) \\ \hat{K}(X_*, X'_{pool}) & \hat{K}(X_*, X_*) & \hat{K}(X_*, X) \\ \hat{K}(X, X'_{pool}) & \hat{K}(X, X_*) & \hat{K}(X, X) \end{pmatrix} =: \begin{pmatrix} P & A & B \\ A^T & R & C \\ B^T & C^T & T \end{pmatrix}. \tag{5.13}$$

In other words, P contains the covariances of the pool without the new samples, R the covariances of the new samples and T the covariances of the training data. The matrices A, B and C contain the respective covariances between these sets of points.

Calculating the posterior covariance matrix **before** adding any points, i.e. with pool $X_{pool} = X'_{pool} \cup X_*$ and training data X in block matrix notation corresponds to calculating

$$\begin{aligned} \hat{K}(X_{pool}|f_{nn}, X) &= \begin{pmatrix} P & A \\ A^T & R \end{pmatrix} - \begin{pmatrix} B \\ C \end{pmatrix} T^{-1} \begin{pmatrix} B^T & C^T \end{pmatrix} \\ &= \begin{pmatrix} P - BT^{-1}B^T & A - BT^{-1}C^T \\ A^T - CT^{-1}C^T & R - CT^{-1}C^T \end{pmatrix}. \end{aligned} \tag{5.14}$$

The posterior covariance matrix **after** adding the points X^*, i.e. with pool $X'_{pool} = X_{pool} \setminus X_*$ and training data $X' = X \cup X_*$ is calculated with

$$\hat{K}(X'_{pool}|f_{nn}, X') = P - \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} R & C \\ C^T & T \end{pmatrix}^{-1} \begin{pmatrix} A^T \\ B^T \end{pmatrix}. \tag{5.15}$$

In line 14 of the Batch GPA Sampler we use in fact precisely this equation to calculate the covariance matrix for the next iteration. We will now see how we can improve this by making use of already calculated values and avoiding to calculate the inversion directly. To this end, first consider the inversion of the inner matrix in block form:

$$\begin{aligned}\begin{pmatrix} R & C \\ C^T & T \end{pmatrix}^{-1} &= \begin{pmatrix} (R - CT^{-1}C^T)^{-1} & -(R - CT^{-1}C^T)^{-1}CT^{-1} \\ -T^{-1}C^T(R - CT^{-1}C^T)^{-1} & T^{-1} + T^{-1}C^T(R - CT^{-1}C^T)^{-1}CT^{-1} \end{pmatrix} \\ &= \begin{pmatrix} M^{-1} & -M^{-1}CT^{-1} \\ -T^{-1}C^T M^{-1} & T^{-1} + T^{-1}C^T M^{-1}CT^{-1} \end{pmatrix}, \\ &\text{with } M := (R - CT^{-1}C^T).\end{aligned} \tag{5.16}$$

Observe that the defined matrix M is precisely the lower right corner of the already calculated matrix from equation 5.14. Plugging this into equation 5.15, a new representation of $\hat{K}(X'_{pool}|f_{nn}, X')$ is derived:

$$\begin{aligned}&\hat{K}(X'_{pool}|f_{nn}, X') \\ &= P - \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} M^{-1} & -M^{-1}CT^{-1} \\ -T^{-1}C^T M^{-1} & T^{-1} + T^{-1}C^T M^{-1}CT^{-1} \end{pmatrix} \begin{pmatrix} A^T \\ B^T \end{pmatrix} \\ &= P - AM^{-1}A^T + BT^{-1}C^T M^{-1}A^T + \\ &\qquad AM^{-1}CT^{-1}B^T - BM^{-1}B^T - BT^{-1}C^T M^{-1}CT^{-1}B^T \\ &= P - BM^{-1}B^T - (A - BT^{-1}C^T)M^{-1}(A^T - CT^{-1}B^T).\end{aligned} \tag{5.17}$$

Comparing this formulation with the already calculated values of $\hat{K}(X_{pool}|f_{nn}, X)$ in equation 5.14 we see that $P - BM^{-1}B^T$ is the old posterior variance of the remaining pool points while $(A - BT^{-1}C^T) = (A^T - CT^{-1}B^T)^T$ are the posterior covariances which we have calculated as well! The only thing that remains to be done is the inversion of M, the lower bottom block of $\hat{K}(X_{pool}|f_{nn}, X)$. Reverting the block matrix notation yields a new updating procedure:

$$\begin{aligned}&\hat{K}_{update} := \hat{K}(X'_{pool}, X_*|f_{nn}, X)\hat{K}(X_*, X_*|f_{nn}, X)^{-1}\hat{K}(X'_{pool}, X_*|f_{nn}, X)^T \\ &\hat{K}(X'_{pool}|f_{nn}, X') = \hat{K}(X'_{pool}|f_{nn}, X) - \hat{K}_{update}.\end{aligned} \tag{5.18}$$

The naive way of calculating the new posterior (eq. 5.15) required the inversion of a matrix of size $|X_*| + |X|$. With the new updating strategy we only invert a matrix of size $|X_*|$. This is beneficial, as the size of the training data $|X|$ is typically big in comparison to the number of requested samples per iteration $|X_*|$. Furthermore, $|X|$ increases every iteration by $|X_*|$ because additional points are added to the training set, while the size of $|X_*|$ does not increase. This result holds as well for the case where only one point is sampled per iteration. Therefore, the updating rule described in the previous section and in particular equation 5.8 follow from here. A similar formula including the regularization term λI can be derived with the same steps (replace T and R by $(T + \lambda I)$ and $(T + \lambda I)$ in equation 5.15, respectively). Now, we formulate the Fast Batch GPA Sampler before continuing with implementation details and experimental results in the next chapter. The pseudocode is very similar to the Batch GPA Sampler (3) and in fact calculates precisely the same variances and sample points. The core difference is the installation of the fast updating procedure in line 13 which reduces the total runtime from roughly $\mathcal{O}((|X| + N)^3(N/M)) = \mathcal{O}((N|X|^3 + N^4)/M)$ to $\mathcal{O}(|X|^3 + NM^2)$. For most applications, $M \ll N \ll |X|$ holds.

Algorithm 5: Fast Batch GPA Sampler for Bayesian Neural Networks

```
Data: number of samples $N$, number of samples per iteration $M$, training
      points $X$, pool $X_{pool}$, neural network model $f_{nn}$ and posterior $p(\theta|\mathcal{D})$,
      number of Monte Carlo samples $S$, regularization parameter $\lambda$
Result: set of points $X_{sample} \subset X_{pool}$, with $|X_{sample}| = N$

1  define $\bar{X} := X \cup X_{pool}$;
2  for $i = 1, \dots, S$ do
3      sample $\theta^i \sim p(\theta|\mathcal{D})$;
4      $y_i = f_{nn}(\bar{X}, \theta^i)$
5  end
6  calculate covariance matrix:
       $$\hat{K}(\bar{X}, \bar{X}) = \frac{1}{S-1} \sum_{i=1}^{S} \left( (y_i - \bar{y})^T \cdot (y_i - \bar{y}) \right), \text{ with } \bar{y} = \frac{1}{S} \sum_{i=1}^{S} y_i$$
7  calculate first posterior covariance matrix:
       $$\hat{K} = \hat{K}(X_{pool}, X_{pool}) - \hat{K}(X_{pool}, X) \left( \hat{K}(X, X) + \lambda I \right)^{-1} \hat{K}(X, X_{pool})$$
8  $X_{samples} \longleftarrow \emptyset$;
9  while $|X_{samples}| < N$ do
10     get posterior variance $\sigma^2_{post}(X_{pool}|f_{nn}, X \cup X_{samples}) = diagonal(\hat{K})$;
11     $X_{new} \longleftarrow$ top $M$ points in $X_{posol}$ according to $\sigma^2_{post}$ ;
12     $X_{samples} \longleftarrow X_{samples} \cup X_{new}$;
13     update posterior covariance matrix:
           $$\hat{K} = \hat{K} - \hat{K}(\bar{X}, X_{new}) \left( \hat{K}(X_{new}, X_{new}) + \lambda I \right)^{-1} \hat{K}(X_{new}, \bar{X})$$
14     $X_{pool} \longleftarrow X_{pool} \setminus \{x^*\}$;
15 end
16 return: $X_{samples}$
```

Chapter 6

Experimental Results

In this chapter the Bayesian neural network from chapter 4 is combined with the active learning approaches from chapter 5. We want to investigate whether the presented active learning sampling policies can keep their promise and in fact do reduce the number of labeled data which is required to achieve an specific error rate. The GPA sampling policies from the last chapter are compared against each other and against other sampling strategies, such as the Variance Sampler (algorithm 2) and the approach presented in [Haf+20].

Before the actual analysis, a short overview over implementation details for the network models and the sampling strategies is given in section 6.1 and two standard regression datasets are introduced in section 6.2. The analysis starts by taking a detailed look at the (Fast) GPA Samplers in section 6.3, reporting convergence results for the Monte Carlo estimates of the covariances as well as empirical runtimes and an analysis of various aspects of the sampling behavior. In section 6.4 different sampling strategies are compared in a typical regression setting, before in sections 6.5 and 6.6 two applications are investigated which fit well into the active learning problem.

6.1 Implementation Details

Overview of technologies: Unless noted otherwise, the models used within this chapter have been implemented in Python ([VD09]) with TensorFlow (`tf`) [Mar+15] and/or TensorFlow-Probability (`tfp`) [Dil+17]. The implementation of the Bayesian neural network with noise contrastive priors uses parts of the code from the corresponding paper [Haf+20]. For the experiments in section 6.5, code was modified from the SchNetPack [Sch+18] which is an open-source implementation of the model developed by them in [Sch+17]. For the experiments in section 6.6, provided models from the respective paper [NGK19b] in PyTorch [Pas+19] are used. The quasi-Monte Carlo methods utilized in the same section stem from the "Magic Point Shop" [KN16]. The implemented models and sampling strategies are presented shortly.

DET: The first network is a feed-forward neural network as described in section 4.1. In contrast to the Bayesian models, the deterministic network does not learn any uncertainty estimations but is trained with respect to the root mean squared error

(RMSE) by the usual backpropagation algorithm. This model was implemented to collect empirical evidence that the presented approaches can also be used outside of the probabilistic setting of BNNs. To do so, an observation made by Gal and Ghahramani in [GG15] is used: They showed that using dropout while training a normal network is actually mathematically equivalent to Bayesian inference with a certain type of Bayesian network. When estimating the (co)variance of the DET network, therefore, the dropout is sampled instead of the posterior of a BNN. This approach also has the big advantage that it can be used with any already implemented deterministic neural network. As we will see in section 6.5 and 6.6, only dropout needs to be added to existing *deterministic* network architectures to apply the samplers which were originally developed for *Bayesian* networks. A word of caution: When estimating the covariance in the first lines of the GPA Samplers, it is important to have the *same* dropout for all forward passes of the network in one iteration. In other words, one Monte Carlo sample should only consist of one dropout configuration. The standard dropout layers in TensorFlow (like `tf.nn.dropout`) and PyTorch use a *different* dropout for each forward pass, so that extra effort is needed at this part of the code.

BNN: This is the implementation of the Bayesian neural network that was introduced in section 4.3. An independent centralized normal distributions is used as prior for the weights and the weight posterior is approximated with independent normal distributions. In TensorFlowProbability (`tfp`) the `distributions` module (`tfd`) can be used to define this distributions via `tfd.Independent(tfd.Normal())`. The layers themselves are described and connected by `tf.layers.dense`. Then, the variational inference on the posterior is added by first defining the prior and variational distributions and then passing them to `tfp.DenseReparameterization()`. This layer uses the representation estimator [KW13] to sample from the posterior and in that sense implements a stochastic forward pass similar to equation 4.13. The output is collected to compute the loss as `tfp.kl_divergence` between variational distribution and prior in addition to the negative log-likelihood. This is precisely the optimization objective given in Bayes by Backprop (refer equation 4.11). Finally, the losses are passed to an Adam optimizer which computes the respective gradients. Following [Tsy+19], the learning rate of the optimizer is reset after each active learning iteration to overcome local minima.

BNN NCP: This network was suggested in [Haf+20] and also used in [Tsy+19]. Normal distributions - which are commonly used as priors for BNNs - only restrict the posterior very weakly, as an alternative they propose noise contrastive priors (NCP). To improve the uncertainty estimates of the network for datapoints which are further away from the training instances, they artificially add out-of-distribution points with high noise to the training process encouraging the network to predict a higher uncertainty for points outside of the training distribution. Parts from their open-source implementation which is available at `https://github.com/brain-research/ncp` (accessed October 12, 2020) were utilized. The implementation is very similar to the BNN above, yet adds another loss term for the out-of-distribution points.

GPA Sampler: The implementation of the different GPA Sampler is rather straightforward and based on matrix operations from the NumPy library [Har+20]. In the implementation of the Fast GPA Sampler it is ensured that the rows and columns of already sampled points are really zeroed and that the symmetry of covariance matrices is asserted. Originally, also the recalculation of the covariance estimates was proposed when the posterior variance got too small. However, after the numerical stability was improved this was not longer necessary.

Infogain Policy: Another active learning policy based on the expected information gain was suggested in [Haf+20]. For a network with posterior $p(\theta|\mathcal{D})$ and a point in the pool $x_* \in X_{pool}$ recall that the distribution over possible labels is given by the network as $p(y_*|x_*) = \int f_{nn}(x_*;\theta)p(\theta|\mathcal{D})d\theta$ (refer equation 4.4). The information gain measures how much new information the addition of the labeled instance (x_*, y_*) would add to the posterior. Since the correct label y_* is unknown, the expected information gain has to be derived by using the distribution given by the network. The resulting acquisition function is:

$$A(x|\mathcal{D}, p(\theta|\mathcal{D})) = \mathbb{E}_{p(y_*|x_*)}\left[KL[p(\theta|\mathcal{D} \cup \{(x_*, y_*)\})||p(\theta|\mathcal{D})]\right]. \tag{6.1}$$

To improve diversity, they pass the information gain through a softmax and sample from the resulting distribution.

6.2 Housing and Airline Datasets

The two standard regression dataset which will be used for analysis in the upcoming sections are shortly outlined in this section.

The **California Housing** dataset was first examined in [KB97]. The dataset is well known and included in the `sklearn.dataset`([Ped+11]) in an already cleaned version. It contains 20,640 instances - each representing a block of 600 to 3,000 people - from the official 1990 U.S. census. The five numerical features {*"median income", "house age", "average number of rooms", "block population", "average house occupancy"*} are used to predict the average housing price. The standard 70%-30% training-validation split results in 14,448 pool and 6,192 validation instances.

The **Airline** or **Flight Delay** dataset was first used by [HFL13] to analyze the performance of Gaussian processes on big data. It was also used in [Haf+20] and [Tsy+19] and can be downloaded directly from the U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics (BTS) (website accessed September 28, 2020). As training data, the first four months of the year 2008 are used and as test data the fifth month. From the about 3,000,000 instances, a subsample of 500,000 points for training and another 500,000 for testing was obtained which do not have any missing information. The seven features { *"Distance", "Month", "DayofMonth", "DayOfWeek", "CRSDepTime", "CRSArrTime", "AirTime"*} are used to predict the total delay (i.e. the sum of *"ArrDelay"* and *"DepDelay"*).

6.3 Behavior of the GPA Samplers

In this section some interesting properties of the presented GPA sampling approaches are investigated. The regression datasets introduced in the last section and the Bayesian neural network implementations as described above are utilized. Unless noted otherwise, a BNN with two layer of 100 neurons each, a training batch size of 16 and a learning rate of 10^{-3} are used. If the runtime is of no importance to the particular experiment, no difference is made between Fast GPA Sampler and GPA Sampler.

6.3.1 Number of Monte Carlo Samples

The first question to answer is how many samples are needed from the posterior of the network to get a good approximation of the covariance of the network. These estimations are essential to calculate the posterior variance of the Gaussian process approximation. We report how this variance changes, as we increase the number of Monte Carlo samples from the network posterior. We limit the number of instances to 5,000, as more samples do not lead to significant changes. The posterior variance calculated on a fraction of the 5,000 samples is compared against the posterior variance based on all the samples. For both datasets, a BNN on 2,500 instances is trained for 2,000 epochs and the posterior variance on 10,000 pool points is calculated.

samples	rel. abs.	corrcoeff
3	0.962	0.38
5	0.931	0.519
10	0.864	0.687
25	0.692	0.883
50	0.488	0.952
100	0.297	0.982
250	0.129	0.995
500	0.064	0.998
750	0.042	0.999
1000	0.03	0.999
2500	0.01	1.0

(a) Housing dataset

samples	rel. abs.	corrcoeff
3	0.981	0.337
5	0.962	0.453
10	0.916	0.625
25	0.786	0.807
50	0.611	0.907
100	0.399	0.96
250	0.183	0.988
500	0.092	0.995
750	0.059	0.997
1000	0.041	0.998
2500	0.012	0.999

(b) Airline dataset

Table 6.1: The average convergence of the posterior variance with number of Monte-Carlo samples over ten random seeds. The number of samples, the relative absolute error with respect to the final variance and the Pearson correlation coefficient are shown. For the latter we only consider the central 90% of the variances to avoid having the coefficient dominated by a single large value.

Table 6.1 shows the results over several random seeds. There is no major difference between the two datasets, yet the convergence appears to be faster on the Housing dataset. In general, 100 Monte Carlo samples from the network posterior already suffice to approximate the posterior variance very well and a multiple has to be added to improve the convergence further. Therefore, 100 samples are used in the following experiments.

6.3.2 Empirical Runtimes of the GPA Samplers

To analyze the actual runtime of the different GPA samplers, several experiments on a local machine with an Intel Core i7-10510U processor (8 threads, 4.90 GHz max frequency) are conducted. In the first two experiments different numbers of requested samples are compared while the number of instances in the pool and training set varies in the third and fourth experiment.

To start, the Fast GPA Sampler (batch size = 1, algorithm 4) is compared with the Batch GPA Sampler (batch size = 1 and batch size 10, algorithm 3) and a "real" Gaussian process as a baseline, namely the `GaussianProcessRegressor` (GPR) from sklearn [Ped+11]. As this experiment only considers runtime, the GPR is allowed to read the true labels of the selected points. This allows for re-fitting in each iteration on the updated training data, before the posterior variance of the remaining pool points is predicted as acquisition function. In the second experiment, the Batch GPA Sampler and the Fast Batch GPA Sampler with a batch size of 25 are compared.

In figures 6.1 and 6.2 the results for the Housing data are reported. Similar plots for the Airline dataset can be found in the appendix (section A.1). Taking a look at the first figure, we note that - as expected - the Fast GPA Sampler performs magnitudes faster compared to the other two. For 100 samples, its average runtime is only 0.7 seconds which is more than 25 times faster than the GPA Sampler and the Gaussian Process and also faster than the Batch GPA Sampler with a batch size of 10. This can most likely be traced back to the fact that the Fast version only computes *one* matrix inverse at the beginning while the other calculates it *every* iteration. Therefore, we can conclude that the name *Fast* is justified and the presented approach of Fast GPA is indeed superior to the Batch GPA.

Larger batch sizes are compared in figure 6.2. Observe that the difference of batch sizes (1, 10 or 25) directly reflects in the runtime of the respective sampler. The Fast GPA Sampler which re-evaluates every iteration has a similar runtime as the Batch GPA Sampler with a batch size of 10, yet the newly introduced Fast Batch GPA Sampler outperforms both by a large margin. Note as well that the runtime of all approaches scales linearly with the number of requested samples. This would be expected as mainly the *number of iterations* increases, however not the *effort per iteration*.

For comparison, consider figures 6.3 and 6.4 which show the scaling behavior of the runtime relative to the size of the pool and training data. It can be seen that the runtime scales worse than linear. This is not surprising, as bigger training and pool sizes result in a bigger covariance matrix slowing down all the necessary matrix operations. Comparing the runtime of the Batch GPA Sampler (in orange) in two figures, it appears to scale even worse with the training size than with the pool size. This stems from the fact that a matrix of roughly the training size has to be inverted in each iteration and therefore cubic scaling can be expected. The fast updating procedure (in blue) is able to flatten this curve. It scales worse than linear, as an initial matrix inversion and ongoing matrix arithmetic remain necessary.

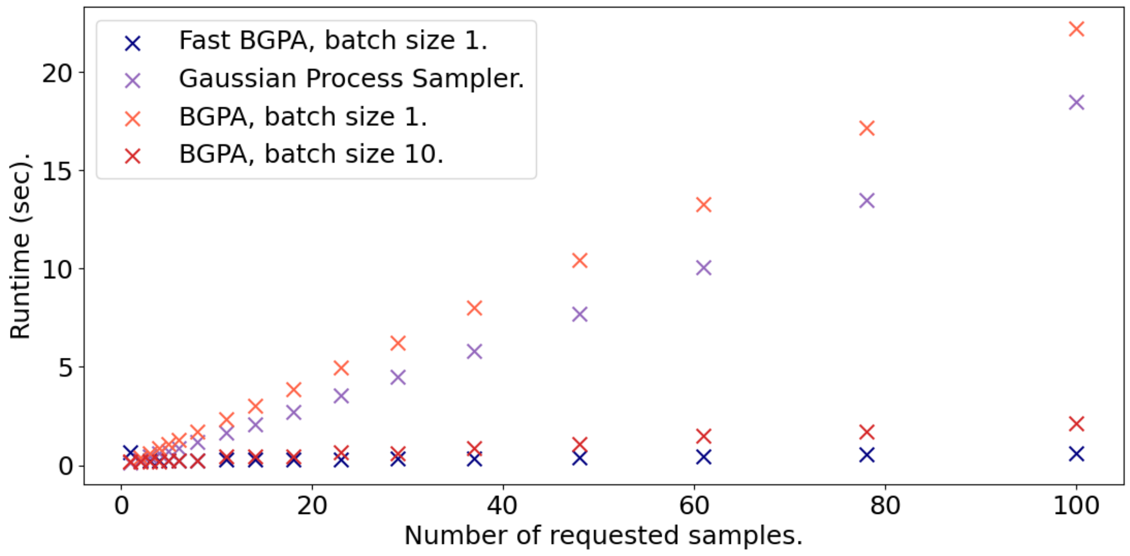

Figure 6.1: Runtime in seconds of Fast GPA Sampler, Batch GPA Sampler (batch size = 1 and batch size = 10) and the policy based on a Gaussian process for different numbers of requested samples on the Housing dataset. We report the average runtime over ten random seeds, for which we also repeat the initial data selection and network initialization. In this first experiment the network is trained for 1,000 epochs on an initial set of 250 points and a maximal amount of 100 points is requested from the policies.

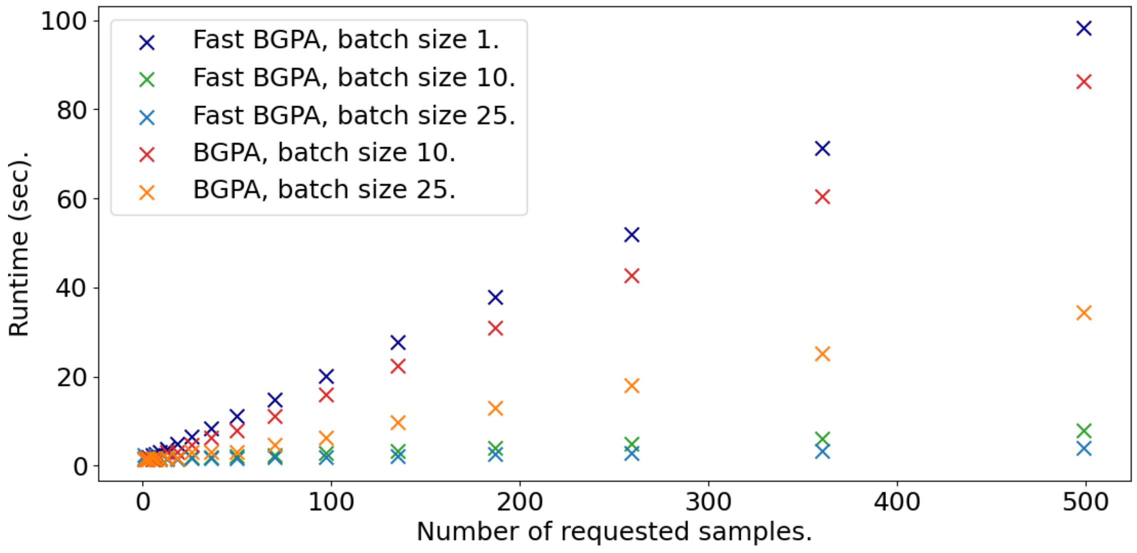

Figure 6.2: Runtime in seconds of Fast Batch GPA Sampler and Batch GPA Sampler for different numbers of requested samples on the Housing dataset. Again, the runtime is averaged over ten seeds. In this experiment, we train for 2,000 epochs on 1,000 training instances, before we request up to a maximum of 500 points from the policies.

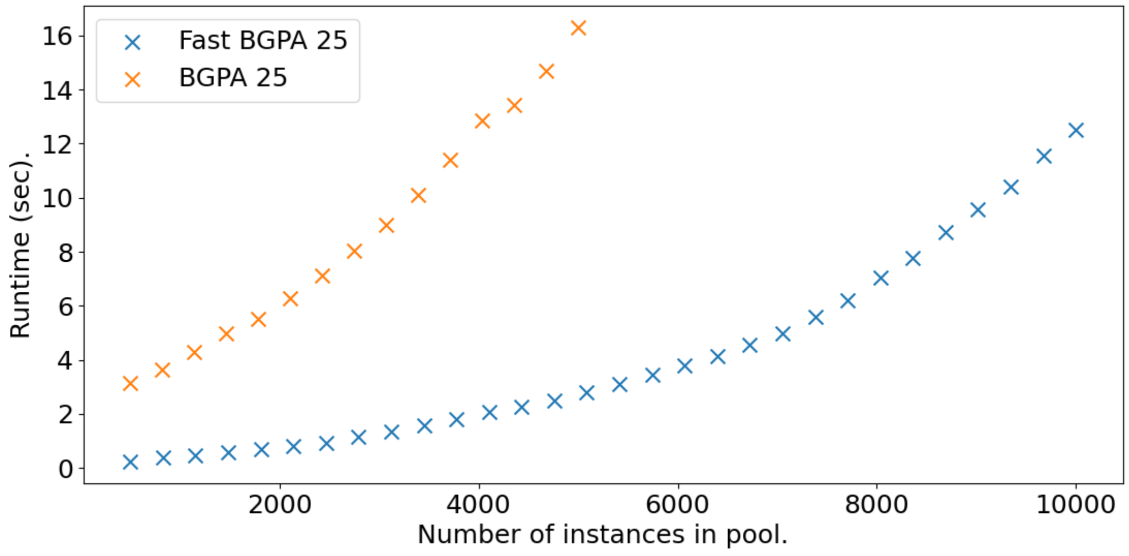

Figure 6.3: Runtime in seconds for Fast Batch GPA Sampler and Batch GPA Sampler with batch size 25 for different pool sizes (average over ten seeds). The training size is fixed at 1,000 instances and the network is trained for 1,000 epochs before requesting 250 samples from the pool.

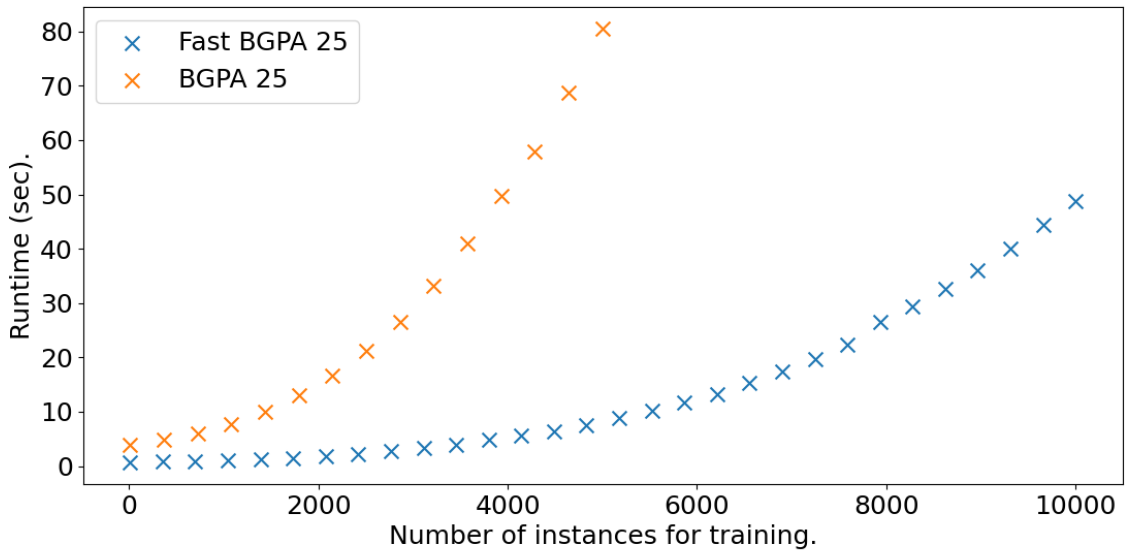

Figure 6.4: Runtime in seconds for Fast Batch GPA Sampler and Batch GPA Sampler with batch size 25 for different train sizes (average over ten seeds). The pool size is fixed at 2,500 instances and the network is trained for 1,000 epochs before requesting 250 samples from the pool.

6.3.3 Influence of the Regularization Parameter λ

Another variable for all GPA Sampler is the regularization parameter λ. Recall that this parameter was introduced in equation 5.10 to increase numerical stability and that it was derived from the formula for the posterior variance of a Gaussian process with the additive noise assumption (refer to equation 3.10). The *posterior* variance calculated by the Fast GPA Sampler with different choices of λ is compared to the *sample* variance. The setup of the active learning framework includes 5,000 pool points and 500 training points which are increased by 250 instances over four active learning iteration to 1,500 training points total. In each iteration the network is trained for 1,000 epochs and the samples are selected based on the Fast GPA sampler with a batch size of 1 and a λ of 0.1.

samples/λ	0.0001	0.001	0.01	0.05	0.1	0.5	1	5	10
500	0.38	0.44	0.54	0.64	0.68	0.79	0.85	0.96	0.98
750	0.15	0.21	0.35	0.47	0.52	0.68	0.76	0.9	0.94
1000	0.04	0.09	0.23	0.37	0.44	0.62	0.71	0.89	0.93
1250	-0.04	0.0	0.15	0.31	0.38	0.56	0.65	0.86	0.91
1500	-0.12	-0.08	0.07	0.25	0.34	0.55	0.65	0.86	0.91

Table 6.2: Correlation coefficient of posterior variance by Fast GPA Sampler and sample variance of the BNN on the Housing dataset. Average over ten random seeds.

samples/λ	0.0001	0.001	0.01	0.05	0.1	0.5	1	5	10
500	0.63	0.72	0.83	0.9	0.93	0.98	0.99	1.0	1.0
750	0.45	0.52	0.66	0.79	0.84	0.94	0.97	1.0	1.0
1000	0.25	0.3	0.44	0.6	0.68	0.87	0.93	0.99	1.0
1250	0.19	0.24	0.4	0.58	0.66	0.86	0.92	0.99	1.0
1500	0.1	0.15	0.32	0.52	0.62	0.83	0.9	0.99	0.99

Table 6.3: Correlation coefficient of posterior variance by Fast GPA Sampler and sample variance of the BNN on the Airline dataset. Average over ten random seeds.

Reported in tables 6.2 and 6.3 are correlation coefficients between the posterior variance and the sample variance for different choices of λ after the initial training and over a period of four active learning iterations. Observe that the correlation coefficient grows (i.e. the distributions become more similar) with the larger sizes of λ and decreases with the number of active learning iterations. To understand the first correlation, recall that the *difference* between the posterior and the sample variance includes the inverse of $\hat{K}(X, X)+\lambda I$. For large λ this term is dominated by λI. The inverse will be roughly $1/\lambda I$ which results in a low difference and therefore in a high similarity between the posterior and the sample variance. However, when the number of training samples is increased during the active learning iterations the number of covariances on which the posterior is conditioned on increases accordingly. This moves the posterior further away from the sample variance. Comparing the two datasets we can see that the correlation is generally stronger on the Airline dataset. This might result from the fact that the BNN predicts a higher uncertainty for the instances from the Housing dataset (maximum value about 2) compared to the Airline dataset (maximum value about 0.2) and therefore the influence of λ is smaller. Relative to the size of the variance, the columns are more similar. Compare for example columns $\lambda = 0.1$ Housing and $\lambda = 0.01$ Airline.

The formulation of the updating procedure of the Fast GPA Sampler in [Tsy+19] section 2.4 does not take the regularization by λ into account. If the updating formula is implemented by dividing the update matrix only by $\hat{\sigma}^2(x * | f_{nn}, X)$ instead of $\hat{\sigma}^2(x^*|f_{nn}, X) + \lambda$, this can lead to a degeneration of the updated covariances. As an example for this behavior, consider figure 6.5 where we compare the updating rule with and without taking λ into account in the updating step.

Again, a network is trained for 1,000 epochs on 1,000 initial points before 1,000 samples are requested from a pool of size 5,000. The *correct* posterior variance is calculated in each iteration directly (i.e. by using equation 5.8) and compared with the updating rule neglecting λ (upper plots) as well as the updating rule taking λ into account (lower plots). Evidently, the lower plots show that the updating rule with λ calculates precisely the correct values even after 1,000 iterations (up to five digits after the comma). The naive updating rule demonstrates the dangerous behavior without proper regularization: Already after around 100 iterations the fast updating results in a posterior significantly smaller than the true posterior (note the logarithmic scale on the y-axis of the right hand side plots). After around 200 iterations the posterior becomes effectively useless: its maximum drops to nearly zero and it contains no useful information. This can be seen by the oscillations of the correlation coefficient around zero on the left-hand side.

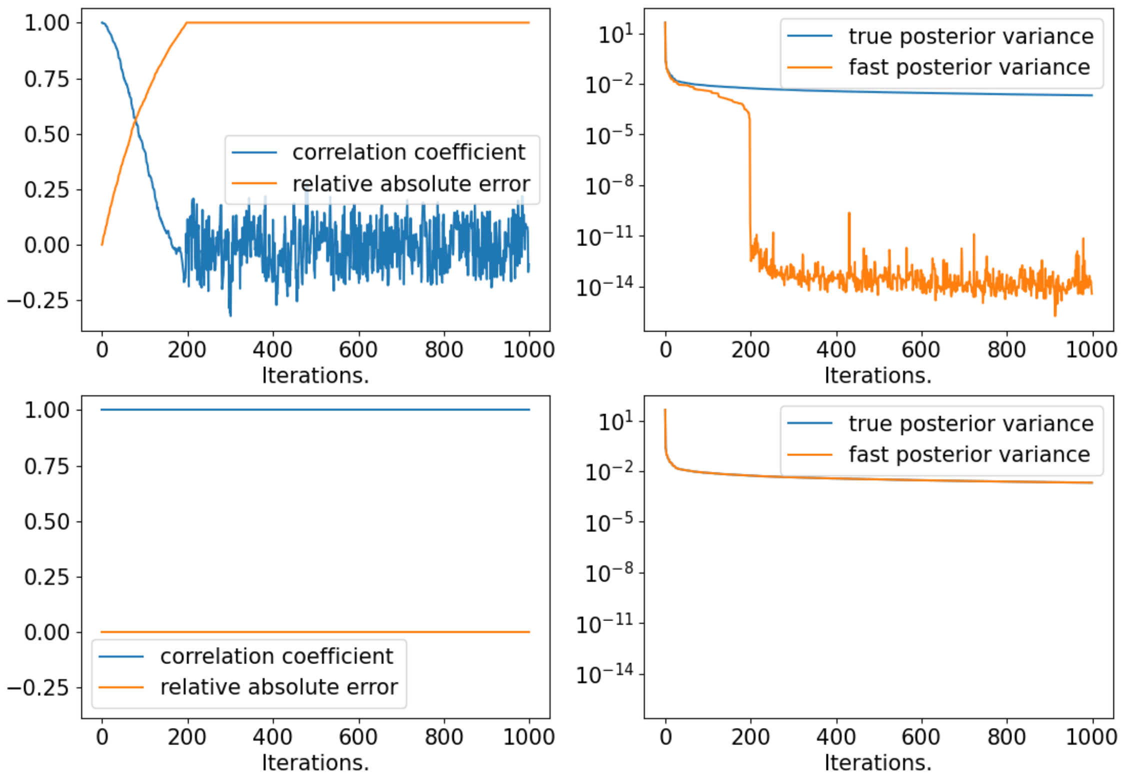

Figure 6.5: **Upper** plots: comparison of the posterior variance from the Fast GPA Sampler without λ in the updating step to the directly calculated value for $\lambda = 0.1$. **Lower** plots: fast updating step with $\lambda = 0.1$. **Left** side: correlation coefficient and absolute error of the fast version compared to the directly calculated value. **Right** side: maximum variance of the true and the fast posterior in each iteration. Note the logarithmic y-scale.

6.3.4 GPA Sampler and Gaussian Processes

The GPA Sampler was motivated by the connection between Gaussian processes and neural networks. Recall that the posterior variance of a Gaussian process is calculated in the GPA Sampler and that this GP is defined by Monte Carlo estimates from the trained BNN. This means that the GP is never trained on the data directly but indirectly via the network. In theory, the network should transfer its knowledge and the GP should fit the data reasonably well. We want to see how close this Gaussian process is to a Gaussian process which is trained on the data directly.

Once again the `GaussianProcessRegressor` from sklearn is used with an RBF kernel and fit on the training data. Because the GPA Sampler computes no predictions, we can only compare the posterior variances. Table 6.4 shows the correlation coefficient between the posterior variance of the Gaussian Process fitted on the training data and the posterior variance calculated by the GPA Sampler for different numbers of Monte Carlo samples and λ. We observe that the correlation increases with the number of samples and reaches its peak for $\lambda = 0.1$. It is interesting however that for the selected values of 100 samples and $\lambda \in [0.01, 0.1]$ the correlation coefficient is very high with 0.75. This means that the approach used in the GPA Sampler - which never considers a Gaussian process directly - indeed calculates a posterior variance similar to an actual Gaussian process. To compare, the sample variance of the BNN only has a correlation coefficient of 0.47.

samples/ λ	0.0001	0.001	0.01	0.1	1	5
3	0.35	0.4	0.35	0.4	0.32	0.34
5	0.46	0.42	0.44	0.39	0.4	0.45
10	0.57	0.56	0.54	0.58	0.53	0.53
50	0.74	0.73	0.73	0.73	0.68	0.61
100	0.73	0.73	0.74	0.75	0.7	0.62
250	0.7	0.71	0.75	0.75	0.7	0.64
500	0.67	0.7	0.75	0.76	0.71	0.63

Table 6.4: Correlation coefficient of GaussianProcessRegressor and GPA Sampler posterior variance for Housing dataset. Average over ten random seeds. The BNN is trained for 2,000 epochs on 1,000 train points. The pool consists of 5,000 instances.

6.3.5 Diversity of Sampled Points

The development of the GPA Sampler was mainly motivated by the concern that the Variance Sampler (algorithm 2) would select to many similar points if multiple points per iteration are requested. Recall that the this sampler simply selects the points with the highest variance estimation. We compare the Variance Sampler with the GPA Sampler, with randomly selected points and also with points selected by a Gaussian process which is re-fit after the addition of each point. A Bayesian neural network is trained on 500 points for 2,000 epochs and 50 points are requested from a pool of 5,000 points. A t-distributed stochastic neighborhood embedding (t-SNE) from [MH08] with default parameters from sklearn is used to visualize the five-dimensional instances from the Housing dataset.

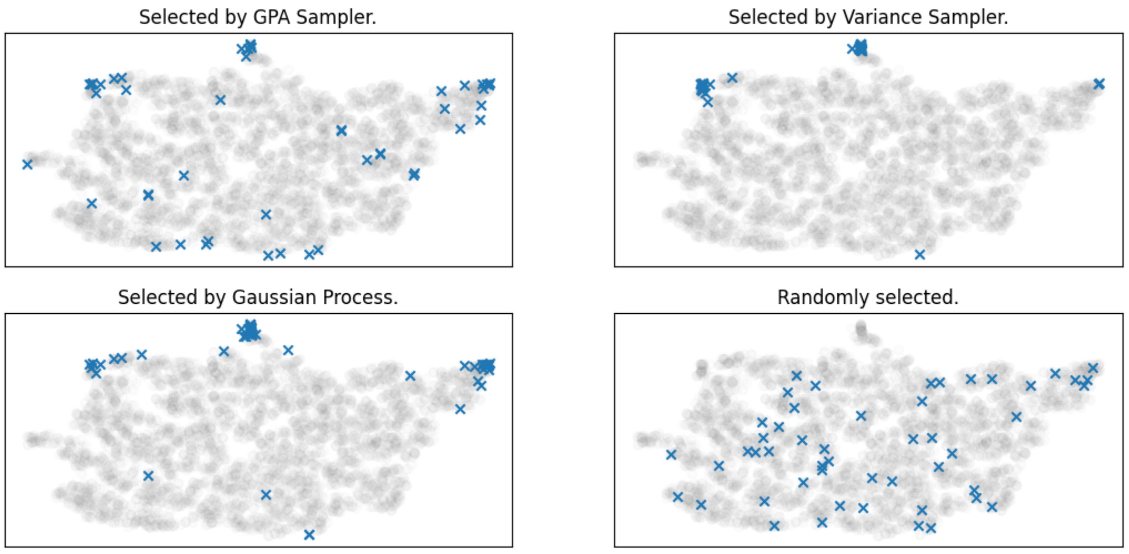

Figure 6.6: The 5,000 pool instances are represented by transparent gray circles. Blue crosses indicate the 50 instances selected by the different Samplers.

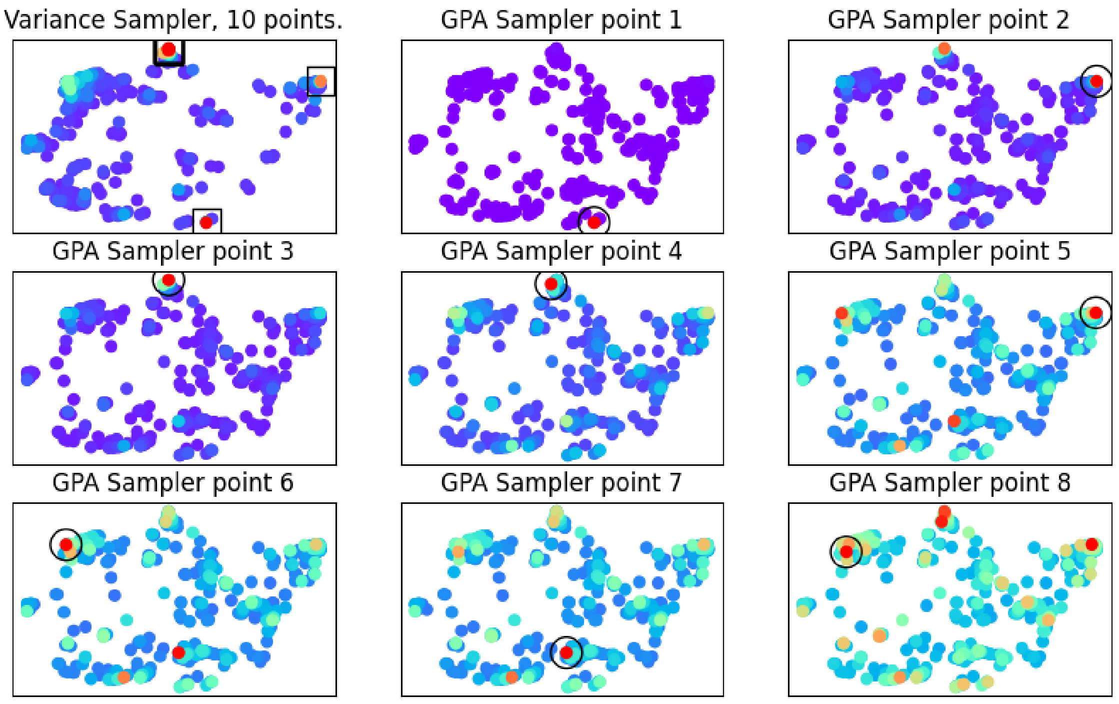

Figure 6.7: **Top left**: first 10 selected points by the Variance Sampler. **Remaining plots**: first 8 points selected by the GPA Sampler. **All plots**: top 500 points from the pool with the highest acquisition function. Colors indicate the value of the acquisition function (0 = blue < green < red = highest acquisition value).

Considering the top right plot in figure 6.6 we see that the Variance Sampler indeed selects very similar points: Out of 50 points, only three (one in the bottom, two on the right) are not contained in two big clusters. This is suboptimal as a much smaller number of points might suffice to get the same information from labeling. In comparison, the GPA Sampler and the Gaussian process are very similar and both select more diverse points. Both decide to sample points from the three apparently interesting/difficult areas on the top but they also spend resources outside of them. The GPA Sampler selects even more diverse points and in fact covers all the points the Gaussian process selected in the lower half of the plot. Taking a look at the randomly selected points we observe that this sampling method completely misses the areas on the top which appear to be important to the other samplers. Keeping in mind that this example might not be representative we conclude that the GPA Sampler appears to find a sweet spot between selecting points with a high variance while maintaining diversity over the complete sample.

Figure 6.7 is intended to give an intuition how this is possible by showing the selection process for the first points of the Variance and GPA Sampler. Consider first the plot on the top left. The 500 pool points with the highest sample variance are shown. Their color indicates the respective variance (blue=small, red=high). The squares indicate the ten points with the highest variance that are therefore selected by the Variance Sampler. We observe that the points only stem from three regions and as we know from the figure above, the next 40 points are located in two clusters on the top-left.

The remaining plots show the decision process of the GPA Sampler. After selecting the points with the highest posterior variance, the updated variance is calculated and another point is sampled. Here, two close points are rarely selected consecutively. Also note the change of color: in the first iteration, it is very clear which point has the highest posterior variance (only one is red). After the addition of more points however, the variances become more similar (more green and red points). This is caused by the largest variances becoming smaller letting the other variances appear larger. Concluding we can say that, at least in this particular example, the GPA Sampler acts precisely in the way it is intended to work leading to the selection of interesting but also diverse points.

6.4 Complete Active Learning Framework

In this section the performance of different sampling policies is compared in the active learning framework that was presented in section 2.2. To mimic a real active learning application with the Housing and the Airline dataset, we define a set of points for initial training and a set of unlabeled pool instances. Multiple active learning iterations are performed in which the labels to the selected points is provided for further training. The first two experiment compare sampling strategies, while the third and fourth apply fine-tuning and different network models.

The first setup uses the Airline dataset and is similar to section 5.2 in [Haf+20] and section 3.1 in [Tsy+19]. The BNN NCP from [Haf+20] is trained initially on 50 points. Afterwards, ten active learning iterations are performed adding 50 points from a pool of 25,000 instances each and continuing the training for 1,000 epochs. Sampling based on the Fast Batch GPA Sampler (batch size 5), the Variance Sampler and the Infogain policy are compared to random selection. The errors computed on a validation set of 10,000 points are shown in figure 6.8. After the 1,000 epochs of initial training, the Fast GPA and the Variance Sampler seem to reduce the error faster in the first active learning iterations. However, after around five iterations and a total of 6,000 training epochs the error becomes very similar. In the final iterations the errors are indistinguishable.

A larger scale experiment was performed on the Housing dataset. This time, a BNN with two layers of 100 neurons is trained on an initial set of 500 points and 250 points are selected in each active learning iteration. Results are presented in figure 6.9. Here, the validation error drops significantly after the first addition of points for all sampling strategies with the exception of random sampling. The Fast GPA Sampler enables the network the achieve a validation RMSE of about 0.6 which can not be reached with the other methods. In fact, the instances selected by the Variance Sampler *increase* the error after active learning iteration two. We suspect that this is caused by overfitting the training data, as the results are an average over multiple random seeds.

An idea called *fine-tuning* suggests that, after new instances are selected, the network should be trained on them for habituation. The intuition behind is that the network is already used to the old training data, while the added data contains new information which thus should be presented more directly. We performed the experiment on the Housing dataset again with the Fast GPA Sampler to investigate if fine-tuning leads to faster or better convergence. In each active learning iteration and after additional data was sampled, the network is first trained on the new instances for 200 epochs before the training continues on all the available training data for 800 epochs.

This approach is compared against 1,000 epochs of training on the complete training data in figure 6.10. Although the idea is intuitive, the conducted experiment contributes no evidence that fine-tuning can improve upon standard training. The 200 epochs trained on the newly sampled data can be clearly observed as the peaks of the orange line. They seem to increase the validation error and further training on the complete dataset has to decrease it again to the previous level. Nevertheless, fine-tuning might have potential when the training process is stuck in a local minimum or when the validation error can be reduced further.

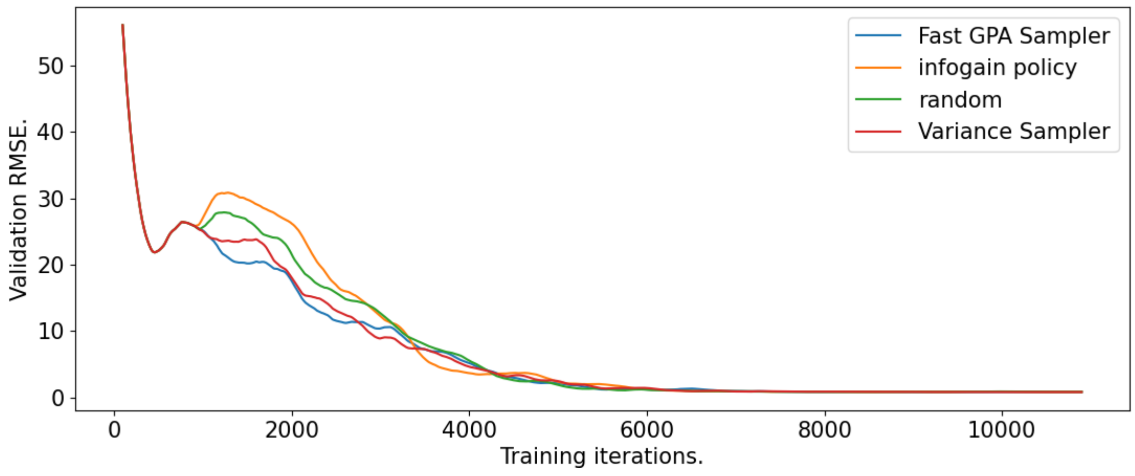

Figure 6.8: Active learning on the Airline dataset. BNN NCP with two layers of 50 neurons, initial training on 50 points, then ten active learning iterations adding 50 points each. Training consists of 1,000 epochs in both cases. Average over five random seeds.

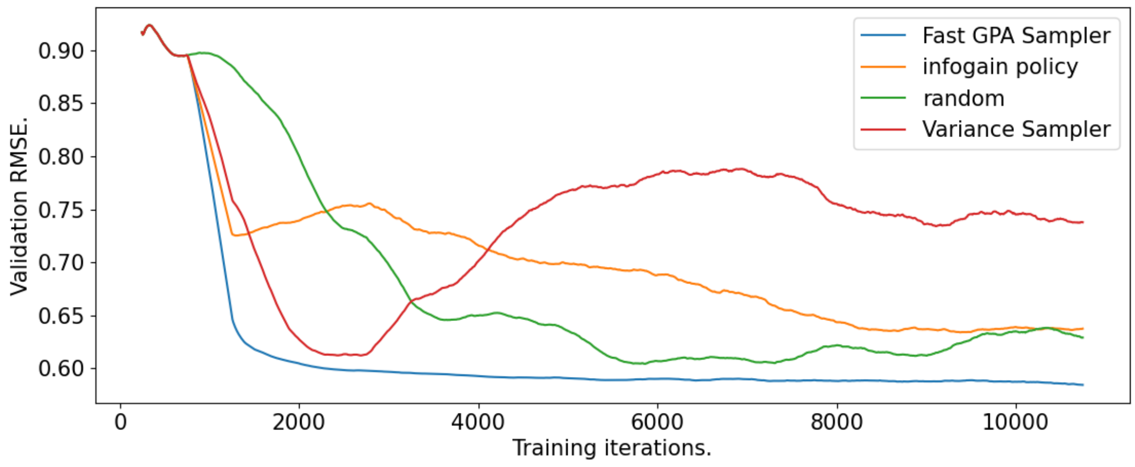

Figure 6.9: Active learning on the Housing dataset. BNN with two layers of 100 neurons, initial training on 500 points, then ten active learning iterations adding 250 points each. Average over five random seeds.

To showcase that the GPA Sampler can also work well together with non-Bayesian neural network a fourth experiment is performed. Again, the setup is similar to the second experiment and uses the Fast GPA Sampler implementation yet the number of initial training instances as well as the number of added instances per iteration are changed to 50. Because the DET network is deterministic, Monte Carlo samples of the dropout are used to estimate the covariance for the GPA Sampler.

The results in figure 6.11 clearly indicate that the active learning framework works well for all three network models. Comparing with figure 6.9, the validation error decreases steadily. An exception is the initial training of the DET network which shows a significant overfitting which, however, is overcome in the active learning iterations. This motivates the usage of the GPA Sampler in the next two chapters where is will be employed in combination with existing and deterministic network architectures.

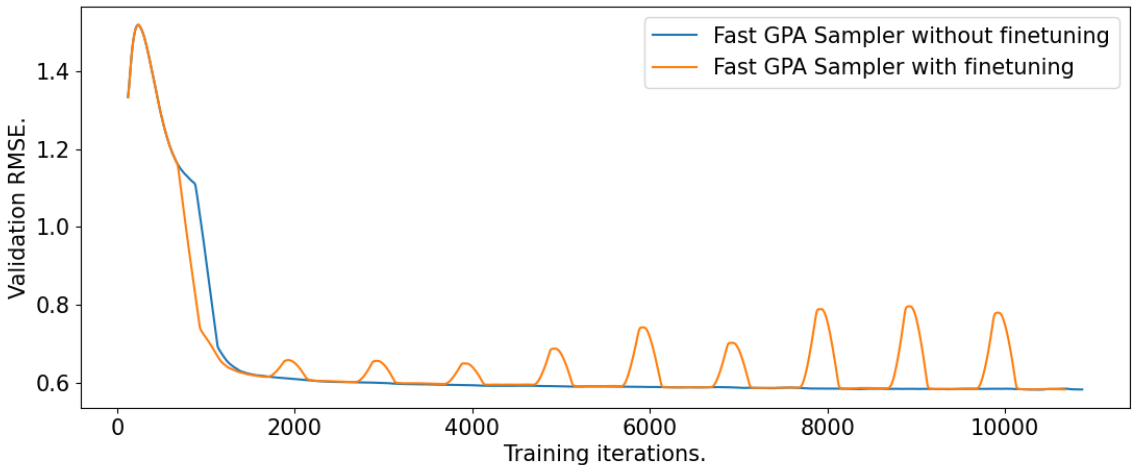

Figure 6.10: Setup as in figure 6.9. The blue line shows the validation error when training the BNN for 1,000 epochs on the complete training data. In orange, the network is first trained 200 epochs on the newly selected data before training for 800 epochs on the complete training data. Average over five random seeds.

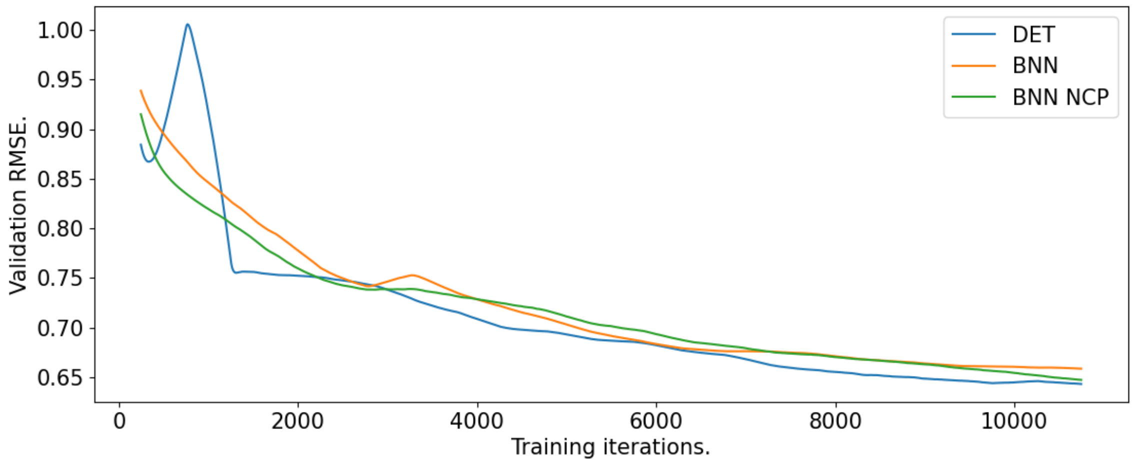

Figure 6.11: The three models presented in section 6.1 are compared in the setup from figure 6.9. The number of initial training points as well as the number of added points are changed to 50. Average over five random seeds.

6.5 QM9 Dataset and SchNet

In this section the effectiveness of active learning is analyzed in an application from the field of quantum chemistry also considered by [Tsy+19]. A state-of-art network called *SchNet* from [Sch+17] is applied to predict the internal energy of different molecules from the QM9 dataset ([Rud+12] and [Ram+14]). We use the implementation available from *SchNetPack* ([Sch+18]) and modify it to fit into the framework we developed. Active learning is of interest for this application because the training process of a large network like SchNet is computationally expensive so that a faster convergence with fewer training instances can lead to significant time savings.

6.5.1 Setup

The **QM9 dataset** contains detailed information about 134,000 organic molecules and is available at http://quantum-machine.org/ (accessed September 28, 2020). It is a widely used benchmark for predicting molecular properties. The contained molecules are in equilibrium which means that their atoms are arranged in such a way that the internal energy lies in a local minimum. The aim is to predict the amount of internal energy based on molecule information like the compounds, nuclear charges and atomic position.

The **SchNet** processes this information by employing different types of layers. First, an embedding layer is applied which maps the molecule information to a feature space. From there, interaction blocks are applied which model the interactions between the atoms of the molecule. These blocks contain so called *cfconv* layers which are convolutional layers that are applied atom-wise to the molecule information and which consist of radial basis and other layers. The training loss includes energy and forces of the molecule and an activation function is chosen in such a way that gradient descent can be used for training. For more details refer to [Sch+17].

As the architecture is rather complex, the SchNet implementation available within the **SchNetPack** and the default parameters are used. The implementation is based on PyTorch ([Pas+19]) and GPU training over several hours is necessary to allow for meaningful training. The SchNet itself is not a Bayesian neural network but it contains dense layers within the interaction blocks to which dropout can be applied to make the output probabilistic. As noted by [GG15], using dropout at the inference stage is equivalent to the inference on a certain BNN. As described before, the dropout is kept constant for a complete forward pass of all molecules in the pool in order to estimate the intended covariances (refer to lines 2-5 in algorithm 5).

6.5.2 Results

As stated, the implementation from the SchNetPack with default parameters is used as a model. The added dropout of 30% is only used for the estimations in the GPA Sampler (dropout during training increases the error rates). 5,000 points are used for validation, 15,000 as pool and the training was performed over 200 epochs on an initial set of 1,000 points. Then, six active learning iterations are performed, each consisting of the addition of 1,000 points and 200 epochs of training. Due to limited computational resources, the Fast Batch GPA Sampler is only compared to random sampling and not with other sampling strategies. The Fast Batch GPA Sampler uses a batch size of 10, regularization parameter $\lambda = 0.1$ and 25 Monte Carlo samples as calculating a single forward pass for the complete pool can already take minutes.

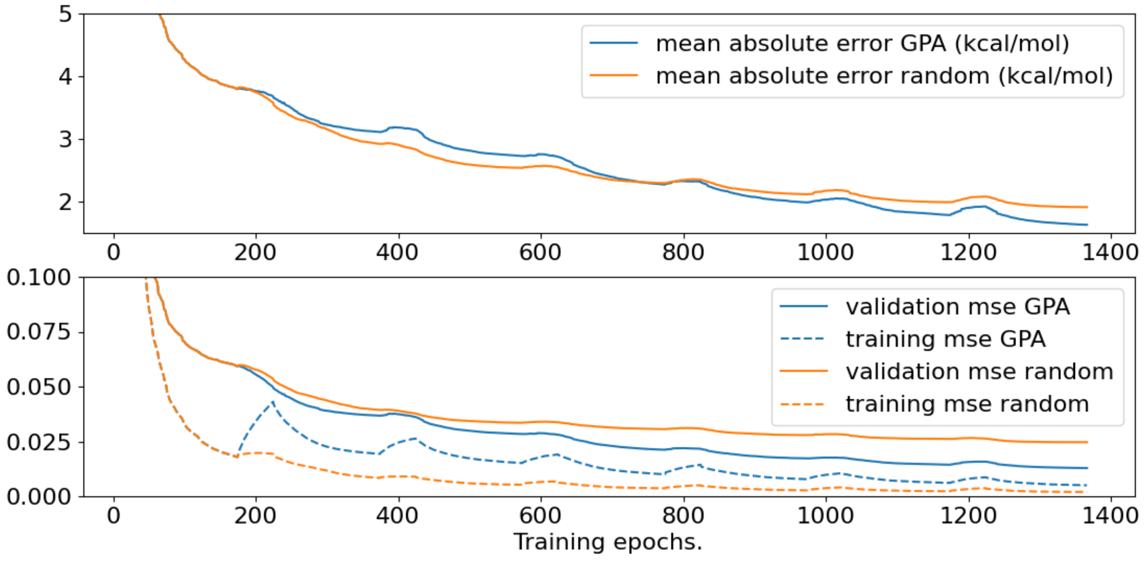

Figure 6.12: Active learning of a SchNet with random sampling (orange) and Fast GPA Sampler (blue, batch size = 10, $\lambda = 0.1$, 25 MC samples). 15,000 points in pool, 1,000 initial training, six active learning iterations with 1,000 points and 200 epochs training. (Plots were smoothed with a rolling average of 50 epochs, therefore their peaks appear earlier and plots do not reach to the very end.)

The results are shown in figure 6.12. The lower plot shows the training and validation errors of both approaches. After 200 epochs the addition of the first points from the pool increases the training error of the SchNet with GPA Sampler. This could be interpreted as the GPA Sampler adding molecules to the training data for which it is particularly hard to predict the correct values. Indeed, this peaking behavior repeats in every active learning iteration and is much less pronounced with the random points. We can also observe that the SchNet training error on the random points quickly converges towards zero while the validation error is magnitudes larger, indicating overfitting. In fact, the final training error for the random points is more than 12 times smaller than the validation error (0.002 vs 0.025). For the GPA sampled points, the SchNet does not overfit so heavily and the final errors are more similar (0.005 vs 0.013).

Besides reducing the validation error after having trained on the same number of points for the same number of epochs by 48%, we can see in the upper plot that the GPA Sampler helps the SchNet to make better predictions for the amount of internal energy. After around 800 epochs the SchNet with GPA sampler surpasses the SchNet with randomly sampled points. The final evaluation shows an improvement of about 15% (1.91 kcal/mol to 1.63 kcal/mol) and the form of the curve suggests that even more improvements could be made with further training.

6.6 HJB Equations

Another interesting opportunity for active learning occurs when additional data can be obtained from computer simulations at runtime, as in the setting presented by [NGK19a]. They take advantage of a neural network as part of the solution process of an optimal control problem (OCP) and employ an active learning framework to pick train points this network. Their published code was used in combination with our implementation of the GPA sampler. The modifications we made include: 1. adding dropout to the network (only for the point selection with the GPA Sampler), 2. increasing the number of initial random points, 3. addition of the option to sample points based on quasi-Monte Carlo techniques from [KN16].

6.6.1 Setup

In their work the authors consider a general optimal control problem: Depending on a state x and a time $t \in [0, t_{max}]$, a control vector field $u^*(x, t)$ has to be computed which minimizes a cost functional $J[u(.)]$ subject to initial conditions and state dynamics. For solving the OCP it can be shown that the optimal control can be obtained from minimizing a Hamiltonial-Jacob-Bellman (HJB) equation if the gradient of the optimal cost function $V(t, x) = J[u^*(.)]$ with respect to the state $V_x(.)$ is available. Under certain (and assumed) conditions and for a fixed initial state, this can be done by solving a two point boundary value problem (BVP). Therefore, a global solution can be approximated by either solving the BVP on a large grid and interpolation, or by solving the BVP online. Although the BVP can be solved reasonably fast by using the time-marching trick, the number of points required for a good interpolation becomes very large if the state space is high dimensional.

Because neural networks generally deal well with high dimensional data, they suggest an improvement by using a neural network for calculating the value function V and the gradient V_x . To obtain labeled training data (i.e. pairs $(x; (V(x), V_x(x)))$ for a fixed state) the BVP can be solved and after the network is trained, optimal controls can be obtained almost in real time. They train the network with respect to a loss function which consists of the squared loss of the value function and the squared loss of the gradient as well. The gradient is not predicted as a value or based on finite differences but by automatic differentiation which is the calculation as derivative of the arithmetic operations which the computer program executes.

This setup fits perfectly in an active learning framework: for any point in the state space the corresponding BVP can be solved to obtain its label and thereby increase the number of instances for the network training. However, this calculation requires the investment of computational resources and the state space is potentially very large. Therefore, a successful active learner can improve the network performance with much fewer samples (compared to random sampling or a grid approximation) by only choosing states which are useful for the training process. As the state space is continuous, the authors suggest to first sample a bigger initial set of states randomly and then select states from that finite set of points. In that sense their approach is a reduction from the query-based active learning to the pool-based version (refer section 2.3.2).

In the presented work they used the norm of the predicted gradient as an acquisition function with the motivation that points with a large gradient should lie in areas which are difficult to approximate and obtaining their labels should therefore contribute well to the training process of the neural network. They test this approach on an OCP which stems from a rigid body model of a satellite. It was originally studied by [KW16] who used a sparse grid method to approximate the value function. A detailed description of the cost functional and the problem dynamics is given in section 5 of [NGK19b]. The OCP has three positional and three velocity related states which are controlled by a three-dimensional torque. Furthermore, it is time independent and the corresponding BVP is solved with a three-stage Lobatto IIIa algorithm ([KS01]).

The authors compared their active learning approach to the sparse grid solution from [KW16]. Another baseline can be established by comparing with random sampling. The framework presented by [NGK19b] is slightly different from what we have seen so far. When N points from the pool are requested, they sample N sets of C points and select one point from each set based on the gradient norm. In their work, they used $C = 2$ yet we increase this value to $C = 10$ to clearly distinguish their approach from random sampling. As the GPA Sampler needs a consistent set of points the framework is slightly changes when using it to sampling all CN points in advance and then selecting points from this bigger set. It is not clear if this poses an advantage or disadvantage: on the one hand, knowing all the points in advance gives more flexibility and a better overview, on the other hand, being able to select one point per iteration already ensures a certain amount of diversity by design.

6.6.2 Results

The first experiment is closer to the framework which was developed by the authors. In particular, the number of points requested in each active learning iteration is decided at runtime. For more details refer to section 4.1 in [NGK19b]. Thus, the number of total training instances varies between each run and random seed. The number of training iterations varies as well because the second order method which is used for training the network (L-BFGS, [Byr+95]) tends to converge before the maximum number of iterations is reached. For validation, 5,000 additional points are generated.

Results for the first experiment are shown in table 6.5 and figure 6.13. After five active learning iterations, the final validation errors are, on average, below the 0.0037 reported by [KW16]. This is especially remarkable because the sparse grid approach used about 45,000 points while for the network fewer than 1,000 suffice to achieve a similar error rate. We observe that the validation errors (crosses) of all approaches decrease nicely, even on a logarithmic scale. This can most likely be traced back to the fact that the number of training points increases roughly exponentially as well: After initial training on 64 points, the number of training instances increases on average by a factor between 1.7 and 1.9. This factor is closer to 1.7 for the randomly selected points and closer to 1.9 for the GPA Sampler. Therefore, the number of training instances after five active learning iterations differs significantly

(577 vs 989). As the GPA Sampler does not lead to the same amount of overfitting, as the other methods, we suspect that presenting more difficult training instances to the network causes the framework to add more points. The larger amount of training points increases the number of training iteration but also reduces the final validation error significantly by almost 30% (from 0.0024 random to 0.0036 GPA). However, the approach from the paper does train on more points compared to the random approach yet achieves a worse final error.

sample mode	train error	validation error	training iterations	number points total
GPA	0.002	0.0024	54569	989
random	0.002	0.0034	45610	577
paper	0.0017	0.0036	52802	726

Table 6.5: Final statistics in figure 6.13, average over five random seeds.

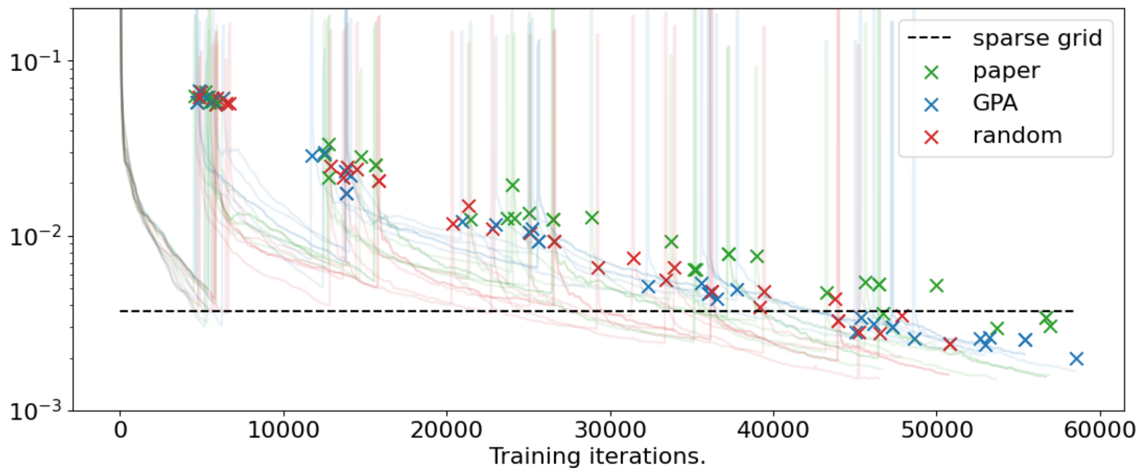

Figure 6.13: Training and validation error of the neural network when choosing points randomly (red), as suggested in the paper (green) and by the GPA Sampler (blue) in five active learning iterations. Validation error (crosses, on 5,000 points solved by BVP) and training error (transparent lines) of five random seeds are shown in logarithmic scale. For comparison, the validation error of the sparse grid method is shown, as reported by [NGK19b].

For a better comparison of the three methods, we performed a second experiment with a similar setup where the number of additionally requested points in each active learning iteration is fixed to a factor of 1.75. Starting from 64 initial training points, this leads to five network trainings on 64, 112, 196, 343 and 600 labeled instances respectively over the period of four active learning iterations.

The results are presented in table 6.6 and figure 6.14. As the schedule is now fixed, all approaches terminate with 600 points. Compared to the first experiment, this is a bit more for the random method and almost one third less for the GPA. As a result, while the validation error for the GPA Sampler increases a bit, the validation error for the random approach decreases by a disproportional amount (points: 577 $\rightarrow$ 600 $= -4\%$, error: 0.0034 $\rightarrow$ 0.0029 $= -15\%$). The approach from the paper performs even worse with fewer training instances. We also note that the reduction in validation error from random sampling to the GPA Sampler is insignificant while also leading to a slower convergence of the network training. However, at least the points chosen by the GPA Sampler lead to a more realistic training error.

sample mode	train error	validation error	training iterations	number points total
GPA	0.0027	0.0028	46190	600
random	0.0017	0.0029	40123	600
paper	0.0019	0.0043	41682	600

Table 6.6: Final statistics in figure 6.14, average over five random seeds.

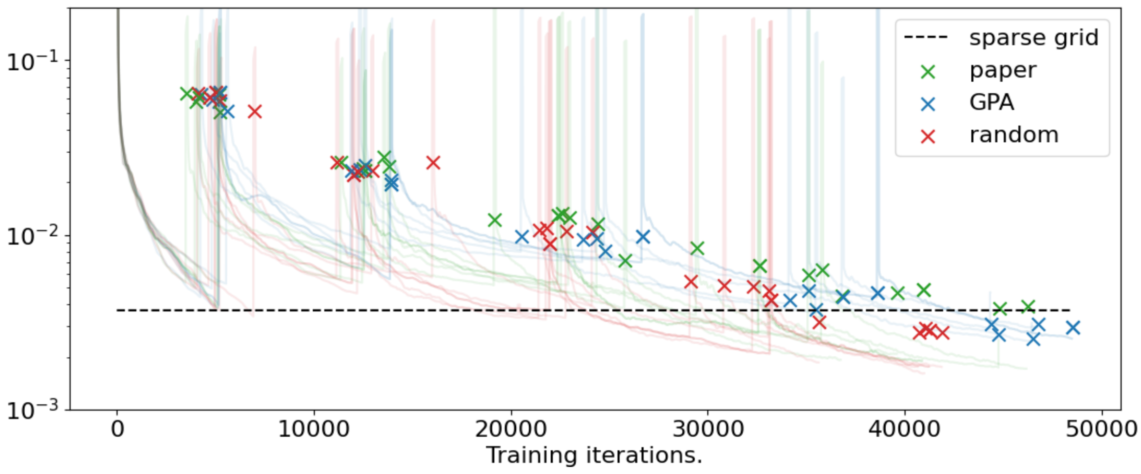

Figure 6.14: As above: validation error (crosses) and training error (lines) of the three approaches over five random seeds. This time, four active learning iterations and the number of selected points in each iteration is fixed to a schedule.

6.6.3 Further Analysis

We tried to improve the performance by replacing the random samples in the state space by quasi-Monte Carlo (QMC) sampled points. When selecting points randomly in the active learning iteration, this means that all points directly stem from the QMC method. For the approach from [NGK19b] and the GPA Sampler, the points *to choose from* are derived from a QMC method. We used the code published by [KN16] which is available at Magic Point Shop (accessed October 19, 2020). Points generated by a lattice and by a Sobol sequences were tested and a random shift was applied to them to compare across different random seeds.

As the results do not differ significantly from the previous experiments, we report them in the appendix (A.2) and intend now to investigate the sampling behavior of the three different approaches. To this end, we reran the second experiment and tracked various parameters. In table 6.7, average statistics for the sampled points are shown (the statistics *per iteration* are similar). First, we can observe that the approach from the paper indeed selects the points with the highest gradient norm yet the points selected by the GPA Sampler also have a clearly higher gradient norm than the randomly selected points.

sample mode	gradient norm	norm V1	norm V2	norm V3
GPA	10.4	0.29	0.30	0.30
random	7.8	0.21	0.19	0.19
paper	12.1	0.23	0.25	0.36

Table 6.7: Statistics from re-running the second experiment. Average over five random seeds and the 536 points which were sampled in total. Gradient norm at the time of sampling and the squared values of the three velocity states are shown.

Taking a closer look at the velocities in table 6.7, we notice that points chosen by GPA Sampler and the paper approach have higher velocity compared to the random sampling. For the paper approach, the third velocity (V3) is particularly high, especially in comparison with the other velocities. This motivated the visualization given in figure 6.15. First, consider the plots without the colors. In the columns, we compare the three sampling methods, in the rows random points with the lattice and Sobol sequences. The two-dimensional embedding shows the three location states of the sampled points. The points are relatively well distributed in space, with exception of the lattice sequence which directly is directly selected in the first column.

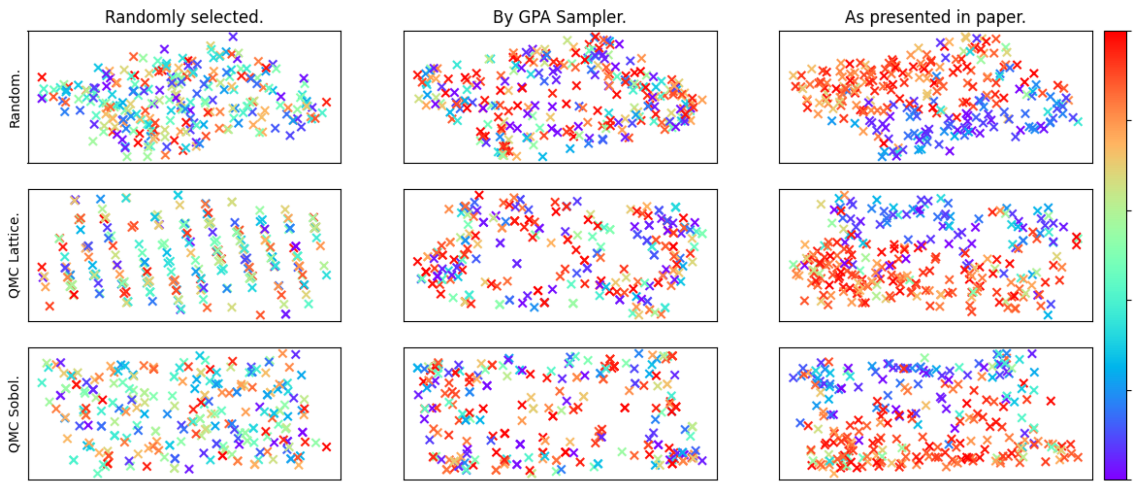

Figure 6.15: 343 points selected in the last active learning iteration of the second experiment. Different samplers and (quasi-)Monte Carlo random points are compared. The three location states are embedded with a principal component analysis on the points of a row. The color indicates the relative value of the third velocity state (from red = highest positive value, to blue = highest negative value).

Colors indicate the value of the third velocity of the selected points. They are normalized in each row, accordingly the red points show points with a (close to) highest positive velocity from the initial points and the blue crosses are (close to) a highest negative velocity. In general, the GPA Sampler tends to choose points with a higher absolute velocity compared to random sampling whose points range over the whole spectrum. In the third column, a distinct behavior is observed: By selecting points with the highest gradients, only those instances are selected which have a certain correlation between location states (location of the cross in plot) and third velocity (color of the plot). More precisely, on one side of the location space, only points with high *positive* velocity are selected while the points on the other side have a high *negative* velocity. This indicates a lack of diversity which we suspect to have caused the lower training successes with the method proposed in [NGK19b].

Chapter 7

Conclusion

In this thesis we have seen how Bayesian neural network can be utilized for active learning and how sampling methods can be tailored towards these models. After we established a general active learning framework and discussed the training process of Bayesian neural networks, a first sampling policy based on the sample variance of the network was presented. To improve the diversity of selected samples, we followed the work of Tsymbalov et al. [Tsy+19]. To establish an active learning strategy, the connection between random neural networks and Gaussian processes was utilized to approximate a posterior variance together with a fast updating procedure. We extended the proposed updating procedure to batches of points and implemented the proposed framework to contribute various empirical results which show - among other things - the runtime savings of the improved updating procedure. For future work the following open challenges could be taken into consideration:

Intrabatch Diversity
Although the proposed sampling strategy based on the Gaussian process approximation allows for an update which - in theory - ensures diversity, in section 6.3 we saw that the recalculation of the complete covariance matrix is still very time consuming when performed once per iteration. Sampling multiple points (batches) per iteration - especially with the proposed fast updating procedure - decreases the runtime significantly but sacrifices diversity *within* the batch. It would be interesting to see if a fast routine (faster than the updating procedure) which reestablishes the diversity could improve the sampling quality without increasing the runtime again.

Dealing with Large Datasets:
In the pool-based active learning framework (2.2) considered throughout this thesis, we typically want to calculate the acquisition function for all points in the pool to decide which point to select. When introducing the GPA Sampler, we discussed and improved its runtime but did not mention the memory complexity. For storing the covariances of the pool X_{pool} a matrix of size $|X_{pool}| \times |X_{pool}|$ is needed. Quickly, this can become a problem as matrix arithmetric requires even more memory. For example, already 10,000 pool points result in a covariance matrix which requires 0.8 GB digital storage (with 64-bit floats). For large datasets it is therefore necessary to develop strategies circumventing such problems. It could be interesting to represent the pool data by a representative set or to consider not all the inter-point covariances but only those above a certain threshold.

Multi-dimensional Predictions

The developed GPA Sampler relies on the posterior variance of a single output value. The connection between Gaussian processes and random neural networks is not restricted to the case of one dimensional predictions and one could imagine that the GPA Sampler could also be generalized to multiple dimensional output. An easy way of doing that would be to estimate uncertainties dimension-wise. Yet, this does not take the inter-dimensional dependencies into account which is why proper modeling could be superior.

Scheduling of Active Learning Iterations

In most of the experiments we conducted in chapter 6, we increased the number of training instances in each active learning iteration by a fixed amount. Although the approach presented in section 6.6 did not seem to improve over a fixed schedule, this line of research should be continued. For example: when employing the GPA Sampler, a decrease of the maximum posterior variance over the period of sampling multiple points can be observed. Therefore, a good number of selected points could be reached when the decrease gets to a certain percentage.

Bibliography

[Agg+14] Charu C. Aggarwal et al. “Active learning: A survey”. English (US). In: *Data Classification*. CRC Press, Jan. 2014, pp. 571–605. DOI: 10.1201/b17320.

[Agh+19] Hamed H. Aghdam et al. *Active Learning for Deep Detection Neural Networks*. 2019.

[Ang88] Dana Angluin. “Queries and Concept Learning”. In: *Mach. Learn.* 2.4 (Apr. 1988), pp. 319–342. DOI: 10.1023/A:1022821128753. URL: https://doi.org/10.1023/A:1022821128753.

[Ang90] Dana Angluin. “Negative Results for Equivalence Queries”. In: *Mach. Learn.* 5.2 (July 1990), pp. 121–150. DOI: 10.1023/A:1022692615781. URL: https://doi.org/10.1023/A:1022692615781.

[Ang04] Dana Angluin. “Queries revisited”. In: *Theoretical Computer Science* 313.2 (2004). Algorithmic Learning Theory, pp. 175–194. DOI: https://doi.org/10.1016/j.tcs.2003.11.004. URL: http://www.sciencedirect.com/science/article/pii/S030439750300608X.

[AAF18] Dana Angluin, Timos Antonopoulos, and Dana Fisman. “Query learning of derived ω-tree languages in polynomial time”. In: *CoRR* abs/1802.04739 (2018). URL: http://arxiv.org/abs/1802.04739.

[AAF20] Dana Angluin, Timos Antonopoulos, and Dana Fisman. “Strongly Unambiguous Büchi Automata Are Polynomially Predictable With Membership Queries”. In: *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*. Vol. 152. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 8:1–8:17. DOI: 10.4230/LIPIcs.CSL.2020.8. URL: https://drops.dagstuhl.de/opus/volltexte/2020/11651.

[Aro+19] Sanjeev Arora et al. “On Exact Computation with an Infinitely Wide Neural Net”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8141–8150. URL: http://papers.nips.cc/paper/9025-on-exact-computation-with-an-infinitely-wide-neural-net.pdf.

[ACL90] Les E. Atlas, David A. Cohn, and Richard E. Ladner. “Training Connectionist Networks with Queries and Selective Sampling”. In: *Advances in Neural Information Processing Systems 2*. Ed. by D. S. Touretzky. Morgan-Kaufmann, 1990, pp. 566–573. URL: http://papers.nips.cc/paper/261-training-connectionist-networks-with-queries-and-selective-sampling.pdf.

[ABL18] Pranjal Awasthi, Maria Florina Balcan, and Philip M. Long. *The Power of Localization for Efficiently Learning Linear Separators with Noise.* 2018.

[BBL09] Maria-Florina Balcan, Alina Beygelzimer, and John Langford. "Agnostic active learning". In: *Journal of Computer and System Sciences* 75.1 (2009). Learning Theory 2006, pp. 78–89. DOI: https://doi.org/10.1016/j.jcss.2008.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0022000008000652.

[BHV10] Maria-Florina Balcan, Steve Hanneke, and Jennifer Vaughan. "The true sample complexity of active learning". In: *Machine Learning* 80 (Sept. 2010), pp. 111–139. DOI: 10.1007/s10994-010-5174-y.

[BU16] Maria-Florina Balcan and Ruth Urner. "Active Learning – Modern Learning Theory". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 8–13. DOI: 10.1007/978-1-4939-2864-4_769. URL: https://doi.org/10.1007/978-1-4939-2864-4_769.

[BB11] Heinz Bauer and Robert B. Burckel. *Probability Theory*. Berlin, Boston: De Gruyter, 3May. 2011. DOI: https://doi.org/10.1515/9783110814668. URL: https://www.degruyter.com/view/title/3786.

[BT19] Jatin Bedi and Durga Toshniwal. "Deep learning framework to forecast electricity demand". In: *Applied Energy* 238 (2019), pp. 1312–1326. DOI: https://doi.org/10.1016/j.apenergy.2019.01.113. URL: http://www.sciencedirect.com/science/article/pii/S0306261919301217.

[Bel+18] William H. Beluch et al. "The Power of Ensembles for Active Learning in Image Classification". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[Blu+15] Charles Blundell et al. *Weight Uncertainty in Neural Networks*. 2015.

[BBB13] M. Bouguelia, Y. Belaïd, and A. Belaïd. "A Stream-Based Semi-supervised Active Learning Approach for Document Classification". In: *2013 12th International Conference on Document Analysis and Recognition*. 2013, pp. 611–615.

[Byr+95] Richard H. Byrd et al. "A Limited Memory Algorithm for Bound Constrained Optimization". In: *SIAM Journal on Scientific Computing* 16.5 (1995), pp. 1190–1208. DOI: 10.1137/0916069. URL: https://doi.org/10.1137/0916069.

[CJN17] Kamalika Chaudhuri, Prateek Jain, and Nagarajan Natarajan. "Active Heteroscedastic Regression". In: ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, June 2017, pp. 694–702. URL: http://proceedings.mlr.press/v70/chaudhuri17a.html.

[Cha+15] Kamalika Chaudhuri et al. *Convergence Rates of Active Learning for Maximum Likelihood Estimation*. 2015.

[CG20] Chun-Teh Chen and Grace X. Gu. "Generative Deep Neural Networks for Inverse Materials Design Using Backpropagation and Active Learning". In: *Advanced Science* 7.5 (2020), p. 1902607. DOI: 10.1002/advs.201902607. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/advs.201902607.

[CP19] Xue Chen and Eric Price. "Active Regression via Linear-Sample Sparsification". In: ed. by Alina Beygelzimer and Daniel Hsu. Vol. 99. Proceedings of Machine Learning Research. Phoenix, USA: PMLR, 25–28 Jun 2019, pp. 663–695. URL: http://proceedings.mlr.press/v99/chen19a.html.

[CS09] Youngmin Cho and Lawrence K. Saul. "Kernel Methods for Deep Learning". In: *Advances in Neural Information Processing Systems 22*. Ed. by Y. Bengio et al. Curran Associates, Inc., 2009, pp. 342–350. URL: http://papers.nips.cc/paper/3628-kernel-methods-for-deep-learning.pdf.

[CAL94] David Cohn, Les Atlas, and Richard Ladner. "Improving Generalization with Active Learning". In: 15.2 (May 1994), pp. 201–221. DOI: 10.1023/A:1022673506211. URL: https://doi.org/10.1023/A:1022673506211.

[DE95] Ido Dagan and Sean P. Engelson. "Committee-Based Sampling For Training Probabilistic Classifiers". In: *Machine Learning Proceedings 1995*. Ed. by Armand Prieditis and Stuart Russell. San Francisco (CA): Morgan Kaufmann, 1995, pp. 150–157. DOI: https://doi.org/10.1016/B978-1-55860-377-6.50027-X. URL: http://www.sciencedirect.com/science/article/pii/B978155860377650027X.

[Das05] Sanjoy Dasgupta. "Analysis of a greedy active learning strategy". In: *Advances in Neural Information Processing Systems 17*. Ed. by L. K. Saul, Y. Weiss, and L. Bottou. MIT Press, 2005, pp. 337–344. URL: http://papers.nips.cc/paper/2636-analysis-of-a-greedy-active-learning-strategy.pdf.

[DHM08] Sanjoy Dasgupta, Daniel J Hsu, and Claire Monteleoni. "A general agnostic active learning algorithm". In: *Advances in Neural Information Processing Systems 20*. Ed. by J. C. Platt et al. Curran Associates, Inc., 2008, pp. 353–360. URL: http://papers.nips.cc/paper/3325-a-general-agnostic-active-learning-algorithm.pdf.

[DKM05] Sanjoy Dasgupta, Adam Tauman Kalai, and Claire Monteleoni. "Analysis of Perceptron-Based Active Learning". In: *Learning Theory*. Ed. by Peter Auer and Ron Meir. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 249–263.

[Den+19] Cheng Deng et al. "Active Transfer Learning Network: A Unified Deep Joint Spectral–Spatial Feature Learning Model for Hyperspectral Image Classification". In: *IEEE Transactions on Geoscience and Remote Sensing* 57.3 (Mar. 2019), pp. 1741–1754. DOI: 10.1109/tgrs.2018.2868851. URL: http://dx.doi.org/10.1109/TGRS.2018.2868851.

[Dil+17] Joshua V. Dillon et al. *TensorFlow Distributions*. 2017.

[ER90] Bonnie Eisenberg and Ronald L. Rivest. “On the Sample Complexity of Pac-Learning Using Random and Chosen Examples”. In: *Proceedings of the Third Annual Workshop on Computational Learning Theory*. COLT ’90. Rochester, New York, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 154–162.

[EVD08] Arkady Epshteyn, Adam Vogel, and Gerald DeJong. “Active reinforcement learning”. In: Jan. 2008. DOI: 10.1145/1390156.1390194.

[FLC17] Meng Fang, Yuan Li, and Trevor Cohn. “Learning how to Active Learn: A Deep Reinforcement Learning Approach”. In: *CoRR* abs/1708.02383 (2017). URL: http://arxiv.org/abs/1708.02383.

[Fre+97] Yoav Freund et al. “Selective Sampling Using the Query by Committee Algorithm”. In: *Mach. Learn.* 28.2–3 (Sept. 1997), pp. 133–168. DOI: 10.1023/A:1007330508534. URL: https://doi.org/10.1023/A:1007330508534.

[FK16] Kaito Fujii and Hisashi Kashima. “Budgeted stream-based active learning via adaptive submodular maximization”. In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 514–522. URL: http://papers.nips.cc/paper/6038-budgeted-stream-based-active-learning-via-adaptive-submodular-maximization.pdf.

[G M+18] Alexander G. de G. Matthews et al. *Gaussian Process Behaviour in Wide Deep Neural Networks*. 2018.

[Gal16] Yarin Gal. “Uncertainty in Deep Learning”. PhD thesis. University of Cambridge, 2016.

[GG15] Yarin Gal and Zoubin Ghahramani. *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*. 2015.

[GIG17] Yarin Gal, Riashat Islam, and Zoubin Ghahramani. *Deep Bayesian Active Learning with Image Data*. 2017.

[GE17] Yonatan Geifman and Ran El-Yaniv. *Deep Active Learning over the Long Tail*. 2017.

[GAS20] Christopher J. Geoga, Mihai Anitescu, and Michael L. Stein. “Scalable Gaussian Process Computations Using Hierarchical Matrices”. In: *Journal of Computational and Graphical Statistics* 29.2 (2020), pp. 227–237. DOI: 10.1080/10618600.2019.1652616. URL: https://doi.org/10.1080/10618600.2019.1652616.

[GG07] Yuhong Guo and Russ Greiner. “Optimistic Active Learning Using Mutual Information”. In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI’07. Hyderabad, India: Morgan Kaufmann Publishers Inc., 2007, pp. 823–829.

[Haf+20] Danijar Hafner et al. *Noise Contrastive Priors for Functional Uncertainty*. 2018, accessed 28.09.2020. URL: https://github.com/brain-research/ncp.

[Har+20] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[HJ15] Tamir Hazan and Tommi Jaakkola. *Steps Toward Deep Kernel Methods from Infinite Neural Networks*. 2015.

[HFL13] James Hensman, Nicolo Fusi, and Neil D. Lawrence. *Gaussian Processes for Big Data*. 2013.

[Hou+11] Neil Houlsby et al. *Bayesian Active Learning for Classification and Preference Learning*. 2011.

[JGH18] Arthur Jacot, Franck Gabriel, and Clément Hongler. *Neural Tangent Kernel: Convergence and Generalization in Neural Networks*. 2018.

[KW16] Wei Kang and Lucas C. Wilcox. *Mitigating the Curse of Dimensionality: Sparse Grid Characteristics Method for Optimal Feedback Control and HJB Equations*. 2016.

[KB97] R. Kelley Pace and Ronald Barry. “Sparse spatial autoregressions”. In: *Statistics and Probability Letters* 33.3 (1997), pp. 291–297. DOI: https://doi.org/10.1016/S0167-7152(96)00140-X. URL: http://www.sciencedirect.com/science/article/pii/S016771529600140X.

[KS01] Jacek Kierzenka and Lawrence F. Shampine. “A BVP Solver Based on Residual Control and the Maltab PSE”. In: *ACM Trans. Math. Softw.* 27.3 (Sept. 2001), pp. 299–316. DOI: 10.1145/502800.502801. URL: https://doi.org/10.1145/502800.502801.

[KW13] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013.

[KAG19] Andreas Kirsch, Joost van Amersfoort, and Yarin Gal. *BatchBALD: Efficient and Diverse Batch Acquisition for Deep Bayesian Active Learning*. 2019.

[Kri12] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: *University of Toronto* (May 2012).

[KL51] S. Kullback and R. A. Leibler. “On Information and Sufficiency”. In: *Ann. Math. Statist.* 22.1 (Mar. 1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[KG20] Punit Kumar and Atul Gupta. “Active Learning Query Strategies for Classification, Regression, and Clustering: A Survey”. In: *Journal of Computer Science and Technology* 35 (July 2020), pp. 913–945. DOI: 10.1007/s11390-020-9487-4.

[KN16] Frances Y. Kuo and Dirk Nuyens. *Application of quasi-Monte Carlo methods to elliptic PDEs with random diffusion coefficients - a survey of analysis and implementation*. 2016.

[LB92] K. Lang and E. Baum. “Query learning can work poorly when a human oracle is used”. In: 1992.

[LC10] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[Lee+17] Jaehoon Lee et al. *Deep Neural Networks as Gaussian Processes*. 2017.

[LG94] David D. Lewis and William A. Gale. “A Sequential Algorithm for Training Text Classifiers”. In: *CoRR* abs/cmp-lg/9407020 (1994). URL: http://arxiv.org/abs/cmp-lg/9407020.

[MH08] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[Mac92] D. J. C. MacKay. "Information-Based Objective Functions for Active Data Selection". In: *Neural Computation* 4.4 (1992), pp. 590–604.

[Mar+15] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[MRE16] Mitchell McIntire, Daniel Ratner, and Stefano Ermon. "Sparse Gaussian Processes for Bayesian Optimization". In: *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*. UAI'16. Jersey City, New Jersey, USA: AUAI Press, 2016, pp. 517–526.

[Mos+07] Robert Moskovitch et al. "Improving the Detection of Unknown Computer Worms Activity Using Active Learning". In: Sept. 2007, pp. 489–493. DOI: 10.1007/978-3-540-74565-5_47.

[NGK19a] Tenavi Nakamura-Zimmerer, Qi Gong, and Wei Kang. *Adaptive Deep Learning for High-Dimensional Hamilton-Jacobi-Bellman Equations*. 2019.

[NGK19b] Tenavi Nakamura-Zimmerer, Qi Gong, and Wei Kang. *Adaptive Deep Learning for High-Dimensional Hamilton-Jacobi-Bellman Equations*. 2019. URL: https://github.com/Tenavi/HJB_NN.

[Nea96] R. Neal. "Priors for Infinite Networks". In: 1996.

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PF18] Remus Pop and Patric Fulop. *Deep Ensemble Bayesian Active Learning : Addressing the Mode Collapse issue in Monte Carlo dropout via Ensembles*. 2018.

[Ram+14] Raghunathan Ramakrishnan et al. "Quantum chemistry structures and properties of 134 kilo molecules". In: *Scientific Data* 1 (Aug. 2014). DOI: 10.1038/sdata.2014.22.

[RW06] CE. Rasmussen and CKI. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, Jan. 2006, p. 248.

[RKG18] Matthias Rottmann, Karsten Kahl, and Hanno Gottschalk. "Deep Bayesian Active Semi-Supervised Learning". In: *CoRR* abs/1803.01216 (2018). URL: http://arxiv.org/abs/1803.01216.

[RM01] Nicholas Roy and Andrew McCallum. “Toward Optimal Active Learning through Sampling Estimation of Error Reduction”. In: *Proceedings of the Eighteenth International Conference on Machine Learning*. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 441–448.

[Rud+12] Lars Ruddigkeit et al. “Enumeration of 166 Billion Organic Small Molecules in the Chemical Universe Database GDB-17”. In: *Journal of Chemical Information and Modeling* 52.11 (2012). PMID: 23088335, pp. 2864–2875. DOI: 10.1021/ci300415d. URL: https://doi.org/10.1021/ci300415d.

[Rus+14] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *CoRR* abs/1409.0575 (2014). URL: http://arxiv.org/abs/1409.0575.

[Sch+18] K. T. Schütt et al. “SchNetPack: A Deep Learning Toolbox For Atomistic Systems”. In: *Journal of Chemical Theory and Computation* 15.1 (Nov. 2018), pp. 448–455. DOI: 10.1021/acs.jctc.8b00908. URL: http://dx.doi.org/10.1021/acs.jctc.8b00908.

[Sch+17] Kristof T. Schütt et al. *SchNet: A continuous-filter convolutional neural network for modeling quantum interactions*. 2017.

[SS17] Ozan Sener and Silvio Savarese. *Active Learning for Convolutional Neural Networks: A Core-Set Approach*. 2017.

[Set09] Burr Settles. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison, 2009.

[SC08] 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. Honolulu, Hawaii: Association for Computational Linguistics, 2008, pp. 1070–1079.

[SCR08] Burr Settles, Mark Craven, and Soumya Ray. “Multiple-Instance Active Learning”. In: *Advances in Neural Information Processing Systems 20*. Ed. by J. C. Platt et al. Curran Associates, Inc., 2008, pp. 1289–1296. URL: http://papers.nips.cc/paper/3252-multiple-instance-active-learning.pdf.

[SOS92] H. S. Seung, M. Opper, and H. Sompolinsky. “Query by Committee”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT ’92. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1992, pp. 287–294. DOI: 10.1145/130385.130417. URL: https://doi.org/10.1145/130385.130417.

[Sha48] C. E. Shannon. “A Mathematical Theory of Communication”. In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x.

[Sma+14] Jasmina Smailović et al. “Stream-based active learning for sentiment analysis in the financial domain”. In: *Information Sciences* 285 (2014). Processing and Mining Complex Data Streams, pp. 181–203. DOI: https://doi.org/10.1016/j.ins.2014.04.034. URL: http://www.sciencedirect.com/science/article/pii/S0020025514004885.

[Sou+17] Jamshid Sourati et al. “Asymptotic Analysis of Objectives Based on Fisher Information in Active Learning”. In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 1123–1163.

[Sri+14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[Sze+16] Christian Szegedy et al. *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*. 2016.

[TH19] Ying-Peng Tang and Sheng-Jun Huang. “Self-Paced Active Learning: Query the Right Thing at the Right Time”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (July 2019), pp. 5117–5124. DOI: 10.1609/aaai.v33i01.33015117.

[TK02] Simon Tong and Daphne Koller. “Support Vector Machine Active Learning with Applications to Text Classification”. In: *J. Mach. Learn. Res.* 2 (Mar. 2002), pp. 45–66. DOI: 10.1162/153244302760185243. URL: https://doi.org/10.1162/153244302760185243.

[Tra+19] Dustin Tran et al. *Bayesian Layers: A Module for Neural Network Uncertainty*. 2019.

[TPS18] Evgenii Tsymbalov, Maxim Panov, and Alexander Shapeev. “Dropout-Based Active Learning for Regression”. In: *Analysis of Images, Social Networks and Texts* (2018), pp. 247–258. DOI: 10.1007/978-3-030-11027-7_24. URL: http://dx.doi.org/10.1007/978-3-030-11027-7_24.

[Tsy+19] Evgenii Tsymbalov et al. “Deeper Connections between Neural Networks and Gaussian Processes Speed-up Active Learning”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence* (Aug. 2019). DOI: 10.24963/ijcai.2019/499. URL: http://dx.doi.org/10.24963/ijcai.2019/499.

[Val84] L. G. Valiant. “A Theory of the Learnable”. In: *Commun. ACM* 27.11 (Nov. 1984), pp. 1134–1142. DOI: 10.1145/1968.1972. URL: https://doi.org/10.1145/1968.1972.

[VD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[VC71] V. N. Vapnik and A. Ya. Chervonenkis. “On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities”. In: *Theory of Probability & Its Applications* 16.2 (1971), pp. 264–280. DOI: 10.1137/1116025. URL: https://doi.org/10.1137/1116025.

[Wan+19] Guijin Wang et al. "A global and updatable ECG beat classification system based on recurrent neural networks and active learning". In: *Information Sciences* 501 (2019), pp. 523–542. DOI: https://doi.org/10.1016/j.ins.2018.06.062. URL: http://www.sciencedirect.com/science/article/pii/S0020025518305115.

[WCC19] Sarah Wassermann, Thibaut Cuvelier, and Pedro Casas. "RAL - Improving Stream-Based Active Learning by Reinforcement Learning". In: *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD) Workshop on Interactive Adaptive Learning (IAL)*. Würzburg, Germany, Sept. 2019. URL: https://hal.archives-ouvertes.fr/hal-02265426.

[WNC06] Rebecca Willett, Robert Nowak, and Rui M. Castro. "Faster Rates in Regression via Active Learning". In: *Advances in Neural Information Processing Systems 18*. Ed. by Y. Weiss, B. Schölkopf, and J. C. Platt. MIT Press, 2006, pp. 179–186. URL: http://papers.nips.cc/paper/2831-faster-rates-in-regression-via-active-learning.pdf.

[WF17] Mark Woodward and Chelsea Finn. *Active One-shot Learning*. 2017.

[Wu+19] Yuexin Wu et al. *Active Learning for Graph Neural Networks via Node Feature Propagation*. 2019.

[YK19] Donggeun Yoo and In So Kweon. *Learning Loss for Active Learning*. 2019.

[Zhd19] Fedor Zhdanov. *Diverse mini-batch Active Learning*. 2019.

Appendix A

Appendix

A.1 Runtimes on the Airline Dataset

The figures in this section report the results from performing the experiments from section 6.3.2 on the airline dataset. They are very similar to figures 6.2, 6.3 and 6.4 and therefore do not reveal any deeper insights.

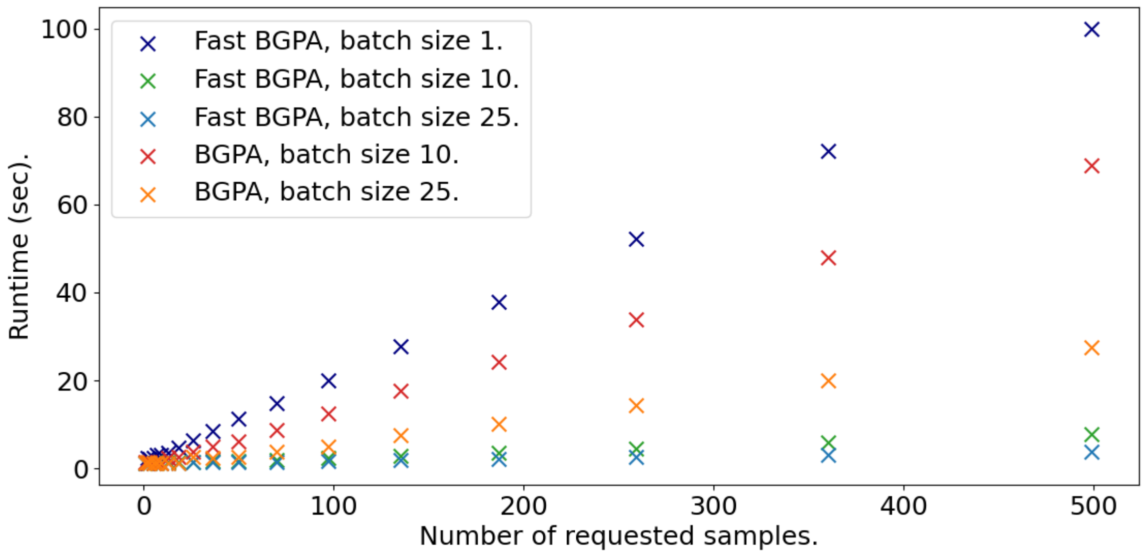

Figure A.1: Runtime of Batch GPA and Fast Batch Samplers on the airline data by number of requested samples, compare figure 6.2.

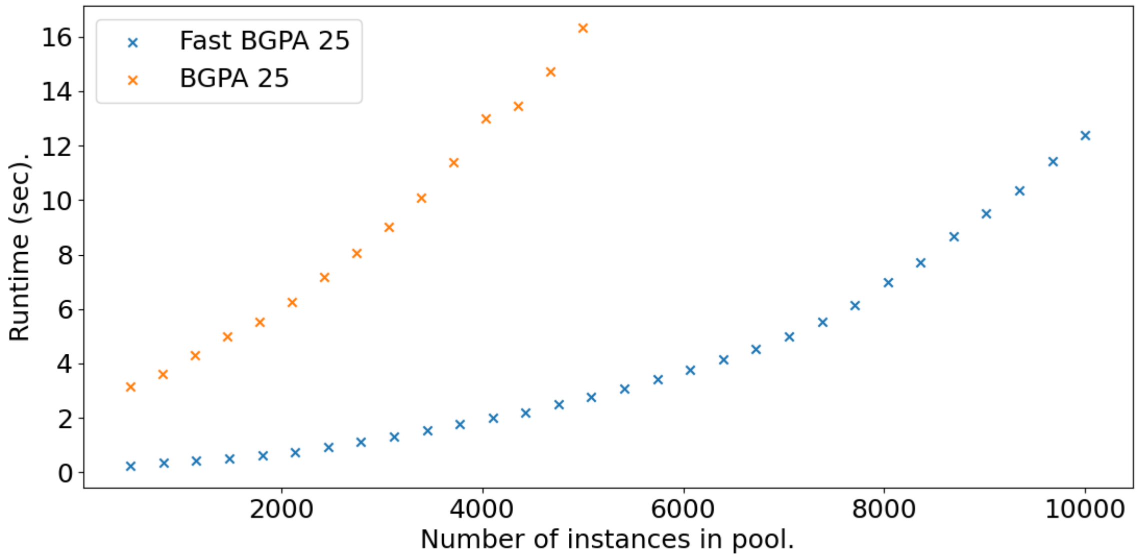

Figure A.2: Runtime of Fast Batch GPA Sampler and Batch GPA Sampler for different numbers of pool instances on the airline data, compare figure 6.3.

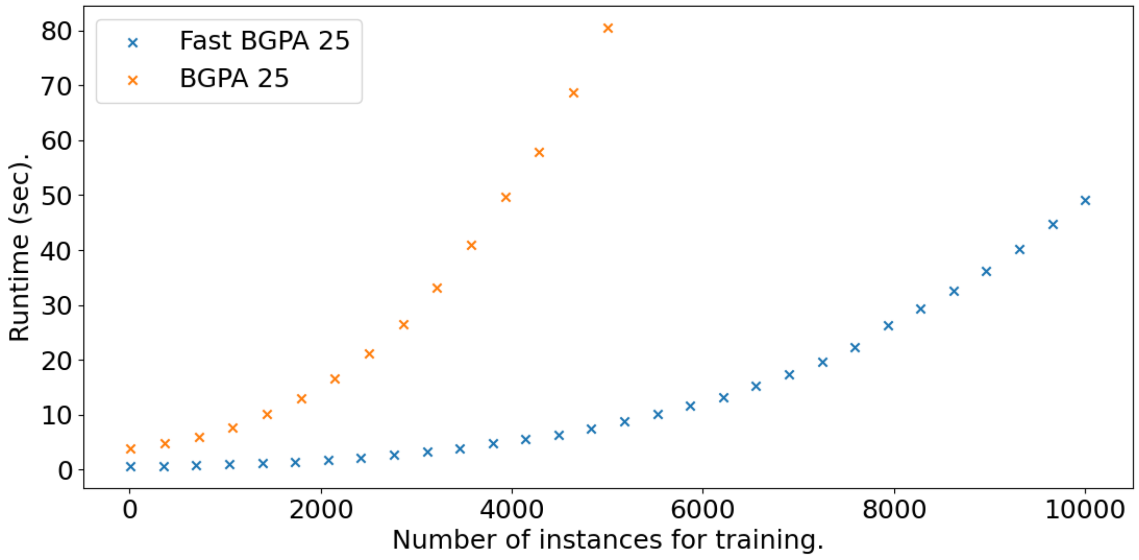

Figure A.3: Runtime for Fast Batch GPA Sampler and Batch GPA Sampler for different numbers of training instances on the airline data, compare figure 6.4.

A.2 Results for HJB Equations with QMC Points

The following tables report the results from using the lattice and Sobol sequence Quasi-Monte-Carlo implementation from [KN16] in the setup from section 6.6. Tables A.1 and A.2 contain the final statistics when the number of sampled points in each active learning iteration is chosen at runtime as proposed in the paper. Tables A.3 and A.4 show the final results when the number of added points is fixed by a schedule. Compared to tables 6.5 and 6.6 they do not show a significant improvement.

sample mode	train error	validation error	training iterations	number points total
GPA	0.0028	0.0031	52860.5	759.0
random	0.0018	0.0028	50841.0	901.5
paper	0.0013	0.0027	54681.5	1007.0

Table A.1: Final statistics with quasi random points from a lattice sequences and number of samples points chosen at runtime.

sample mode	train error	validation error	training iterations	number points total
GPA	0.004	0.0038	51467.0	618.5
random	0.0019	0.0026	51600.5	897.5
paper	0.0018	0.0038	52912.0	807.5

Table A.2: Final statistics with quasi random points from a Sobol sequences and number of samples points chosen at runtime.

sample mode	train error	validation error	training iterations	number points total
GPA	0.0038	0.0038	44318.0	600.0
random	0.002	0.0037	38679.5	600.0
paper	0.0016	0.0041	46844.5	600.0

Table A.3: Final statistics with quasi random points from a lattice sequences and number of samples points per iteration fixed.

sample mode	train error	validation error	training iterations	number points total
GPA	0.0036	0.0037	42424.5	600.0
random	0.0021	0.0034	36949.5	600.0
paper	0.002	0.0038	42527.5	600.0

Table A.4: Final statistics with quasi random points from a lattice sequences and number of samples points per iteration fixed.