Batch Active Learning with Deep Kernel Gaussian Process Classifiers

Candidate Number: 1032626

Computer Science

University of Oxford

Word count: 8620

Trinity 2021

Abstract

In this project report, we investigate Batch Active Learning Methods and how to take advantage of the improvements that have been made in scalable and good performing Deep Kernel Gaussian Process Classifiers to improve the usefulness of these methods.

The contribution of this project is in 2 parts.

- We modify DUE to enable it work well on small amounts of data, to be able to be used in the context of active learning, and get the active learning methods to work with this type of model and verify the performance in an active learning setting. Here we equal the results of the original BatchBALD paper. - We reduce the computational complexity of computing the required values for the investigated active learning methods to make this practical. We improve the cost of each acquisition from being $O(n^3)$, related to the pool size if implemented directly to $O(n)$.

Keywords— Active Learning - Gaussian Processes - Deep Kernel Learning

Table of Contents

List of Figures

List of Algorithms

List of Theorems

List of Abbreviations

GP . . . . . Gaussian Process

GPC . . . . . Gaussian Process Classification

BALD . . . . . Bayesian Active Learning by Disagreement

ELBO . . . . . Evidence Lower Bound

ML . . . . . Machine Learning

MNIST . . . . . Modified National Institute of Standards and Technology (A standard handwritten digit dataset)

BALD . . . . . Bayesian Active Learning by Disagreement

vDUQ . . . . . Variational Deterministic Uncertainty Quantification

DUE . . . . . Deterministic Uncertainty Estimation

CUDA . . . . . Compute Unified Device Architecture

VRAM . . . . . Video Random Access Memory

CIFAR10 . . . . . A standard academic image classification dataset

1 | Introduction

Active learning is an area of machine learning where we are attempting to select which unlabelled datapoints we believe would be best to learn the label of. It is important because in many use cases (for example, medical data), it is often either quite costly or just simply impossible to collect an incredibly large dataset. However current approaches to do it suffer from issues related to quality of acquired datapoints, model requirements and computational issues.

Bayesian ML models are the standard ways of building machine learning models which can calculate uncertainty well, however many of the standard approaches to doing so (MC Dropout BNNs) can give suboptimal performance, for example requiring a substantial amount of sampling. Gaussian Processes are a ML approach which give very good uncertainty performance, and new results in the field [DUE] have shown successful training of Variational Gaussian Processes with feature extractors and demonstrate that they maintain good uncertainty performance.

1.1 Aim and Objectives

The aims of this report is to

- Give an outline of active learning and the the various active learning methods we will be investigating.
- Extend these methods to enable them to work well with DK-GPCs
- Reduce the computation required to perform these calculations to make it of practical use.
- Perform experiments to validate the performance of the methods.

1.2 Project Report Outline

The remainder of this report is organized as follows:

Chapter 2 — Defines active learning, and introduces the different active learning methods which we will be using.

Chapter 3 — Bayesian Machine Learning Models

Chapter 4 — introduces DUE

Chapter 5 — GPC specific considerations

Chapter 6 — Implementation details and computational complexity.

Chapter 7 — Experiments and Results

2 | Active Learning Methods

In a common active learning formulation we have the following setup:

- X_{pool} is the distribution of the pool of points.
- X_{true} is the true real world distribution.

A common (and sensible) assumption is to assume that these distributions are the same.

Our machine learning models, which we use to model a problem constrain the set of possible functions which we can represent and learn. This is how we imbue the problem with our prior beliefs about the nature of the problem.

- There are hard constraints (eg. clipping the output of the model), which even given an unlimited amount of data our model can not possibly "learn around".
- There are softer constraints (eg. priors over weights in a layer), which our model can learn correctly given sufficient amounts of data even if our prior is poor*.

This parameterization of our models is very important, as this is an assumption we are making about our problem.

An important observation is that the dataset we are using for training is, unless we are performing very simple active learning approaches (eg. random acquisition) is not going to be the same as the true dataset which we are working with.

This violates a very common assumption that we assume for convergence in many ML methods. This statistical bias is looked at in the following paper (Farquhar, Gal, & Rainforth, 2021).

When working with active learning, we are normally training our models on much smaller datasets than is standard in machine learning. Given our models are often over parameterized to begin with (Zhang, Bengio, Hardt, Recht, & Vinyals, 2017), the smaller dataset will make this even more so, this can lead to training issues.

In the next sections we will describe the active learning methods which we will be investigating in this report.

2.1 Random

The most straightforward acquisition function is to randomly sample from the poolset. This is a surprisingly strong baseline for for many datasets. However random acquisition can have some failure modes in comparison to some other methods.

An example of this failure can occur when only a small subset of the poolset contain useful new information. Datasets such as this can exist when the poolset is unbalanced, relative to the test set. The number of datapoints of a particular class which will be selected from the poolset will approximately in proportion.

2.2 BALD

Bayesian Active Learning by Disagreement (BALD) (Houlsby, HuszÃ¡r, Ghahramani, & Lengyel, 2011) is an active learning method which is designed to select the datapoint which we expect would reduce the uncertainty of the posterior by the greatest amount if the value of y at that point was known.

$$\arg\max_x H\left[\theta|D\right] - \mathbf{E}_{y\sim p(y|x,D)}\left[H\left[\theta|y,x,D\right]\right]$$

To try and compute this directly we would have to calculate the entropy in the parameter space, however this is can be a very high dimensional space. However, the authors of this paper observe that this objective can be rewritten as the conditional mutual information of the output variable and the parameters.

$$I\left(y,\theta|D,x\right) = H\left[\theta|D\right] - \mathbf{E}_{y\sim p(y|x,D)}\left[H\left[\theta|y,x,D\right]\right]$$

By taking advantage of this formulation they then rewrite the objective as follows, which requires the computation of the entropy over the output variable instead which is normally over a much smaller dimensional space.

$$\arg\max_x H\left[y|x, D\right] - \mathbf{E}_{\theta\sim p(\theta|D)}\left[H\left[y|x,\theta\right]\right]$$

Using BALD to select multiple datapoints to acquire at the same time presents an issue. Selecting the n points with the greatest individual information content will not necessarily give you the n points which jointly have the greatest information content. This is transparent when considering the case when there is duplicates in the dataset.

2.3 BatchBALD

BatchBALD (Kirsch, van Amersfoort, & Gal, 2019), is an extension of BALD designed especially to deal with the above issue, which we run into when using BALD to acquire batches of points.

To do this the objective function is modified, and the original BALD objective is retained in the case when we set $n = 1$ in the new BatchBALD objective.

The BatchBALD score function is as follows.

Definition 2.3.1 (BatchBALD)**.**

$$s_{BatchBald}(x_1, \ldots, x_n) = H(y_1, \ldots, y_n) - E_{p(f)}\left[H(y_1, \ldots, y_n|f)\right]$$

Definition 2.3.2 (Submodularity)**.** A function $f : P(X) \to \mathbf{R}$ is sub modular if for all $A, B \subseteq X$ we have the following.

$$f(A) + f(B) \geq f\left(A \cup B\right) + f\left(A \cap B\right)$$

The authors of the paper prove that the objective function they specify is *sub-modular*, which enables a greedy $1 - \frac{1}{\epsilon}$ approximation algorithm for selecting the optimum batch.

As the variables are independent, conditioned on the value of the function, the right hand side of the equation factors into a sum over conditional expectations for each individual datapoint. The computationally difficult part of this objective is the computation of the

joint entropy as this does not factor similarly. In fact, the summation required to compute this value is exponential in the number of datapoints we are currently considering. Estimating the entropy of a discrete distribution is something which we will investigate later here.

The authors of the paper address this issue by using sampling to estimate the joint entropy for larger batch sizes. The BatchBALD paper deals with Bayesian Neural Networks (BNNs) in their formulation of their problem and in their solution. They are then able to sample from the weight distribution once to obtain a function draw, which is defined for all possible inputs of the BNN. This is not an ability we have when using Gaussian Process Classifiers, which we will come back to later.

2.4 Joint Entropy

We can also simply use the joint entropy alone as the active learning objective

$$s_{Entropy}(x_1, \ldots, x_n) = H(y_1, \ldots, y_n)$$

Proof 2.4.1 ($s_{Entropy}$ is submodular)**.** Proof. Let $A_x, B_x \subseteq D_{pool}, C_x = A_x \cap B_x$

$\hat{A} = A_y \setminus C_y, \hat{B} = B_y \setminus C_y$

$$\begin{aligned}
s_{Entropy}(A_x) + s_{Entropy}(B_x) &= H(A_y) + H(B_y) \\
&= H(C_y \cup \hat{A}) + H(C_y \cup \hat{B}) \\
&= H(\hat{A}|C_y) + H(\hat{B}|C_y) + 2H(C_y) \\
&\geq H(\hat{A}, \hat{B}|C_y) + 2H(C_y) \\
&= H(\hat{A}, \hat{B}, C_y) + H(C_y) \\
&= H(A_y \cup B_y) + H(A_x \cap B_x) \\
&= s_{Entropy}(A_x \cup B_x) + s_{Entropy}(A_x \cap B_x)
\end{aligned}$$

The fact that this objective is sub-modular enables us to use the same greedy $1 - \frac{1}{\epsilon}$ approximation algorithm as used above.

A justification for the use of this objective can be seen as attempting to minimize the entropy over the remaining items in the pool given the batch which we select.

$$\min_B H((P \setminus B)\,|B) = \min_B H(P) - H(B) = \min_B -H(B) = \max_B H(B)$$

If we assume that the samples in the poolset are samples from the true distribution, we can view this as attempting to minimize the entropy of the output variables maximally fast. This is in contrast with the BALD objective which attempts to minimize the entropy of the hypothesis maximally fast.

2.4.1 Distinction between Joint Entropy and Predictive Entropy

The predictive entropy of a machine learning classification model for a particular input, is the entropy of probabilities which it has predicted.

This represents how unsure the model is about this particular datapoint. They key point here is that it is possible for our model to be sure that it is unsure, this is the behavior which we would want in the case that the output truly is ambiguous. In this case we would not gain much by selecting a known ambiguous datapoint to learn from.

It is important for us to make this distinction clear for clarity of explanation. This expected value of this quantity is the second term in the BatchBALD objective.

3 | Bayesian Machine Learning Models

Here we will detail some Bayesian Machine Learning models, in particular focusing on 2 types.

- Gaussian Processes
- Bayesian Neural Networks

3.1 Introduction

In machine learning we are often attempting to optimise the values of some function relative to some objective. This is often done by attempting to find a single set of parameters which give the best performance. We can interpret this as attempting to find the parameters which best match a dataset, which can be represented by maximizing a *likelihood* function, a function which represents how good of a fit to the data this set of parameters are. This likelihood function is of the form $p(x|\theta)$.

This is called **Maximum Likelihood Estimation**.

Definition 3.1.1 (Maximum Likelihood Estimation)**.** Given a likelihood function $l(x|\theta)$

$$MLE_\theta(x) = \arg\max_\theta l\left(x|\theta\right)$$

For example Ordinary Least Squares regression will give you the Maximum Likelihood Estimate of a linear regression model.

This approach can be extended if we have some belief/information about what the parameters of the model are.

Definition 3.1.2 (Bayes Theorem)**.** Let A, B be events.

$$P\left(A|B\right) = \frac{P\left(B|A\right)P\left(A\right)}{P\left(B\right)}$$

Define the random variables θ, which represent the distribution of our parameters, and X

which represents our data.

$$P(\theta|X) = \frac{P(X|\theta)\,P(\theta)}{P(X)}$$

Given all of these quantities we could update our beliefs about our parameters in a principled fashion.

Definition 3.1.3 (Maximum a posteriori)**.** Given a likelihood function $l(x|\theta)$

$$MAP_\theta(x) = \arg\max_\theta \frac{p(x|\theta)\,p(\theta)}{p(x)}$$

As $p(x)$ is going to be the same for all of the values of θ for a fixed x we can ignore it in our maximization. We can view the MAP as selecting the value of θ which maximizes the likelihood of x, weighted by our prior belief about how likely that parameter was.

Given this new extended method, the MLE can now be reframed as a special case of MAP, when there is uniform prior on the values of the parameters.

By selecting a single value from the parameter distribution we are discarding information. When using bayesian models, we instead of selecting a value from this distribution which maximizes some objective, we endeavour to maintain and use the full distribution.

3.2 Bayesian Neural Networks

A Bayesian Neural Networks is a Bayesian Machine Learning model which are structured much like typical neural networks. However instead of having a single output function specified by values of the parameters of the network, we maintain a distribution over the parameters of the model. This distribution and the structure of the BNN together induce a distribution over functions. Much like in other bayesian methods, we are required to have a prior belief for the values of the parameters (weights). Investigations for good priors for weights are investigated in (Fortuin et al., 2021).

To update the value of the weights or perform inference, we need to calculate the values of the corresponding integrals. Unfortunately many of the integrals we need to perform

for bayesian inference are intractable, which requires us to approximate these quantities to use these methods.

A very simple and computationally efficient way of performing approximate bayesian inference is by using MC Dropout (Gal & Ghahramani, 2016). Dropout is a simple method where while training we randomly, with a probability p set certain values in the network to 0. This is combined with certain scaling to maintain the size and scale of the outputs.

Dropout was original designed as a regularization method before this was discovered. The key difference between using MC Dropout and simply using dropout as a regularization method is that with MC Dropout we also use dropout at inference time.

While neural networks can as they get larger better and better approximate arbitrary functions (), for a BNN of any finite size there is a limit to its expressivity.

The next section we will look into Gaussian Processes, which are another type of bayesian model that can approximate arbitrarily () complex functions.

3.3 Gaussian Processes

Definition 3.3.1 (Gaussian Process)**.** A Gaussian processes, is a collection of random variables, where any finite subset of them have a joint Gaussian distribution. (Rasmussen, 2003)

We can completely specify a Gaussian Process over a space χ by giving a mean function $m(\cdot), m : \chi \to R$, and covariance (kernel) function $k(\cdot, \cdot), k : \chi \times \chi \to R$.

The output distribution for any subset of points in χ can be easily found by evaluating the mean and covariance functions, and constructing a Gaussian Distribution with these values.

3.3.1 Benefits of Gaussian Processes

When we are using a GP for regression, we are immediately constraining ourselves to having Gaussian distributed outputs. In other bayesian models, we often use normal distributions as our priors, or as our noise model (). This restriction to working with Gaussian

Distributions is less restrictive than it may first appear.

We are trading off the ability for arbitrary output random variables in our model in exchange for our model being **conjugate** () given gaussian distributed inputs. We are also able to compute the posterior exactly and efficiently, unlike in the general bayesian case.

Another key reason is that Gaussian Processes can model arbitrarily complex functions given enough data paired with a corresponding kernel function (the space of functions which a GP can learn is completely parameterized by the kernel).

A Gaussian Process is the limit of an infinite width feedforward neural network when the weights of the network are initialized from a Gaussian distribution. (Williams, 1996)

3.3.2 Kernel Functions

For $k : \chi \times \chi \to R$ to be a valid covariance (kernel) function we need for any evaluation of a covariance matrix defined by the kernel, to be a symmetric positive definite matrix (as all covariance matrices of random variables must be S.P.D).

Definition 3.3.2 (Positive Definite Matrix)**.** A positive definite matrix is a matrix which has all positive eigenvalues.

The kernel function of a GP defines the space of functions over which it is going to be defined.

3.3.3 Gaussian Distributions

Gaussian Distributions have some very useful properties which make them very useful for this particular case.

3.3.3.1 Marginalisation

We have defined our Gaussian Process over a space, and require that any finite subset of the values to have a Gaussian Distribution. For this to be consistent it is important that if we generated the GP output for n points and then marginalize over one of them, we should get the same result as if we generated the distribution over the remaining $n - 1$ directly.

Theorem 3.3.1 (Marginalisation of Gaussian Distributions)**.**

$$\begin{bmatrix} X \\ Y \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} \Sigma_{XX} & \Sigma_{XY} \\ \Sigma_{YX} & \Sigma_{YY} \end{bmatrix}\right) \rightarrow X \sim N\left(\mu_X, \Sigma_{XX}\right)$$

This means that we don't need to compute the marginals when we wish to use them, we can instead drop the relevant indices from the mean and covariance.

3.3.3.2 Conditioning

When we want to condition one Gaussian given another Gaussian distribution we have the following result.

Theorem 3.3.2 (Conditioning a Gaussian)**.**

$$\begin{bmatrix} X \\ Y \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} \Sigma_{XX} & \Sigma_{XY} \\ \Sigma_{YX} & \Sigma_{YY} \end{bmatrix}\right) \rightarrow$$
$$X|Y = y \sim N\left(\mu_X + \Sigma_{XY}\Sigma_{YY}^{-1}\left(y - \mu_y\right), \Sigma_{XX} - \Sigma_{XY}\Sigma_{YY}^{-1}\Sigma_{YX}\right)$$

3.3.4 Update Equations for Gaussian Processes

We can use the equations above for Gaussian Distributions and extend them to our Gaussian Processes.

The posterior of our Gaussian Process given data is defined similarly to the conditional distribution above, however as do not know yet what points we wish to evaluate the GP on the mean, and covariances are left in terms of the mean and covariance function which we have specified.

This can be done as follows:

Theorem 3.3.3 (Posterior of a Gaussian Process)**.**

$$GP\left(m\left(X\right), k\left(X, X\right)\right)|D_Y = d \sim$$
$$GP\left(m(X) + K(X,Y)K(Y,Y)^{-1}\left(d - mu(Y)\right), K(X,X) - K(X,Y)K(Y,Y)^{-1}K(Y,X)\right)$$

3.3.5 Space of functions which can be represented

While the technical details of the exact space of functions from which our GP is a distribution over is beyond the scope of this report. We do have the following if we have an infinitely differentiable kernel (C_∞) eg the RBF kernel, the functions we learn will also be C_∞. The functions drawn from a Gaussian Process inherit their smoothness properties from the kernel.

3.4 Training

Using a straightforward Gaussian Process with no hyper-parameters, for regression there is nothing to train. We simply have the exact equations to compute the posterior distribution at any possible input. In contrast to many optimization problems our output for any particular value is not a point estimate, it is a distribution so our objective should take this into account.

3.4.1 KL Divergence

The KL Divergence is a measure of how different 2 probability distributions are, (it is not symmetric so it is not strictly a distance functions). It has many useful properties and unlike shannon entropy it has a sensible and natural extension to continuous random variables.

Definition 3.4.1 (KL Divergence)**.** For 2 probability distributions P, Q defined on the same probability space χ

$$D_{KL}\left(P||Q\right) = E_{x\sim P(x)} \log\left(\frac{P(x)}{Q(x)}\right)$$

It is often quite difficult to compute the **KL Divergence**, as often we don't have direct access to one of the distributions which we are attempting to calculate it with.

The Evidence Lower Bound is a tractable objective which it is much easier for us to compute than the KL divergence and maximizing this objective is equivalent to maximizing

the KL Divergence.

Definition 3.4.2 (ELBO)**.**

$$L(X) = H(Q) - H(Q; P(X, Z))$$

Maximisng the ELBO will minimise the KL Divergence. We can write the KL diverge between our the true distribution our approximate distribution Q and the true distribution $P(Z|X)$ in terms of the ELBO

$$D_{KL}(Q||P(Z|X)) = \log P(X) - L(X)$$

As the $\log P(X)$ term is not impacted by modification of Q, we can ignore it for the purposes of optimization. To minimize the left hand side w.r.t to Q, we will need to minimize $-L(X)$. Which is equivalent to maximizing $L(X)$. □

3.4.2 Scaling to large data

In contrast with other machine learning methods such as

3.4.2.1 Inducing Points

In inducing point GPs, we instead of using all of the datapoints in our model. We instead use a smaller number of datapoints as an approximation to the true model. Various approaches exist for selecting the points we keep in our approximation.

One of the most successful approaches is to treat these inducing points as model parameters which we attempt to learn by minimizing some objective.

Minimizing the evidence lower bound (ELBO) is a common and principled object for this task.

Minimizing this objective minimizes the KL Divergence between the true output distribution and the approximate distribution we are learning, leading to us learning the closest conjugate distribution to the true distribution.

3.4.2.2 Random Fourier Features

In this approximation approach, we approximate the kernel function using finite basis function expansion. (Rahimi & Recht, 2008).

3.5 Gaussian Process Classification

To use Gaussian Processes for classification we have several issues to overcome.

The output of a Gaussian Process is a Multivariate Normal, this is a continuous distribution over the real space, however in the classification setting we wish to have a categorical output. Which means we need to have a distribution over the the simplex.

We can in theory use any link function to convert a sample from our MVN in R^n to a sample over the simplex, different link functions have different properties computationally and statistically.

3.5.1 Link Functions

The link function between the latent function (our GP) and the output of our classifier is again another parameter of the model.

3.5.1.1 Logit

The logit is the standard link function for the vast majority of GPC use cases. The use of the logit implicitly makes our GPC model obey the independence of irrelevant alternatives axiom from decision theory (He, Zhang, Ren, & Sun, 2015).

$$p\left(y=k|f\right)=\frac{e^{f^{T}w_{k}}}{\sum_{i=1}^{K}e^{f^{T}w_{i}}}$$

The marginal distribution using this link function does not have a closed form.

3.5.1.2 Probit

The probit is a different link function which is normally defined for the binary classification case as the inverse of the CDF of a Gaussian. We can generalize this link function as follows.

$$p\left(y = k|f\right) = \mathbf{1}_{argmax_i(f_i)=k}$$

The probit function is easier to work with analytically, and great use of this is made in the binary classification case (Houlsby et al., 2011). However it is clear from the definitions that a single sample draw from a GPC using a probit will not give an accurate assessment of the confidence of the model.

If we use probit as our link function, we can take advantage of the fact that it is possible to exactly compute the marginal to get exact results.

If we can compute $p(y = k)$, we can also compute the joint entropy exactly (unlike in the case with an arbitrary link function where we are generally restricted to using MC integration), however the complexity of this exact computation is very high.

To perform exact inference of a Probit GPC, we must perform integration of a Gaussian R.V. This is a well studied problem in the literature. (Hayter & Lin, 2012).

We wish to find the probability that a certain element of a MVN is the largest element. This corresponds to integrating the Gaussian over the subset of the space where this coordinate is the largest.

$$P(X_C > X_1 \wedge \ldots \wedge X_C > X_n) = \int_{x_c \in \{-\infty,\infty\}} \ldots \int_{x_n \in \{-\infty,x_c\}} p(x)dx$$

This can be interpreted as integrating on one side of several hyperplanes.

These planes are of the form $x_c = x_i$

We can perform a linear transformation on this space to project it to a subspace of dimension $n - 1$, and align each of the hyperplanes with an axis.

$$Z = TX.$$

$$Z \sim N(T\mu, T\Sigma T^T)$$

If we are doing this over multiple datapoints, we can transform each of the subsets of variables via the method above.

After performing this transformation we have a standard orthant integral which we can compute. Algorithms exist for computing this exactly in $O(p^2 2^p)$ using recursive integration and subspace projection. (Nomura, 2014)

However the number of dimensions which we are performing this integration over is $O(dc)$. With the exponential complexity in the number of dimensions this is an infeasible method.

4 | vDUQ / DUE

vDUQ / DUE (van Amersfoort, Smith, Teh, & Gal, 2020) is a recent model which deals with some of the issues with using deep kernel learning with GPCs. DUE uses an inducing point GP to enable it to scale to large amounts of data.

This model is designed to have good uncertainty properties by enforcing sensitivity and bounding the amount by which any 2 inputs can diverge in the feature space. These properties are maintained by the addition of residual connections to the model and by performing spectral normalization to bound the lipchitz constant of the layers in the feature extractor.

4.1 Spectral Normalization

The lipchitz constant is a measure of how far apart 2 outputs to the function can be in terms of their inputs.

Definition 4.1.1 (Lipshitz Constant)**.** Given a function $f : X \to Y$, where (X, d_x) and (Y, d_y) are metric spaces, a lipchitz constant of the function f is a value of k such that.

$$\frac{d_y(f(x_1), f(x_2))}{d_x(x_1, x_2)} \leq K, \forall x_1, x_2 \in X$$

Some times we refer to *the lipchitz constant* of a function, this is the smallest value k which is a lipchitz constant.

In our particular problem we are dealing with spaces in $\mathbf{R}^n$. The metric which we will be defining on this space is the L_2 norm, (also known as the euclidean distance).

In the case where $f : R^n \to R^m$ and f is a linear transformation we get a familiar result.

$$\min_k \forall x_1, x_2 \in X \frac{d_y(f(x_1), f(x_2))}{d_x(x_1, x_2)} \leq K = \min_k \forall x_1, x_2 \in X \frac{L_2(Ax_1 - Ax_2)}{L_2(x_1, x_2)} \leq K$$
$$= \min_k \forall x_1, x_2 \in X \frac{||A(x_1 - x_2)||_2}{||x_1 - x_2||_2} \leq K$$
$$= \min_k \forall x \in X \frac{||Ax||_2}{||x||_2} \leq K$$

After the substitution we obtain an expression which is the matrix norm of A induced by the vector nor, in this case the L_2 norm. The matrix norm induced by the L_2 vector norm has the property that it is the *largest singular value* of the matrix which is also the square root of the dominant eigenvalue.

For all of the linear layers in our model we now know how we can compute the respective lipchitz constants.

Theorem 4.1.1 (Lipschitz constant of function compositions)**.** If $lip(f) = a$ and $lip(g) = b$, then $lip(f \circ g) \leq a * b$

The majority of activation functions which we use in our neural networks have a lipchitz constant of ≤ 1, eg. relu (Scaman & Virmaux, 2019). This enables us to compute an upper bound of the lipchitz constant of our model by computing the lipchitz constant of all of the individual layers and taking the product of those constants.

4.1.1 Power Method

Directly computing the largest singular value via computing the Singular Value Decomposition of each linear transform is quite computationally prohibitive. However using the power method we can approximate this value quite cheaply and accurately.

The **power method** is defined by a recurrence relation.

Definition 4.1.2 (Power Method(for square matrices))**.**

$$v_{k+1} = \frac{Av_k}{||Av_k||}$$

This recurrence relation will converge if 2 conditions are met.

- A has a strictly largest eigenvalue
- v_0 has a non zero component in the direction of the largest eigenvalue

We compute the eigenvalue from the rayleigh quotient from our approximate maximum eigenvector. $\frac{v_k^T A v_k}{v_k^T v_k}$

If we chose a starting vector v_0 with uniform probability over all possible directions the second condition will be fulfilled with a probability of 1.

The convergence ratio of this algorithm is $\frac{|\lambda_1|}{|\lambda_2|}$, so the convergence rate depends on the size of the second largest eigenvalue also.

To extend this to work with non square matrices we can either compute $A^T A$ or AA^T, use the power method to calculate its maximum eigenvalue and this is the spectral norm of A. There is another approach which enables us to bypass this computation by instead of maintaining just 1 eigenvector, we maintain a left and a right dominant eigenvector.

Definition 4.1.3 (Power Method(for non square matrices))**.**

$$v_{k+1} = \frac{A^T u_k}{||A^T u_k||}, u_{k+1} = \frac{A v_k}{||A v_k||}$$

Then with these approximate eigenvectors we can approximate the singular value with $u^T A v$

To bound the lipchitz constant of a linear layer to a values, call this value σ. We can calculate the spectral norm α, and check if $\alpha > \sigma$. If this is true, we multiply the linear layer by $\frac{\sigma}{\alpha}$.

To maintain this during training we are required to compute this value during every forward pass. Given a sufficiently small learning rate we expect that the change to linear layer will be small. If we use the previous forwards passes final vector(s) as our starting vector instead of randomly restarting we can greatly speed up the convergence of the recurrence relation. We find that empirically we can actually get very good performance with a single iteration of this method at each stage.

4.2 Residual Connections

Residual connections are a type of Neural Network structure, where the input of one layer can be passed directly to the input of the next layer. The intention of these connections in deep learning is to alleviate the vanishing/exploding gradient problem (He et al., 2015).

$$H(x) = F(x) + x$$

4.3 Modifications

While DUE is an extremely powerful model, some modifications were required to help the performance and stability of the model when working with small amounts of data.

4.3.1 Natural Gradient Descent

Natural Gradient Descent is an optimization method which is designed to optimise some objective of a distribution. The use of this method greatly improved the rate of convergence and overall performance. (*Natural Gradient Descent*, n.d.)

Definition 4.3.1 (Gradient Descent)**.** Gradient Descent is an iterative optimization algorithm where we modify the parameter in the direction which has the steepest gradient with respect to our objective function.

$$p_{k+1} = p_k - \alpha \nabla L(p_k)$$

With a well chosen hyperparameter α and a convex loss function L we can obtain convergence guarantees (Nesterov, 2014). Many machine learning problems which we use gradient descent for do not obey the constraints which we require for guarantees of convergence [], empirically however the performance achieved by gradient descent and its derivatives is impressive even these problems in many cases.

Instead of applying the gradient descent algorithm to the parameters in the euclidean geometry of the parameter space, instead we can use the geometry of the likelihood space

parameterized by our chosen parameters to enable superior optimization. This makes the optimization independent of the parameterization of the distribution and only related to the distribution induced by the the parameters.

To derive the update equations we will need to define some objectives and take some taylor expansions.

The first thing we will define is out loss function. q is the true distribution which we are attempting to approximate, p_θ is our parameterized distribution.

$$L(\theta) = D_{KL}(p_\theta || q)$$

We wish to choose a descent direction which will minimize our loss function, much like in standard gradient descent. Normally in gradient descent we bound the magnitude of the step of the gradient by the euclidean length of the descent vector, we change the constraint in natural gradient descent to instead bound the KL divergence between p_θ and $p_{\theta+\epsilon}$.

$$\epsilon' = \underset{\epsilon \, s.t \, D_{KL}(p_\theta || p_{\theta+\epsilon}) \leq k}{\arg\min} L(\theta + \epsilon) \tag{4.1}$$

$$\tag{4.2}$$

We can take the first order taylor expansion of $L(\theta + \epsilon)$ around θ.

$$\begin{aligned} L(\theta + \epsilon) &\approx L(\theta) + J_L(\theta)((\theta + \epsilon) - \theta) \\ &\approx L(\theta) + J_L(\theta)\epsilon \end{aligned}$$

Given the constraints above we have we can rewrite our objective using the method of lagrange multipliers as below.

$$\epsilon' = \underset{\epsilon \, s.t \, D_{KL}(p_\theta||p_{\theta+\epsilon})\leq k}{\arg\min} L\left(\theta+\epsilon\right) + \alpha\left(D_{KL}\left(p_\theta||p_{\theta+\epsilon}\right) - k\right) \tag{4.3}$$

$$\approx \underset{\epsilon \, s.t \, D_{KL}(p_\theta||p_{\theta+\epsilon})\leq k}{\arg\min} L\left(\theta\right) + J_L\left(\theta\right)\epsilon + \frac{1}{2}\left(\epsilon\right)^T H_L\left(\theta\right)\left(\epsilon\right) + \alpha\left(D_{KL}\left(p_\theta||p_{\theta+\epsilon}\right) - k\right) \tag{4.4}$$

$$\tag{4.5}$$

Now we can apply a taylor expansion to $D_{KL}\left(p_\theta||p_{\theta+\epsilon}\right)$ around θ. $\theta' = \theta + \epsilon$

$$D_{KL}\left(p_\theta||p_{\theta+\epsilon}\right) \approx D_{KL}\left(p_\theta||p_\theta\right) + \left(J_{D_{KL}(p_\theta||p_{\theta'})}\left(\theta\right)\right)\epsilon + \frac{1}{2}\epsilon^T\left(H_{D_{KL}(p_\theta||p_{\theta'})}\left(\theta\right)\right)\epsilon \tag{4.6}$$

The KL Divergence between a distribution and itself is 0 so the first term disappears.

Lets now deal with the second term. The KL Divergence is a positive function, and we know that as we mentioned above it is 0 when the 2 distributions are equal. Then second term is the Jacobian of the KL Divergence between 2 parameterized distributions, *evaluated when the parameters are equal*. So we are taking the Jacobian of a function evaluated at a global minimum, which implies that $J_{D_{KL}(p_\theta||p_{\theta'})}\left(\theta\right) = 0$. This makes the second term disappear.

$$D_{KL}\left(p_\theta||p_{\theta+\epsilon}\right) \approx \frac{1}{2}\epsilon^T\left(H_{D_{KL}(p_\theta||p_{\theta'})}\left(\theta\right)\right)\epsilon \tag{4.7}$$

We can expand out the matrix in the remaining quadratic term to obtain that it is equal to the negative fisher information matrix.

$$H_{D_{KL}(p_\theta||p_{\theta'})}(\theta)_{ij} = J\left(J\left(E_{x\sim p_\theta(x)}\log\left(\frac{p_\theta(x)}{p_{\theta'}(x)}\right)\right)^T\right)_{ij}(\theta) \tag{4.8}$$

$$= \frac{\partial}{\partial\theta'_i}\frac{\partial}{\partial\theta'_j}E_{x\sim p_\theta(x)}\log\left(\frac{p_\theta(x)}{p_{\theta'}(x)}\right)(\theta) \tag{4.9}$$

$$= E_{x\sim p_\theta(x)}\frac{\partial}{\partial\theta'_i}\frac{\partial}{\partial\theta'_j}\log\left(\frac{p_\theta(x)}{p_{\theta'}(x)}\right)(\theta) \tag{4.10}$$

$$= -E_{x\sim p_\theta(x)}\frac{\partial}{\partial\theta'_i}\frac{\partial}{\partial\theta'_j}\log\left(p_\theta(x)\right) \tag{4.11}$$

$$= F \tag{4.12}$$

$$\tag{4.13}$$

Where F is the definition of the Fisher information matrix.

Definition 4.3.2 (Fisher Information Matrix)**.**

$$\mathcal{I}(\theta)_{ij} = -E\left[\frac{\partial^2}{\partial\theta_j\partial\theta_i}\log p(x)\,|\theta\right]$$

Returning to our objective from earlier.

$$\epsilon' \approx \underset{\epsilon\, s.t\, D_{KL}(p_\theta||p_{\theta+\epsilon})\leq k}{\arg\min} L(\theta) + J_L(\theta)\,\epsilon + \alpha\left(\frac{1}{2}\epsilon^T F\epsilon - k\right) \tag{4.14}$$

$$\tag{4.15}$$

To find the best value of ϵ, we can take the derivate of the objective with respect to ϵ at set it to 0.

$$\frac{\partial}{\partial \epsilon}\left(L\left(\theta\right)+J_L\left(\theta\right)\epsilon+\alpha\left(\frac{1}{2}\epsilon^T F\epsilon-k\right)\right)=\frac{\partial}{\partial \epsilon}\left(J_L\left(\theta\right)\epsilon+\frac{\alpha}{2}\epsilon^T F\epsilon\right)=0 \tag{4.16}$$

$$0=J_L\left(\theta\right)^T+\frac{\alpha}{2}\left(F^T+F\right)\epsilon \tag{4.17}$$

$$\tag{4.18}$$

Assuming some smoothness conditions we get that the order of the partial differentiation doesn't impact the Hessian, and in turn the fisher information matrix. $F=F^T$

$$0=J_L\left(\theta\right)^T+\alpha F\epsilon \tag{4.19}$$

$$-J_L\left(\theta\right)^T=\alpha F\epsilon \tag{4.20}$$

$$\epsilon=-\frac{1}{\alpha}F^{-1}J_L\left(\theta\right)^T \tag{4.21}$$

We have now arrived at the update equations which we can use as our optimization method.

Definition 4.3.3 (Natural Gradient Descent)**.** Natural Gradient Descent is similar to gradient descent expect in place of the the gradient we use the natural gradient defined as $\nabla'=F^{-1}J_L\left(\theta\right)^T$

4.3.2 Inducing Points

In (van Amersfoort et al., 2020), it is demonstrated that the excellent uncertainty which DUE can achieve can be reached when using very few inducing points. In the paper they demonstrate using 10 inducing points with CIFAR10. When working with small amounts of data, it was observed that when using 10 inducing points only, that training would often become quite unstable. These issues resolved when increasing the number of inducing points from 10 to 20.

5 | GPC specific considerations

5.1 Sampling from the function posterior

The active learning score functions which we have specified earlier in this report, if implemented directly would require us to be able to take function samples from the underlying Gaussian Process.

In comparison with the approach performed in the BatchBALD paper where sampling is done over the weights of the Bayesian Neural Network, when we have a Gaussian Process as our output.

Taking a function sample from a BNN can be done by sampling the weights, these weights along with the structure of the BNN define a function over the entire space where the BNN itself is defined.

Samples from a Gaussian Process are not taken in a similar fashion, they are taken as samples from the MVN defined over a set of **particular** inputs. Without knowing all of the points where you wish to evaluate the GP you can't create a function sample, (in general).

If we wished to take a function draw over our poolset. This would have $O(n^3)$ complexity, which given our poolset can be quite large, is unacceptably costly.

We can however instead of attempting to take a sample over all possible inputs, we can incrementally create function samples. This can be done by repeatedly creating conditional distributions, sampling from that conditional distribution and using this sample to create a new distribution to repeat the process. We exploit this fact later on in this report for computational speed ups.

These issues mean that we can not easily use many of the same computational tricks used previously to speed up these calculations, however we can take advantage of other properties of Gaussian Processes to improve computational performance in other ways.

5.2 Calculating Entropy

Definition 5.2.1 (Entropy)**.** For a discrete r.v X the entropy is defined as

$$H(X) = -\sum_{x \in X} p(x) \log p(x)$$

We require the ability to compute the entropy of random variables with a large state space. (When considering the joint entropy of random variables, the size of the state space is exponential in the number of random variables).

When we are considering larger and larger pool sizes the number of possible states of our random variable grows at an exponential rate.

With C classes the entropy summation will have C^n terms in it.

5.2.1 Gaussian Process Classifier

As we are using a Gaussian Process Classifier, the random variable of which we are trying to estimate the value of is of a known form.

$$\mathbf{f} \sim N(\mu, \Sigma)$$

$$\mathbf{p} = \sigma(f)$$

$$X_i \sim Cat(p_i)$$

The distribution $p(x)$ does not have a closed formula (in general). We can estimate $p(x)$ via estimating the integral via monte carlo integration.

As we have that $X_i \perp\!\!\!\perp X_j | f$, for a given likelihood sample we can store the exponential number of terms in a factored form. We for a sample f, we can compute $p(x_i = c|f)$ for each of the datapoints. To reconstruct the joint distribution for this function sample we take the product of the terms and relevant categories.

$$p(X) = p(x_1, \ldots, x_n) = \mathbb{E}_{l \sim p(l)}\, p(x_1, \ldots, x_n | l) = \mathbb{E}_{l \sim p(l)} \prod_{i=1}^{n} p(x_i | l)$$

5.2.2 Estimating the entropy

The are 2 different approaches we can use in the estimation of entropy.

- Estimating by taking samples from the distributions
- Estimating given independent conditional variables

5.2.3 Taking samples from the distribution

We can take samples from the distribution, and then use these samples to estimate the entropy.

5.2.3.1 Plugin Estimator

The most straightforward approach is to use the *plugin estimator* (van der Vaart, 2000). This approach is to take k samples from the distribution of interest, calculate the observed probability of each class and compute the entropy.

As we define the term in the entropy summation as 0, when the probability is 0. Using this approach bounds the number of terms in the summation by k.

$$p_j = \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}_{X=j}$$

$$\hat{H}_k = -\sum_{i=1}^{c} \hat{p}_i \log \hat{p}_i$$

Some of the convergence properties of the plugin estimator are below.

$$\mathbb{E}\left(H - \hat{H}_k\right)^2 = O\left(\frac{1}{k}\right)$$

$$\frac{\sqrt{K}}{\sigma}\left(H - \hat{H}_k\right) \sim \mathbf{N}(0,1)$$

5.2.3.2 LP Estimator

Using an *LP Estimator* (Valiant & Valiant, 2011) we can estimate the the entropy with far less samples, when comparing to using a plugin estimator. The required number of samples from the distribution to estimate the entropy well is *sub-linear*, of the order $O\left(\frac{d}{\log d}\right)$, where d is the number of values in the domain which have positive support. However the amount of values with support will be the entire state space, so in our problem this is going to be exponential due to our final layer being a softmax, which means that $d = c^n$.

5.2.4 Estimating given independent conditional variables

As we can actually compute the values $p(x)$ without sampling from the final categorical distribution, and we can take advantage of this fact to improve the performance of our estimator. We can directly obtain the probabilities after the softmax. This enables us to perform the estimation in 2 steps.

- Repeatedly sample from $p(x)$, these samples are the values of x for which we will be using to compute the expectation.
- For this fixed value x, estimate $p(x)$ directly from the function samples which we have computed.

We can also take advantage of the fact that the output variables are conditionally independent given a function sample. This enables us to sample from $p(x)$ without requiring an exponential amount of computation by sampling from the distribution in the following the following sequence of steps.

- Sample from the likelihood distribution $p(f)$
- Put this sample though the softmax
- As the outputs of the different variables are conditionally independent given l, we sample once from the categorical distribution for each variable.

- This is a sample from $p(x)$

6 | Implementation

All of the experiments referenced here can easily be replicated using the codebase for this project which is linked and details in the appendices.

The code for this project is written in Python and makes significant use of the frameworks (Paszke et al., 2019, Pytorch) and (Gardner, Pleiss, Bindel, Weinberger, & Wilson, 2018, GPytorch).

6.1 Computational Complexity

In this project we endeavour to minimize the computationally complexity of the operations which we need to perform to compute the quantities involved to make this project of practical use in real world active learning scenarios.

6.1.1 Sampling from an Multivariate Distribution

During the creation and sampling of the Multivariate distributions we are required to compute several quantities.

Definition 6.1.1 (Cholesky Decomposition)**.** The Cholesky Decomposition of a positive-definite matrix B is a decomposition of the form LL^*, where L is a lower triangular matrix.

This can be computed using a modified version of Gaussian Elimination.

We generate samples from MVNs by generating samples from the standard mvn $Z \sim N(0, 1)$, and then transforming these samples using the mean and the cholesky decom-

position of the covariance matrix.

Algorithm 1: Sampling from a Multivariate Gaussian

`CreateSamples(`μ, Σ`)` **Result:** Sample from $N(\mu, \Sigma)$

$n = len(\mu)$;

$v = array[n]$;

$i = 0$;

while $i < n$ **do**

  $v[i] = mvnsasmple()$;

  $i++$;

end

$L = cholesky(\Sigma)$;

return $\mu + Lv$;

To sample from a distribution of size n, if the cholesky distribution is already computed this requires $O(n^2)$ operations due to the matrix-vector product. If it is not it requires $O(n^3)$. This encourages us to attempt to minimize the number of distinct times where we must compute the cholesky distribution.

6.1.2 Sampling over our candidate batches

Computing samples for each possible candidate batch at each acquisition naively would lead to a complexity at each stage of $O\left(nk(d \times c)^3\right)$ for sampling, where k is the number of samples, c is the number of categories, n is the number of points in the pool and d is the size of the current candidate batch.

If we cache the cholesky distributions we can reduce the complexity to $O(n(d \times c)^3 +$

$nk(d \times c)^2)$.

Algorithm 2: Sampling from all possible batches

`CreatePoolSamples` (μ, Σ) **Result:** K samples from each of the N candiate batches

$v = array[N][K][D][C]$;

$i = 0$;

while $i < N$ **do**

  $j = 0$;

  while $j < K$ **do**

    $v[i][j] = CreateSamples(\mu_i, \Sigma_i)$;

    $j++$;

  end

  $i++$;

end

return v;

This level of complexity is unacceptable, however we can take advantage of the structure of our model to improve the efficiency.

As the Gaussian Process Model we are using is an *independent multitask model*. This means that our covariance matrix is a block diagonal. We can take advantage of this to instead of working with covariance matrices of size $cd \times cd$, we can work with a batch of matrices of size $c \times d \times d$ in our computation.

This reduces the complexity of the sampling to $O(nc(d)^3 + nkc(d)^2)$.

6.1.3 Conditional Distributions

We can reduce the amount of computation required by sharing computation between datapoints.

We can do this as follows first by sampling from the current batch, then sampling from the conditional distribution of each of the datapoints. The size of the current batch will

increase as our acquisitions progress, the size of the conditional distributions will not.

Algorithm 3: Sampling from all possible batches

`CreatePoolSamplesEfficent` $(\mu, \Sigma, \mu_{batch}, \Sigma_{batch})$ **Result:** K samples from each of the N candiate batches

$v = array[N][K][D][C]$;

$samples = array[K][D][C]$;

$a = 0$;

while $a < K$ **do**

  $samples[a] = CreateSamples(\mu_{batch}, \Sigma_{batch})$;

  $a++$;

end

$i = 0$;

while $i < N$ **do**

  $j = 0$;

  while $j < K$ **do**

    $\mu_{conddist}, \Sigma_{conddist} = CreateConditional(\mu_i, \Sigma_i, samples[j])$;

    $v[i][j] = CreateSamples(\mu_{conddist}, \Sigma_{conddist})$;

    $j++$;

  end

  $i++$;

end

return v;

$CreateConditional$ will require a $O(1)$ matrix vector products each with a complexity of $O(cd^2)$, and we will require the matrix inverse of the current batch covariance matrix. This will be the same over all function calls so we don't need to recompute this value.

The complexity of this algorithm is $O\left(cd^3 + kcd^2 + nkcd^2\right)$.

The improvement here is to disconnect the size of the poolset from the cubic d^3 term.

The single cubic term can also be removed by performing a rank-1 update on the inverse of the previous batch. This can be performed by using a block matrix inversion identity.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B\left(D - CA^{-1}B\right)^{-1}CA^{-1} & -A^{-1}B\left(D - CA^{-1}B\right)^{-1} \\ -\left(D - CA^{-1}B\right)^{-1}CA^{-1} & \left(D - CA^{-1}B\right)^{-1} \end{bmatrix}$$

Substituting in our rank 1 format.

$$\begin{bmatrix} A & u \\ u^T & \alpha \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}u\left(\alpha - u^TA^{-1}u\right)^{-1}u^TA^{-1} & -A^{-1}u\left(\alpha - u^TA^{-1}u\right)^{-1} \\ -\left(\alpha - u^TA^{-1}u\right)^{-1}u^TA^{-1} & \left(\alpha - u^TA^{-1}u\right)^{-1} \end{bmatrix}$$

If we compute the upper quadrant of the result in such an order that we compute matrix vector products until we are left with a outer product, this will give us a $O(d^2)$ matrix inverse update.

This method gives us a complexity of $O\left(nkcd^2\right)$ for the entire sampling.

We can once again reduce the amount of computation required to take K samples from each candidate by taking $S = K//M$ samples from the batch and taking M samples from the conditional distributions.

If we do this our complexity will become $O\left(scd^2 + nscd^2 + nsmc\right) = O\left(nscd^2 + nsmc\right)$.

Here we have been analyzing the complexity for a single step acquisition step. The total cost of this for the acquisition of a complete batch will be of the order of $O\left(nscd^3 + dnsmc\right)$.

Performing this naively as drawing a function sample over the entire pool set would give a complexity of $O\left(kd\left(c \times n\right)^3\right)$.

Performing the same analysis without using the additional proposed computational speed ups would give us a complexity of $O\left(nk\left(c \times d\right)^4\right)$

To summarize, our algorithm for computing the required samples does the following:

- Reduces the $O(n^3)$ at each stage and replaces it with $O(nd^3)$
- Reduces the $O(d^3)$ for each acquisition step, to $O(d^2)$.

6.1.4 Creation of GP Outputs required

To be able to compute the samples as we have above, at each stage of the acquisition we require the covariance of each point in the candidate batch and the pool points. To compute this at each batch batch directly we end up requiring $O(ncd^3)$ computation to evaluate the covariance matrices. However there is clearly wasted computation as all of the candidate batches will have an identical sub-matrix of size $d \times d$.

The approach taken in this project is as follows. We at each stage compute the GP output of the most recently acquired datapoint with each of the datapoints in the pool. The complexity of this operation is $O(nc)$, as we are only computing covariances between pairs of points.

We can then cache these computations as the acquisitions progress, we can compute all of the possible rank-1 updates to the current batch in time $O(ncd)$, from these stored values. This once again eliminates the cubic term and replaces it with a linear one.

Both the conditional distributions and the rank 1 updates are lazily generated to reduce the memory usage of the program, in addition to the computations are dynamically chunked to enable them to fit in memory and to be able to execute with limited RAM. This is essential to obtain good performance using CUDA operations on a GPU, as the amount of VRAM available limits the amount of parallel computation one can take advantage of.

7 | Experiments and Results

We have set up some of these experiments to demonstrate the issues with some of the active learning methods detailed in this paper.

The general outline of our hypothesis of some of our experiments are quite similar to (Kirsch et al., 2019).

- In MNIST with unbalanced pool data, (where we have significantly more training data of a particular class compared to the test set), we would expect that the performance of random acquisition would deteriorate.
- In MNIST with repeated datapoints, (where the poolset consists of the repeated datapoints with some noise added), we would expect that BALD would deteriorate.
- In using DUE, compared to a BNN we would hope for superior uncertainty performance which we would hope would enable us to achieve superior active learning performance with our objectives.

An observation which was made when creating the experimental framework which we are using here is that the architecture in the feature extractor is very important to the performance of these models on low amounts of data, and seemingly small changes to the model will impact the performance.

While DUE has no problems in learning MNIST with more expressive models, eg larger and standard resents ResNets (He et al., 2015). When using these the performance on very small datasets (eg small subsets of MNIST), the performance was reduced significantly.

To account for this observed effect, for all of our models we use the feature extractor architecture from (Kirsch et al., 2019), in their MNIST experiments, with minimal modifications. This has the added benefit of making the comparison of our models directly comparable to their experimental framework.

For these experiments, we used an acquisition size of 10 and acquire up 300 points.

7.1 Model Outlines

7.1.1 Base Architecture

This is the BNN from (Kirsch et al., 2019), which we are basing our models on.

- CNN layer 1

  2d CNN layer with a kernel size of 5, and 32 output channels

  2d MC dropout layer

  2D max pooling layer

  relu

- CNN layer 2

  2d CNN layer with a kernel size of 5, and 62 output channels

  2d Mc dropout layer

  2D max pooling layer

  relu

- FC Layer 1

  Fully connected layer

  MC dropout layer

  relu

- Fully connected layer
- softmax

7.1.2 DUE

7.1.2.1 Changes

The changes which we make to this architecture for the DUE feature extractor is below.

- We remove the MC Dropout Layers.
- We add residual connections between the layers to satisfy the constraints on DUE. For layers which do not have exactly matching sizes we follow the approach from (van Amersfoort et al., 2020) and use CNN layers to bridge this discrepancy.
- We apply spectral normalization to all of the layers in the model.
- We remove the final projection layer for the feature space dimension to the output dimension.
- We uee 20 inducing points
- We use an RBF kernel

  With a learnable output scale

  With a learnable length scale
- The GP is an independent multitask GP

We initialize the inducing points using the method from (van Amersfoort et al., 2020).

To train the model we train the parameters using 2 different optimizers simultaneously.

- We train the parameters of the GP using a variational optimizer which takes advantage of the NGD method described earlier to improve the rate of convergence.
- We use ADAM to train the parameters of the feature extractor. This is done instead of using NGD for both as the observed performance of the model did not significantly differ when using NGD vs ADAM on the feature extractor, however the time complexity of using NGD is greater than ADAM so we reverted to ADAM for these parameters.

The learning rate of the variational optimizer was 0.1, and the learning rate of ADAM was 0.003. The spectral normalization coefficient was 9.

7.1.3 BNN

The BNN is the same design as from (Kirsch et al., 2019), as detailed in the base model section. The MC Dropout parameter used is 0.5 in each of the layers.

7.1.4 DNN

The DNN used in these experiments is the same as the feature extractor we use for DUE, with an additional linear layer to project the feature dimension to the number of classes 10.

7.2 Charts

7.2.1 DUE

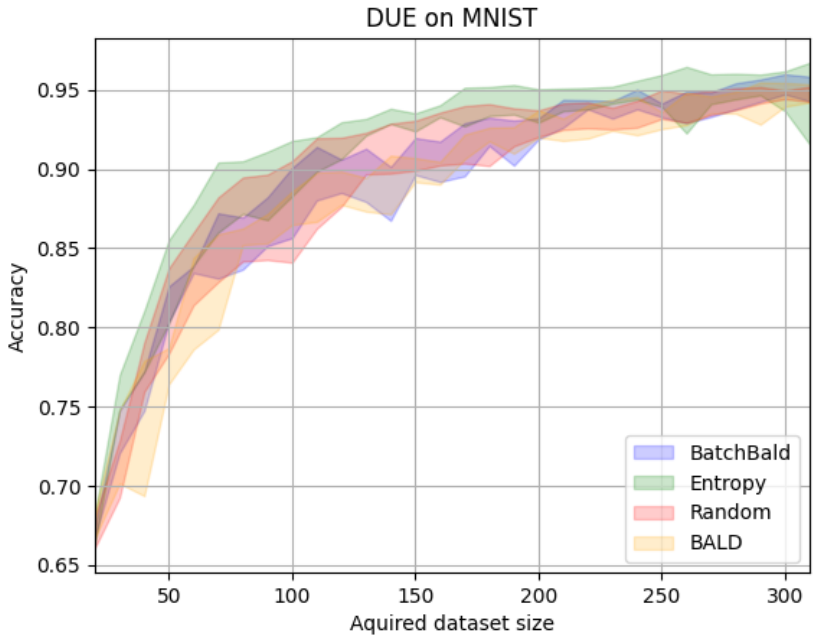


Figure 7.1: Random MNIST

On the standard MNIST dataset, we observe that the entropy objective performs the best. With the random acquisition performing very strongly.

In (Kirsch et al., 2019), there is no direct comparison of the performance of BatchBALD

and random on standard MNIST. Here we perform that comparison, in part to emphasis how strong this baseline is.

To continue the comparison with the original BatchBALD paper, we have collated the results and plotted the 25%, 50% and 75% quartiles.

	90% accuracy	95% accuracy
BALD	120/140/160	300/ >300/ > 300
BatchBALD	100/125/150	270/290/310
Entropy	70/90/110	170/200/250

An interesting take away is that BatchBALD with DUE performs worse than with a BNN. However the joint entropy with DUE performs marginally better, however with seemingly larger variance.

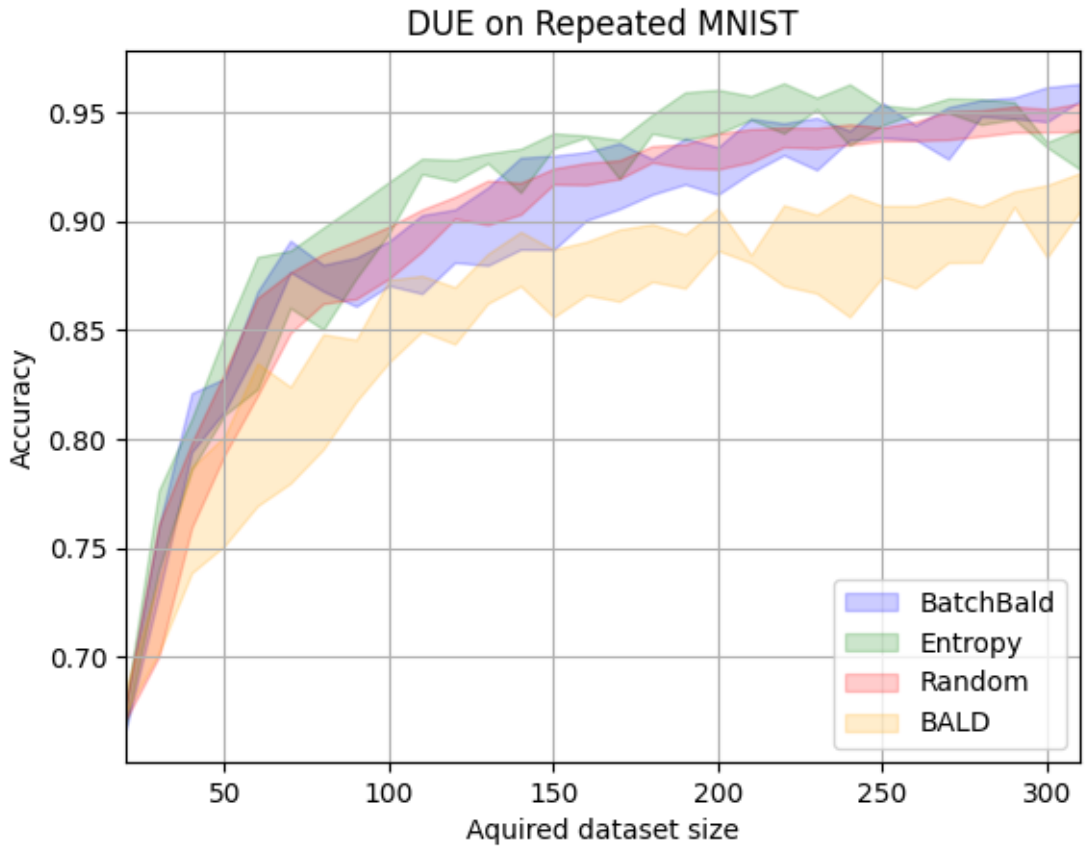


Figure 7.2: Random Repeated MNIST

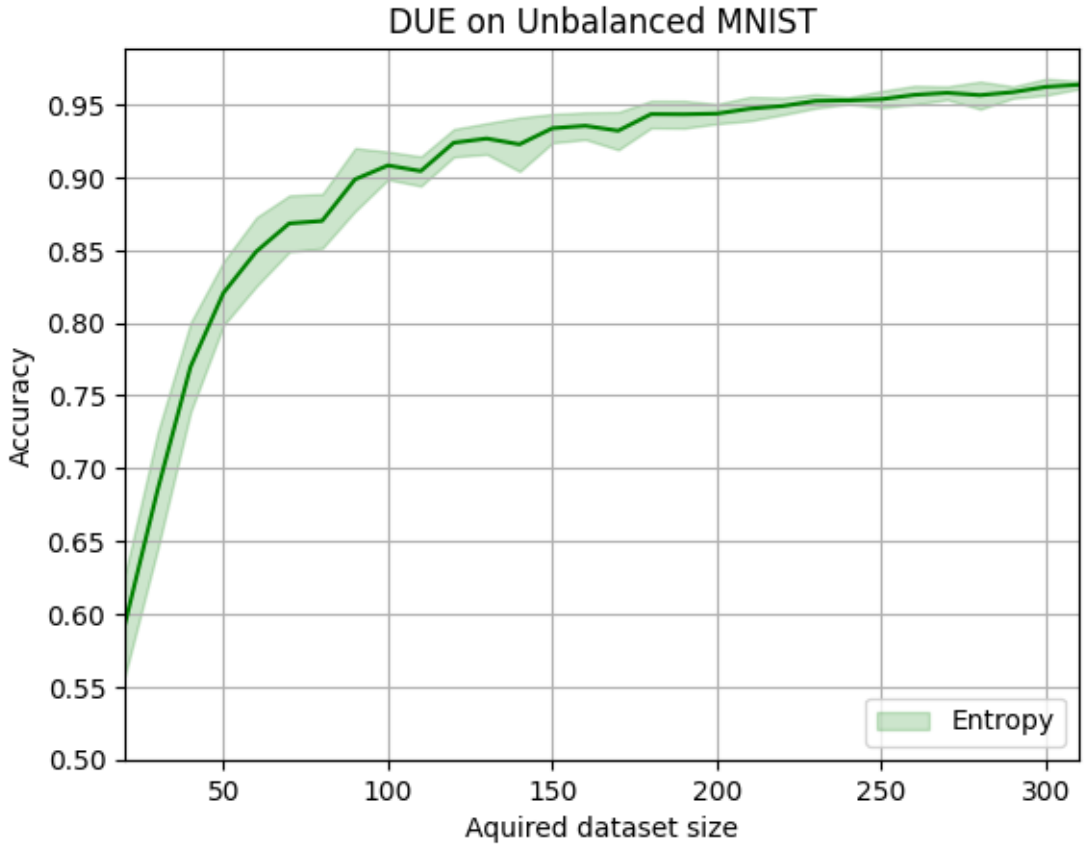


Figure 7.3: Unbalanced MNIST with DUE

7.2.2 BNN

The performance of our BNNs was within a reasonable margin of the results of the original BatchBALD paper. This discrepancy may simply be due to a different starting set, or a slightly different hyperparameter

7.2.3 DNN

We can only use the random acquisition strategy when using a DNN. Positive results from this experiment indicated that the feature extractor was capable of learning in low data regimes.

As the results of these experiments went as expected and there is little to comment on, the charts for them have been placed in the appendices.

7.3 Interpretation of experimental results

The outline of the results from these experiments as as follows.

- The specific failure modes of BALD and Random which we tested for occur when using DUE, much like when using a BNN
- The performance of simply using the joint entropy of the output variables leads to superior performance than the use of the BatchBALD objective when working with DUE. This is not observed when using BNNs.

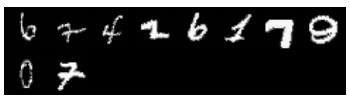
7.3.1 Qualitative results at the acquired batches

See appendices for a wider selection of acquired batches for the various active learning method.

We can view the batches and the sorts of datapoints which they select. As we would expect from the BALD acquisition function we get points which are quite similar all selected in the same batch.

Figure 7.4: BALD selected batch

The randomly selected datapoints are not very useful to observe but are included in the appendices for completeness.



(a) Entropy selected batch

(b) BatchBALD selected batch

We can qualitatively observe that the points which the entropy method selects are diverse, as are the points selected by BatchBALD. BatchBALD selects points which are similar but the model believes could be of distinct classes. The 3 and 8 selected look similar but are of 2 different classes. The points selected by the entropy method are diverse.

7.3.2 Entropy vs BatchBALD for DUE

From our experiments we can observe that using the joint entropy alone performs better than the BatchBALD objective. Here we will explore one reason why this maybe the case.

The difference between the 2 objectives is that with the BatchBALD objective we are subtracting the conditional entropy from the joint entropy term.

7.3.2.1 Conditional Entropy

The second part of the BALD object is the conditional entropy of the output variable conditioned on the function value. The output length scale of our model is a learned parameter of our model. When our GPs on the feature space reverts to the prior, the conditional entropy will become a function of the output scale of the GP.

All of the covariance matrices will be scaled by this output length scale, this is clear by the definition of what the output scale is. We are defining our kernel as $\sigma k\left(x, x^{*}\right)$, where k is itself a kernel function. All of the covariances will be scaled by σ, *even far away from the data we have seen*.

We can view that the output scale monotonically increases during training, until we stop training with early stopping.

To demonstrate the problem which this can cause we can take an example of datapoints sufficient far from all other datapoints that we can treat its value as being that of the prior, which will give us a uniform marginal distribution (assuming a constant mean as our prior).

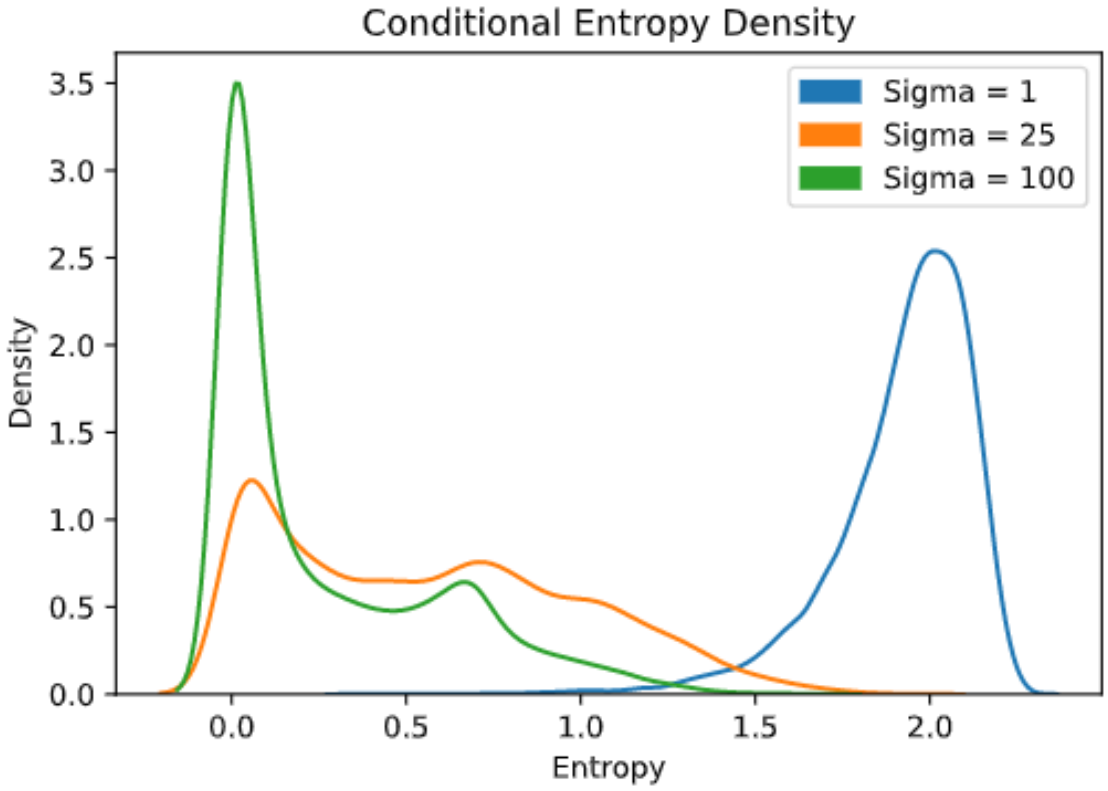


Figure 7.6: Conditional Entropy of a uniform belief with different output scales

As the length scale gets larger we no longer obtain a uni-modal distribution over the entropy. The tails of the distribution are much heavier. This would make it more difficult ot estimate the conditional entropy than in the case when it is small.

When the value of sigma is very low, we will get that the conditional entropy $\approxeq$ joint entropy in many cases (as the output probability values will change very little) leading to the statistical noise from calculating this properties having an outsized influence. When value of the scale parameter is to large, the conditional entropy will approach 0 everywhere.

These 3 different output scales and their corresponding density plots, could motivate augmenting DUE with an additional form of regularization on the output scale to avoid both of those failure modes when in DUE.

7.3.3 Training variance

In from the experiments that we have run it appears that DUE has higher variance in run to run training in comparison to training with a BNN. While attempts were made to improve the stability of training of DUE by increasing the number of inducing points and the use of NGD, the difference in this is still noticeable.

7.4 Future Extensions

- Given this experimental framework works and is on par with the SOTA performance from the (Kirsch et al., 2019) paper when using DUE inplace of BNNs. DUE performs well on more complex datasets, eg. CIFAR10. So a natural extension of this project would be to rerun the experiments on CIFAR10. This would be of minimal human effort given the framework that has been created but was not completed in the scope of this project due to time pressure.
- To deploy this as a standalone library for others to be able to easily install.
- To investigate whether some sort of regularization or constraint of the output scale kernel in DUE, could improve performance of this model with BatchBALD

Bibliography

Farquhar, S., Gal, Y., & Rainforth, T. (2021). *On statistical bias in active learning: How and when to fix it.*

Fortuin, V., Garriga-Alonso, A., Wenzel, F., RÃ¤tsch, G., Turner, R., van der Wilk, M., & Aitchison, L. (2021). *Bayesian neural network priors revisited.*

Gal, Y., & Ghahramani, Z. (2016). *Dropout as a bayesian approximation: Representing model uncertainty in deep learning.*

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., & Wilson, A. G. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *Advances in neural information processing systems.*

Hayter, A., & Lin, Y. (2012, 09). The evaluation of two-sided orthant probabilities for a quadrivariate normal distribution. *Computational Statistics - COMPUTATION STAT*, *27*, 1-13. doi: 10.1007/s00180-011-0267-z

He, K., Zhang, X., Ren, S., & Sun, J. (2015). *Deep residual learning for image recognition.*

Houlsby, N., HuszÃ¡r, F., Ghahramani, Z., & Lengyel, M. (2011). *Bayesian active learning for classification and preference learning.*

Kirsch, A., van Amersfoort, J., & Gal, Y. (2019). *Batchbald: Efficient and diverse batch acquisition for deep bayesian active learning.*

Natural gradient descent. (n.d.). https://wiseodd.github.io/techblog/2018/03/14/natural-gradient/.

Nesterov, Y. (2014). *Introductory lectures on convex optimization: A basic course* (1st ed.). Springer Publishing Company, Incorporated.

Nomura, N. (2014, 07). Evaluation of gaussian orthant probabilities based on orthogonal projections to subspaces. *Statistics and Computing*, *26*. doi: 10.1007/s11222-014-9487-8

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035).

Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Rahimi, A., & Recht, B. (2008). Random features for large-scale kernel machines. In J. Platt, D. Koller, Y. Singer, & S. Roweis (Eds.), *Advances in neural information processing systems* (Vol. 20). Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper/2007/file/013a006f03dbc5392effeb8f18fda755-Paper.pdf

Rasmussen, C. E. (2003). Gaussian processes in machine learning. In *Summer school on machine learning*.

Scaman, K., & Virmaux, A. (2019). *Lipschitz regularity of deep neural networks: analysis and efficient estimation.*

Valiant, G., & Valiant, P. (2011, 01). Estimating the unseen: An n/log(n)-sample estimator for entropy and support size, shown optimal via new clts. In (p. 685-694). doi: 10.1145/1993636.1993727

van Amersfoort, J., Smith, L., Teh, Y. W., & Gal, Y. (2020). *Uncertainty estimation using a single deep deterministic neural network.*

van der Vaart, A. (2000). *Asymptotic statistics*. Cambridge University Press. Retrieved from https://books.google.co.uk/books?id=UEuQEM5RjWgC

Williams, C. K. I. (1996). Computing with infinite networks. In *Proceedings of the 9th international conference on neural information processing systems* (p. 295-301). Cambridge, MA, USA: MIT Press.

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2017). *Understanding deep learning requires rethinking generalization.*

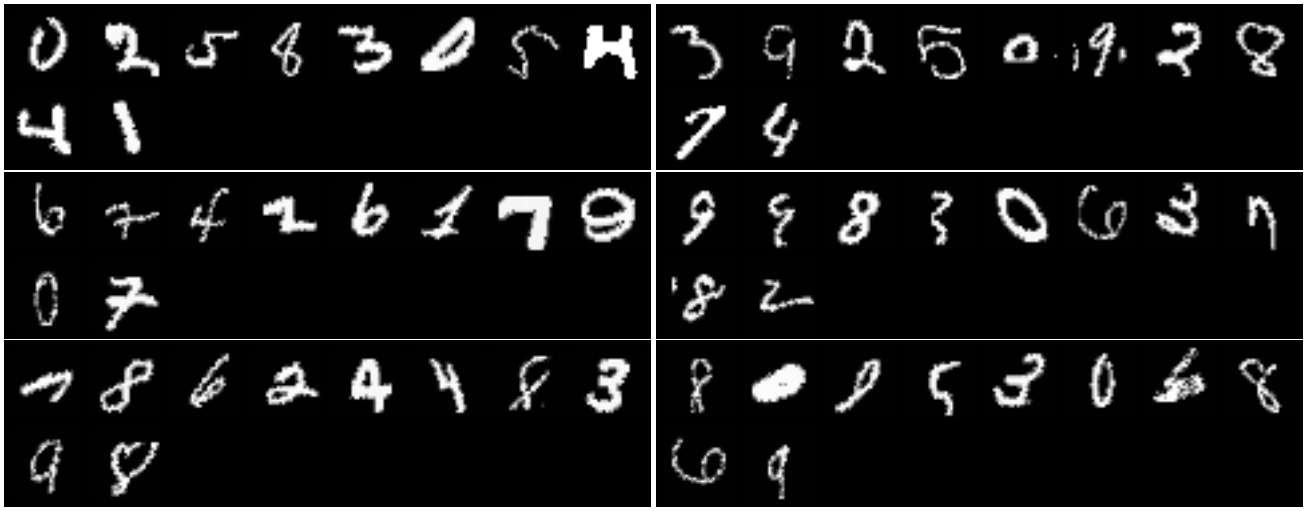
A | Appendix

A.1 Examples of selected batches

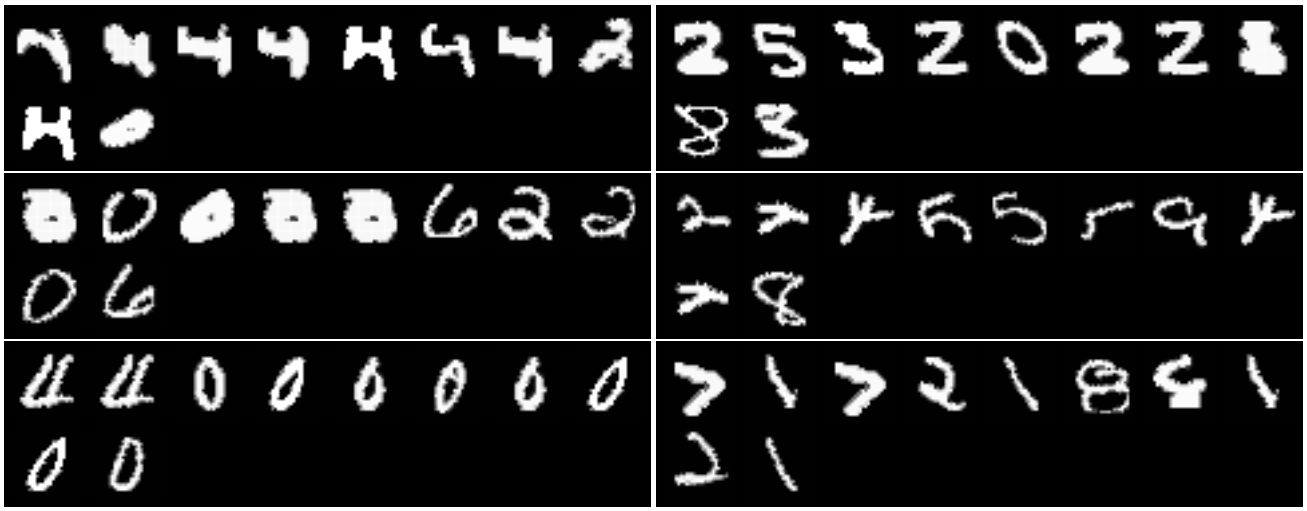
These are examples of the first 6 batches acquired by the various active learning methods on a particular run.

A.1.1 DUE

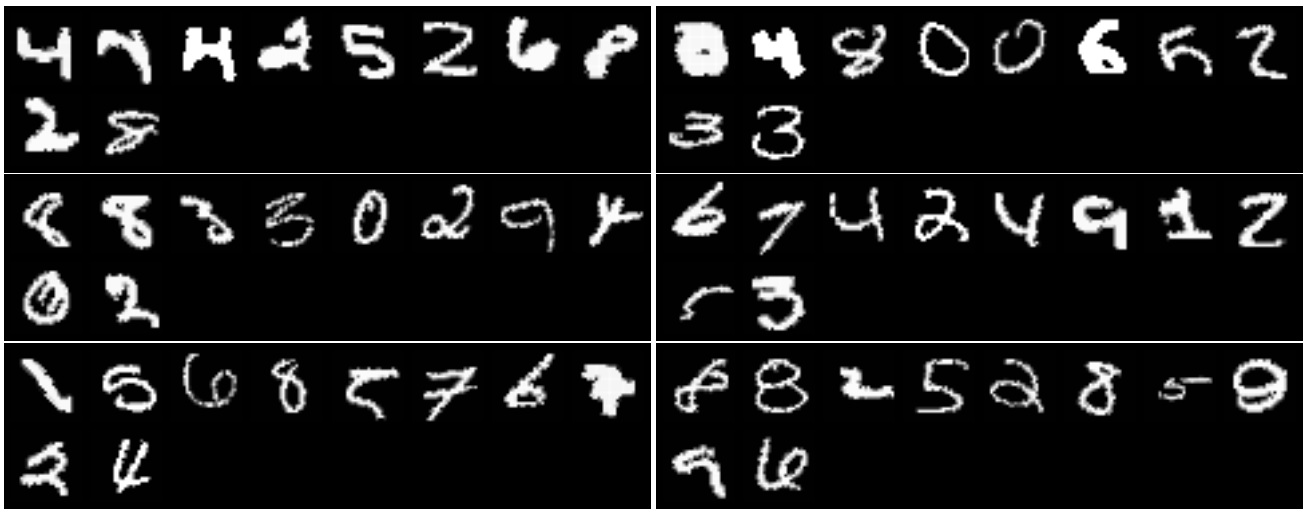
A.1.1.1 Entropy



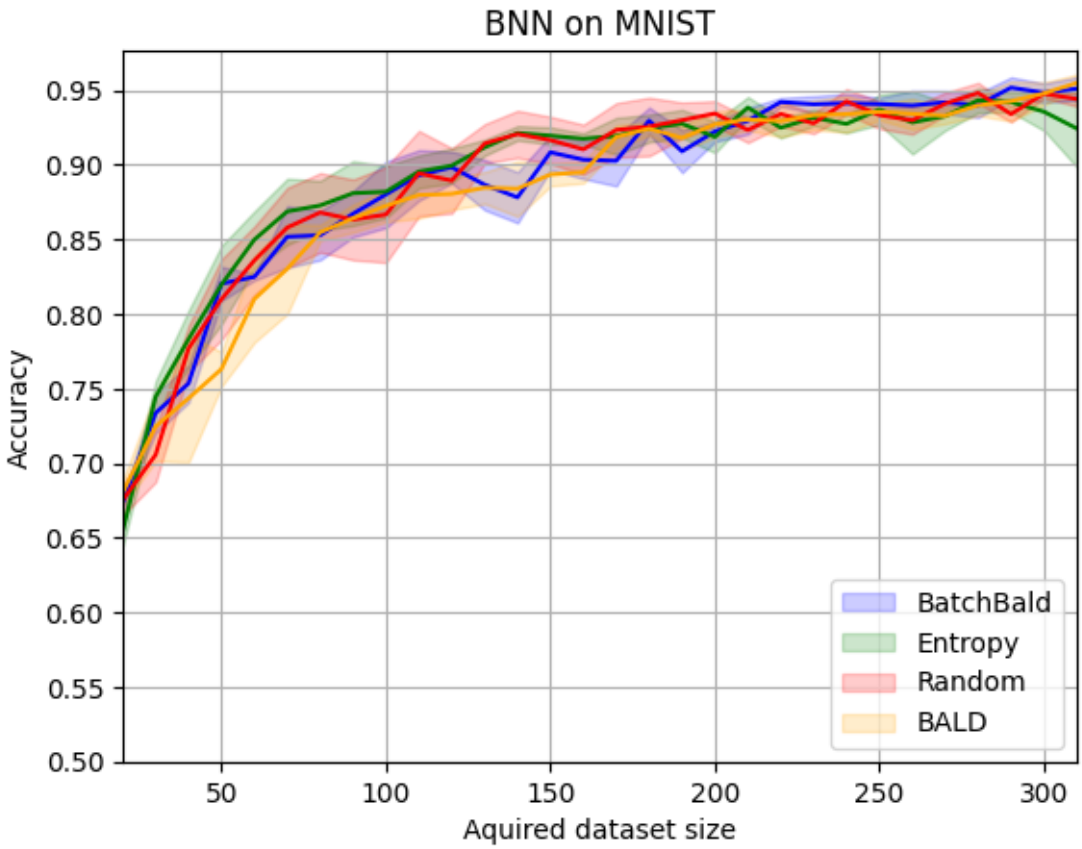
A.1.1.2 BALD

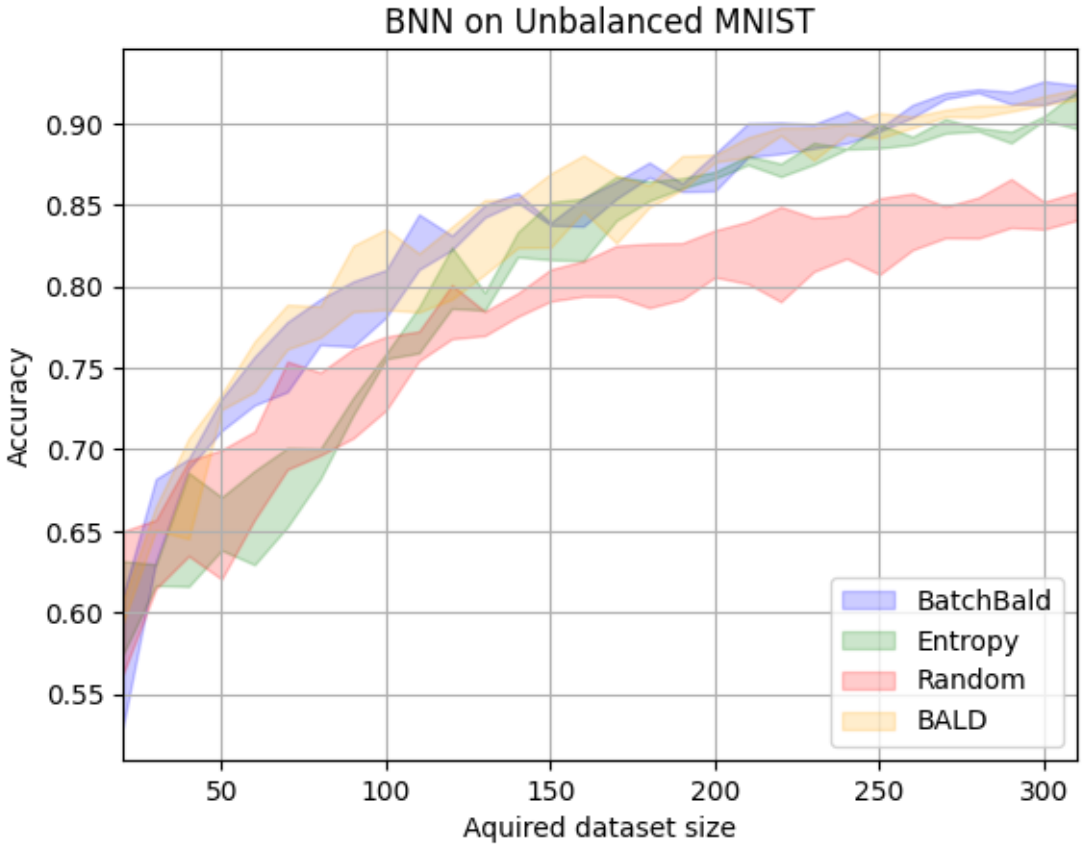


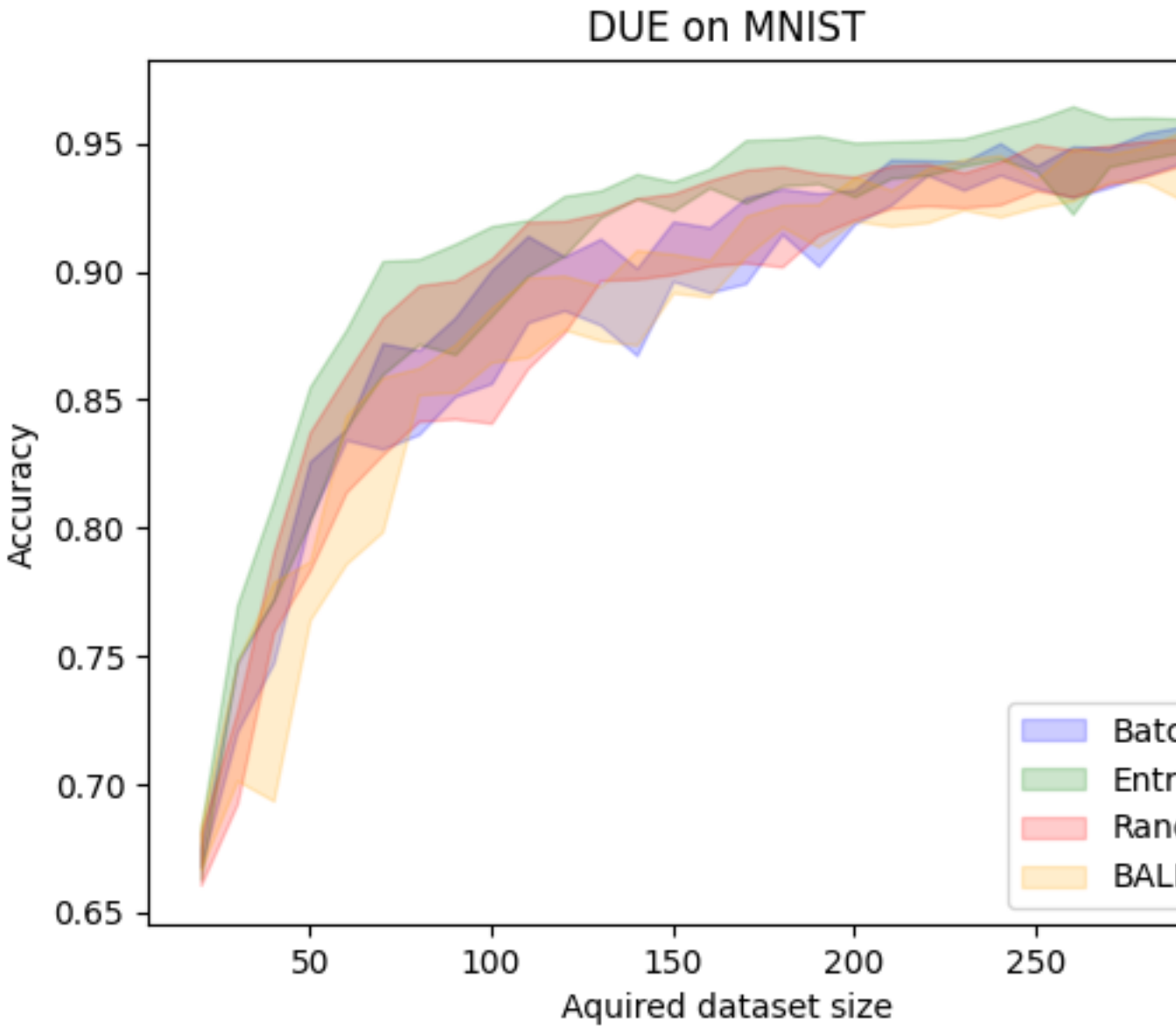
A.1.1.3 BatchBALD

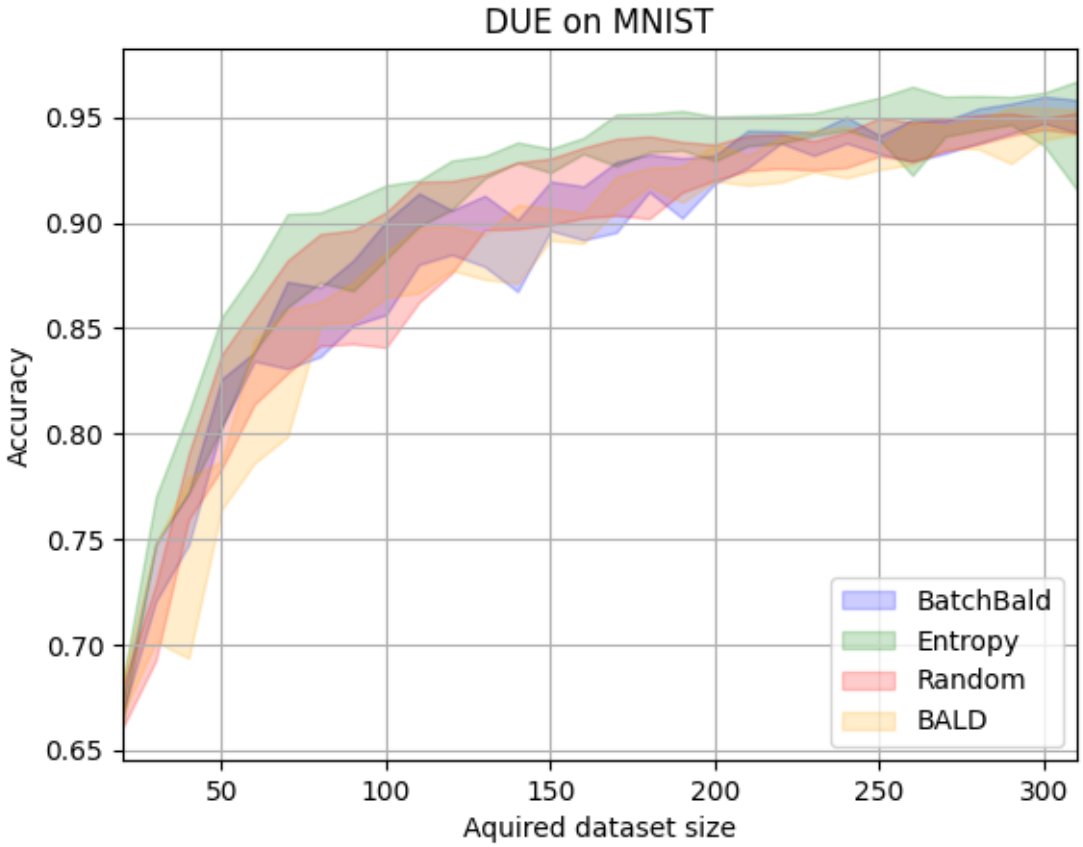


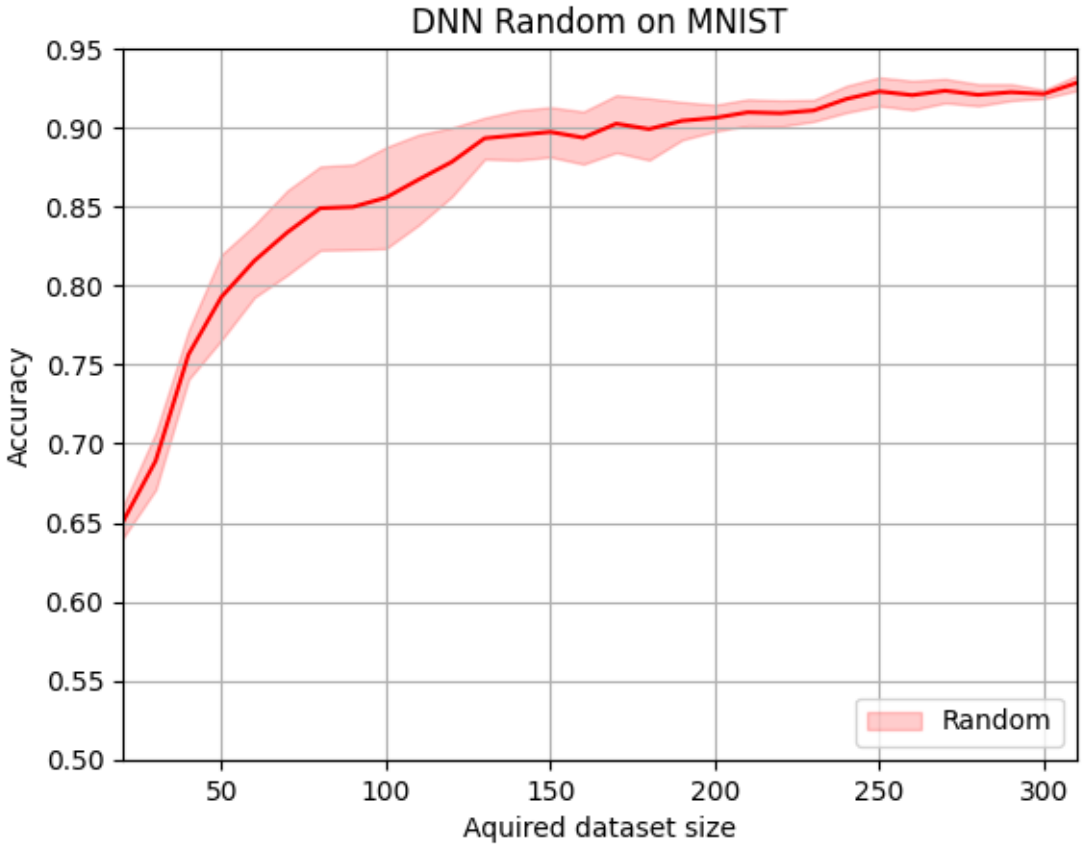
A.2 Experiment Charts




BNN on Unbalanced MNIST
Accuracy
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
50
100
150
200
250
300
Aquired dataset size
BatchBald
Entropy
Random
BALD


DUE on MNIST
Accuracy
0.95
0.90
0.85
0.80
0.75
0.70
0.65
50
100
150
200
250
Aquired dataset size
Bato
Entr
Ran
BAL


DUE on MNIST
Accuracy
Aquired dataset size
BatchBald
Entropy
Random
BALD


DNN Random on MNIST
Accuracy
Aquired dataset size
Random