Project Machine Learning :
— Milestone 3 —

Denis Semko

February 28, 2023

1 Introduction

This project involves the implementation of the methods outlined by Blundell et al. in their paper **Weight Uncertainty in Neural Networks** (2015) over the course of three milestones. The first milestone includes the implementation of the Bayes by Backprop (**BBB**) algorithm in Python, as well as the training of baseline "vanilla" multilayer perceptrons (**MLP**s) for comparison against Bayesian neural networks (**BNN**s). The second milestone successfully recreates the experiments from the aforementioned paper, training BNNs for a simple regression problem and the classification of flattened MNIST data. Through this, we were able to observe the ability of BNNs to assert uncertainty in "out-of-distribution" data for regression tasks, their superior accuracy compared to MLPs in classification tasks, and the potential for extensive pruning of classification BNNs without increasing (or even decreasing) the test error.

The final milestone of this project involves expanding upon the methods presented in the paper by Blundell et al.. With our project supervisor we have agreed to the following goals for the final milestone of the project:

- Implementing and evaluating the BBB algorithm using the vectorizing map (**vmap**) higher-order function
- Implementing and evaluating the Local Reparameterization Trick (**LRT**) as described by Kingma, Salimans & Welling in **Variational Dropout and the Local Reparameterization Trick** (2015)
- Analysing the results of inferring on out-of-distribution (**OOD**) classification data with the trained MNIST BNN
- Evaluating the effect of adversarial attacks on BNNs compared to traditional MLPs

2 Automatic Vectorization

The vmap higher-order function was recently popularized in the field of machine learning by the release of the Google JAX machine learning framework, and was consequently implemented for the PyTorch framework, which we make use of in our project.

If consecutive iterative steps in a for-loop do not depend on each other, that for-loop can generally be parallelized: while "simple" multiprocessing and multithreading can yield a substantial speedup, vectorized implementations can additionally make use of dedicated SIMD (single instruction, multiple data) features that modern CPUs have, leading to even greater speedups. Vectorized implementations are also a necessity when one wants to make use of general-purpose computing on GPUs (which for the most part are constructed on SIMD instructions themselves), which is an integral part of modern-day deep learning.

It is common for iterative steps in programs to be implemented using for-loops, as this allows for simpler and more easily understandable code. This is because operations performed within these loops are typically applied to individual scalar values, as opposed

to vectors, which would add an extra layer of complexity to all operations and functions performed on the data.

The vmap higher-order function is an autovectorizer, meaning it automatically maps a given function over one axis of a given data tensor, resulting in vectorized operations. This process is a classic trade-off between execution speed and working memory, as the overall speed of the program is decreased at the cost of a larger working memory footprint. Factors such as CPU speed, number of cores, available memory, and GPU can greatly impact this trade-off. While vmap does not offer any novel functionality per se, it greatly simplifies and automates the process of vectorization, much like how autodiff automates differentiation for deep learning. Thus, similar to autodiff, vmap takes over a crucial yet time-consuming and error-prone task from the developer, enabling a simpler scalar implementation of a function to be vectorized effortlessly.

For our experiments we make use of PyTorch's functorch library, which seeks to port the composable function transforms from Google JAX, including vmap, to the PyTorch framework. Our goal is to vectorize the process of drawing x individual samples from our BNN, which originally loops over a subroutine which performs the forward pass and calculates the ELBO for each sample. Instead of performing this task sequentially, we would like the forward passes and weight sampling to be performed in parallel, vectorized even. To achieve this, we defined an argument-function to vmap to execute the forward pass, return the ELBO complexity and likelihood costs, as well as the final prediction. After the vmap higher-order function completes all tasks in parallel, we receive the outputs we would expect from our sequential implementation, only with an additional axis/dimension each, over which we must compute the mean to arrive at our final ELBO and model prediction.

We conducted experiments to train our BNN using both the original sequential implementation and the new vmap implementation on both, the curve regression task and the MNIST classification task. We utilized a series of samples from 1 to 10 during training. As mentioned above, the performance and speedup factor of a vectorized implementation may depend heavily on the hardware available; for our experiments we have used the available Grid Engine cluster for the project with a total of 64 CPU cores, an Nvidia A100 GPU and 755GB of available RAM. The total amount of trainable parameters in the regression task is roughly 320,000 while for the classification task its roughly 2,500,000. The training data consists of 1,000 scalar values for regression and 60,000 vectors of size $(28 \times 28 =)$ 784 for classification.

In our experiments, for sample sizes ranging from one to ten, the increase in computation time per training epoch with each increase in the number of randomly drawn samples appears to be nearly linear. By fitting a linear regression line to the average training epoch time as a function of the number of samples, we obtained a near-perfect R-Squared coefficient of over 0.99 in all cases. By examining the regression coefficient for the number of samples, we can approximate the time added with each additional sample. The results can be seen in Table 1. We found that for each additional sample, we observed a speedup of 2.15 for the classification task and 7.0 for the regression task when using the vmap implementation instead of our original sequential implementation. The average training time per epoch for differing amounts of sampled models compared to the original, sequential implementation can be observed in Tables 2, and Figures 1.

Implementation	**Regression**	**Classification**
Sequential	0.070	8.8
Vectorized map	0.010	4.1
vmap + LRT	0.007	0.8
# Parameters	320k	2.5m

Table 1: The slope coefficient after regressing the average time per training epoch on number of models sampled during training. R-Squared of over 0.99 for all regressions.

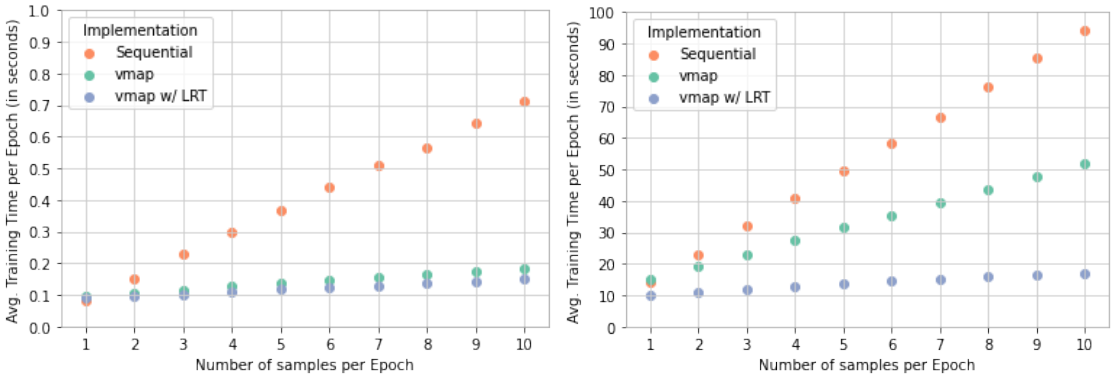
# Samples	Sequential	Vectorized map (vmap)	vmap + Local Reparameterization Trick
1	0.083	0.096	0.090
2	0.151	0.103	0.094
3	0.229	0.115	0.100
4	0.298	0.125	0.107
5	0.368	0.136	0.116
6	0.442	0.144	0.122
7	0.508	0.153	0.129
8	0.564	0.163	0.135
9	0.642	0.172	0.139
10	0.713	0.182	0.149

(a) Regression.

# Samples	Sequential	Vectorized map (vmap)	vmap + Local Reparameterization Trick
1	14.3	15.1	10.2
2	23.0	19.0	11.0
3	32.3	23.1	11.9
4	41.0	27.4	12.7
5	49.5	31.7	13.6
6	58.2	35.4	14.4
7	66.7	39.4	15.1
8	76.2	43.6	15.9
9	85.3	47.7	16.4
10	94.0	52.0	17.1

(b) Classification.

Table 2: Average training time per epoch (in seconds) for different amounts of sampled BNN models (rows) for different implementations (columns).



(a) Regression.

(b) Classification.

Figure 1: Average training time per epoch in seconds (y-axis) for different amounts of sampled BNN models (x-axis) for different implementations (colors).

3 The Local Reparametrization Trick

The Local Reparameterization Trick (LRT) as presented by Kingma, Salimans & Welling in the paper **Variational Dropout and the Local Reparameterization Trick** (2015) seeks to enhance both statistical and computational efficiency in the calculation of stochastic gradients for variational Bayesian inference of posterior distributions over model parameters.

The LRT achieves statistical efficiency "by translating uncertainty about global parameters into local noise that is independent across datapoints in the mini-batch", that is, by sampling a separate weight matrix for every training sample in the mini-batch, thereby resulting in a covariance of zero between the ELBO estimates of each separate training sample. This ultimately leads to a variance that is inversely proportional to the mini-batch size, resulting in faster convergence.

In this setting, we find ourselves dealing with mini-batches of size N and a weight matrix of a BNN layer with dimensions $n \times m$. This results in a total $N \times n \times m$ samples for one layer, which can quickly become computationally infeasible to compute. However, Kingma, Salimans, & Welling propose a solution to this computational challenge through the proposal of the **local** reparameterization trick, which allows for the direct sampling of activations (given that the posteriors are Gaussian-distributed) instead of having to sample the weight matrices and then computing the activations. The reparameterization is defined in Equation 1, with $\zeta_{n,j} \sim \mathcal{N}(0, 1)$, and with $\gamma_{n,j}$ and $\delta_{i,j}$ as defined in Equations 2 and 3.

$$b_{m,j} = \gamma_{n,j} + \sqrt{\delta_{i,j}}\zeta_{n,j} \tag{1}$$

$$\gamma_{n,j} = \sum_{i=1}^{n} a_{n,i}\mu_{i,i} \tag{2}$$

$$\delta_{i,j} = \sum_{i=1}^{n} a_{n,i}^2\sigma_{i,i}^2 \tag{3}$$

This strategy significantly reduces the computational effort per layer from $N \times n \times m$ required samples to $N \times m$ samples, without compromising the quality of the results. Because we sample activations instead of weights, which we need to estimate the KL divergence part of the ELBO Monte Carlo estimate, we must resort to using the closed-form solution of the KL divergence (see Equation 4) - which restricts us to only using the Gaussian prior.

$$\begin{aligned} KL(p, q) &= -\int p(x) \log q(x) dx + \int p(x) \log p(x) dx \\ &= \frac{1}{2} \log(2\pi\sigma_2^2) + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}(1 + \log 2\pi\sigma_1^2) \\ &= \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \end{aligned} \tag{4}$$

We have implemented the Local Reparameterization Trick as described in the paper by Kingma, Salimans & Welling, while also making use of the faster auto-vectorized implementation from the previous chapter. We trained a model for the regression and classification tasks described above, reusing the hyperparameters of the non-LRT implementation making use of a Gaussian prior. Regarding the first claim by the paper; in our specific experiments we have not found that the models implementing LRT converge faster, or that the loss/error displays less variance (see Figure 2 for the learning curve for the classification task).

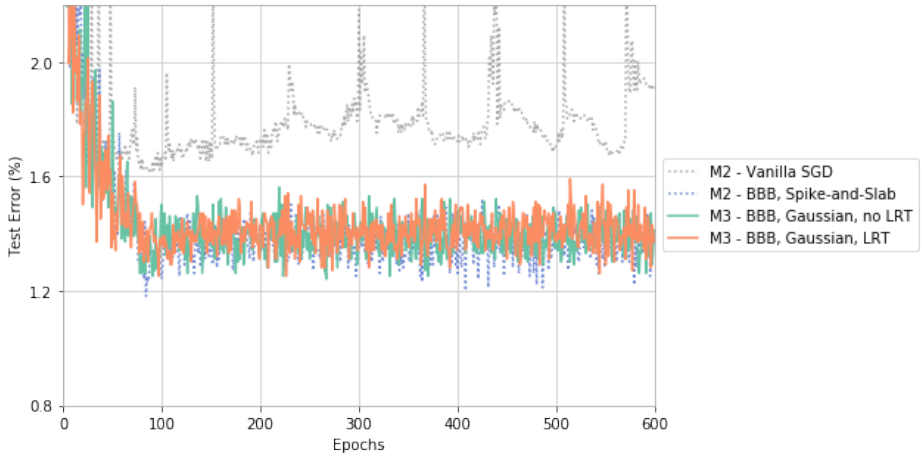


Figure 2: Test error as a function of training epoch. 600 epochs of training on MNIST.

We did observe a computational speedup when comparing LRT implemented with vmap compared to the original method with vmap. Again, the average training time per epoch appears to be linearly proportional to model samples drawn; fitting a linear regression results in an R-squared coefficient of greater than 0.99 for regression and classification. Per Table 1 the speedup factor of the LRT with vmap compared to the original method with vmap is roughly 1.4 for our regression task, and 5.1 for classification. When compared to the original sequential implementation of Milestones 1 and 2, vmap + LRT provides a speedup factor of 10.0 for the regression problem, and 11.0 for classification. The average training time per epoch for differing amounts of sampled models compared to our other implementations can be observed in Tables 2(a), 2(b) and Figures 1(a), 1(b).

4 Inference on OOD Classification Data

The key idea behind the introduction of BNNs is to take a probabilistic approach to inference/prediction, in contrast to the traditional MLPs with point-estimates for prediction. When inferring with a BNN on the same type of data it was trained on, we would like the model to have high accuracy with high confidence/low variance, while when inferring on unseen datasets (out-of-distribution (OOD) data), we would like the model to portray lower confidence/higher variance.
In the second milestone of this project, we have demonstrated the uncertainty which our trained BNN shows when performing a regression on an input-domain it has not seen during training. Consequently, we want to investigate the uncertainty of our BNN when performing classification on OOD data. We let our model (trained on MNIST) infer on 3 different datasets; namely FashionMNIST, KMNIST and OMNIGLOT [1]. Figure 3 gives a glimpse of these datasets.

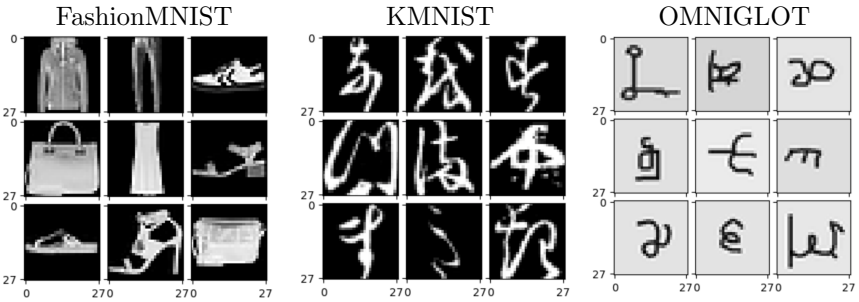


Figure 3: 9 random datapoints out of each dataset.

A first trivial result is that the test accuracy on these datasets is very poor; it never exceeds 10 %. Next we fed randomly-picked datapoints from each of the 4 datasets to our trained BNN, drawing 100 model-samples for each datapoint. Figure 4 displays our 4 randomly selected datapoints.

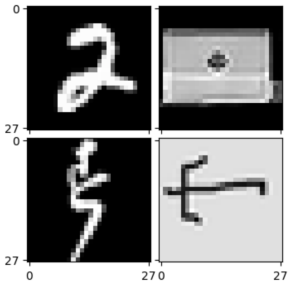


Figure 4: One random datapoint out of each dataset for inference with BNN.

The BNN consistently manages to predict (where a prediction corresponds to the argmax over the softmax scores of a sampled model) the correct label for the MNIST datapoint

[1] In OMNIGLOT, the datapoints are comprised of 105 x 105 pixels. These were downsized to 28 x 28 to fit our model input.

over all samples. However, as can be seen in Figure 5, when it comes to inferring on the other datasets, we notice more variance in the predicted classes - this can be interpreted as a first indication of higher uncertainty from our model in its predictions.

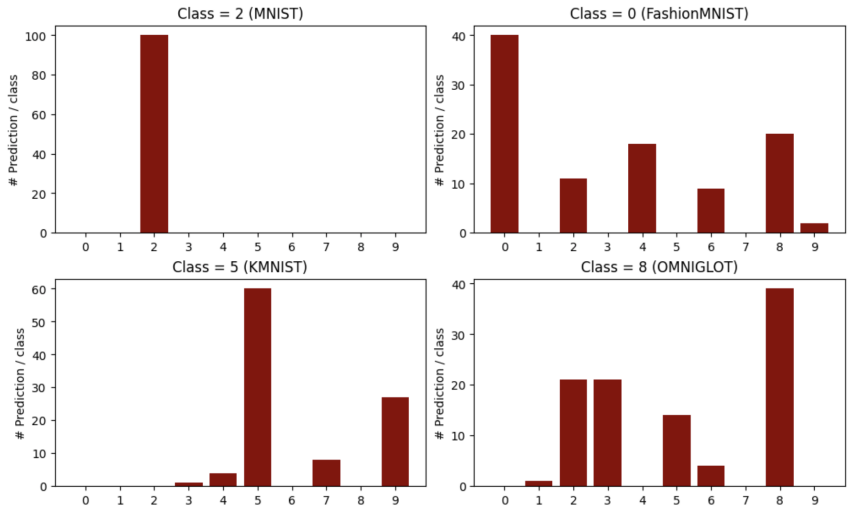


Figure 5: For each of the 4 datapoints, BNN produces 100 inferences. A histogram depicts the distribution of predicted classes.

It is tempting to view the output of the softmax layer, often-times perceived as probabilities, as a representation of uncertainty; but this would be erroneous. Softmax outputs represent the relative likelihood of a set of classes, but do not necessarily reflect the total degree of uncertainty in the prediction. Even when the largest softmax output is close to 1, the model may still be uncertain about its prediction. We guide ourselves by Chapter 4.2.2 of Yarin Gal's dissertation "**Uncertainty in Deep Learning**" (2016), where the author measures uncertainty by looking at the spread of softmax scores, rather than at the mean of the softmax score. Accordingly, we looked at the distribution of softmax scores that the BNN outputs at each of the 100 samples for the top predicted class (that is, the argmax of the mean over all the softmax outputs of our sampled models). The histograms in Figure 6 show how 100 inferences on the MNIST datapoint return a probability (softmax score, to be more precise) of 1.0 for the ultimately predicted class "2". In contrast, inferring on the OOD datasets yielded significantly less probabilities of 1.0 for the ultimately predicted class, with a much wider spread of values, and even a considerable amount of predicted softmax scores below 0.1. These predictions can ultimately be interpreted as having low uncertainty.

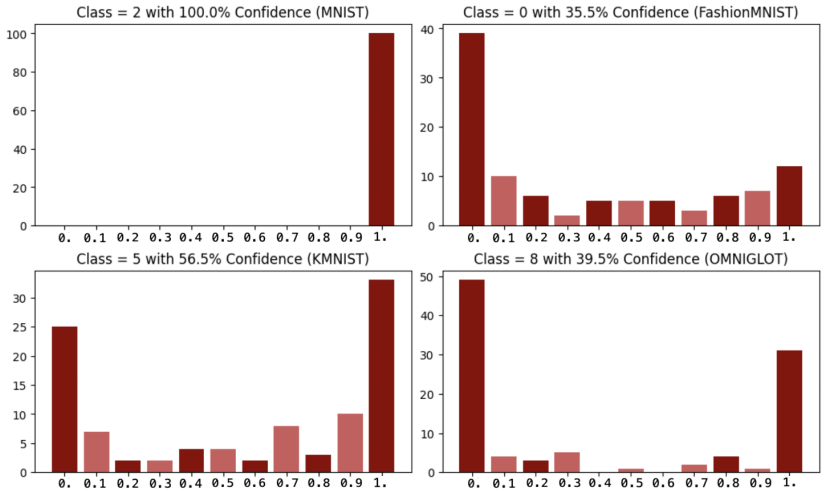


Figure 6: After 100 inferences on the same datapoint, the most frequent class is retrieved as the overall prediction. The 100 probabilities for this same class are plotted in a histogram.

One last point to highlight is that the BNN is remarkably trying to recover its knowledge from the training to infer on unknown datasets. Notice the similarities of the KMNIST datapoint to a 5 (perhaps also to a 9). One could also argue that the handbag in FashionMNIST is closest to a 0. As a non-trivial result, a datapoint from an OOD dataset that closely resembles an MNIST-class is going to be classified with more confidence than a datapoint resembling no digit at all.

5 Adversarial Attacks on Bayesian Neural Nets

In this section, BNNs are examined for their robustness against adversarial attacks. For this, a trained network will be tricked by an attacker who modifies an openly known input to cause the neural network to misclassify the input. The main objective is often to introduce the least amount of disturbance to the input data necessary to get the intended misclassification. One of the currently most popular adversarial attacks, the Fast Gradient Sign Attack (FGSA), will be used in this experiment. The FGSA disturbs the initial input x as follows:

$$x_{perturbed} = x + \epsilon * sign(\nabla_x J(\theta, \boldsymbol{x}, y))$$

The attack is simple yet very effective. It uses the exact same mechanism neural networks are trained with, namely the cost gradients. The concept is straightforward: the attack modifies the input data to maximize the cost based on the same backpropagated gradients (as for training) rather than attempting to minimize the cost by altering the weights based on them. In other words, the attack maximizes the cost by adjusting the input data rather than the weights (the model's parameters stay untouched, only the input data is corrupted).

In this experiment, we compare the robustness of three algorithms towards adversarial attacks: Our plain vanilla MLPs from Milestone 1 - with and without dropout - and our final implementation of a BNN, all trained on the MNIST dataset. Five attacks were ran on every model, each time with a little increment in ϵ $(= 0, 0.05, 0.1, 0.15, 0.2)$. Intuitively we would expect that the larger the epsilon, the more noticeable the perturbations, but the more effective the attack in terms of reducing the model accuracy. As a reference, Figure 7 displays some examples of datapoints with varying degrees of perturbation.

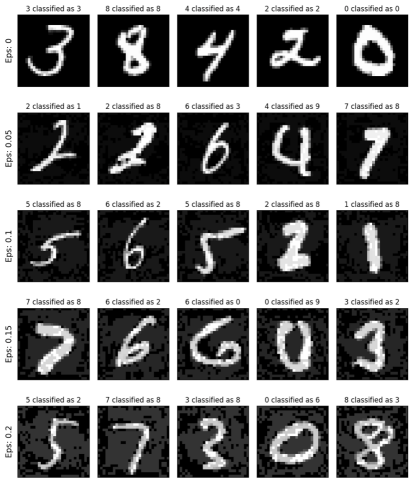


Figure 7: Examples from the MNIST dataset with varying degrees of adversarial attacks.

Figure 8 shows how the 3 models behave in regards to the test accuracy, while increasing ϵ in the data corruption process. Notice that the first value for ϵ is 0, which corresponds to no attack since the data stays unchanged.

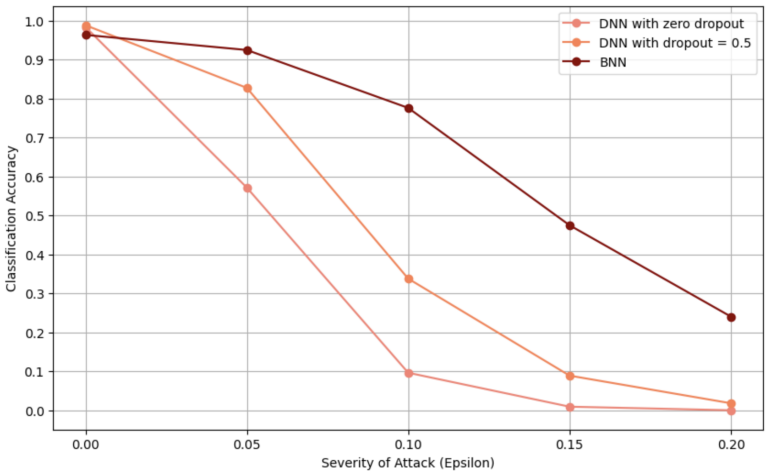


Figure 8: Test accuracy as a function of severity of attack.

As expected, the BNN shows the best robustness during an active FGSA. The use of a distribution over weights reinforces the stability and generalizability of the feature representations learned by the network, leading to more reliable predictions on new data. Up until $\epsilon = 0.1$, one could argue the BNN keeps a decent accuracy. The MLP without dropout is already far off with $\epsilon = 0.05$. While the BNN predicts with near 50 % accuracy at $\epsilon = 0.15$, both MLP reached accuracies of less than 10 %.

6 Conclusion

After having successfully recreated the experiments by Blundell et al. in Milestone 2, we have now significantly enhanced the performance of the BNN model training procedure through auto-vectorization of the code and implementation of the Local Reparameterisation Trick, as proposed by Kingma, Salimans, and Welling. This resulted in a remarkable reduction in runtime by a factor of 10 for regression and 11 for classification tasks, at the cost of being restricted to a Gaussian prior and posterior over the model parameters.

The outcome of our experiments indicates that classifying OOD data results in a heightened predictive uncertainty, manifesting as a broader range of predicted softmax scores, in comparison to performing inference on the in-distribution data that the model was originally trained on. This is similar to the phenomenon observed for OOD regression inference last milestone. That being said, BNNs are not a silver bullet when it comes to OOD detection. Our experiments with the MNIST dataset have revealed that a small amount of datapoints from it can exhibit a broad range of softmax scores (especially those, that are misclassified by our trained model). On the other hand, we have encountered instances where OOD datapoints displayed extremely high softmax scores, yet bore little resemblance to any of the MNIST classes.

In our experiments we have found that BNNs exhibit a significantly higher level of robustness against adversarial inputs in comparison to conventional MLPs. This finding highlights the potential of BNNs as a promising solution to addressing the vulnerability of MLPs to adversarial attacks.

In conclusion, our results over the course of this 3-milestone project support the findings of Blundell et al. in their paper "Weight Uncertainty in Neural Networks" (2015) and demonstrate the effectiveness of BNNs in modeling uncertainty, achieving improved performance over MLPs, and providing robustness against adversarial inputs. However, the increased training time remains a challenge and further research is needed to address this issue. Our enhanced training procedure provides a step forward in this direction, as it significantly reduced the runtime for regression and classification tasks, but being restricted to a Gaussian prior and posterior over the model parameters may limit the potential for further improvement. Overall, our work highlights the potential of BNNs as a promising solution for tackling many challenges in deep learning.