Project Machine Learning :
— Milestone 1 —

Denis Semko

February 28, 2023

1 Introduction

This project consists in implementing the methods presented by Blundel et al. in the paper **Weight Uncertainty in Neural Networks** (2015). The algorithm, called *Bayes by Backprop* (BBB), will be implemented and empirically evaluated on a classification and regression task, and later compared to vanilla neural networks.

In Milestone 1, the data is inspected and preprocessed in order to gain insights and to tailor the input to the algorithm. Furthermore, baseline vanilla models from the original paper are implemented to serve as a comparison to the Bayesian Neural Networks (BNNs) - in this context Multi-Layer Perceptrons (MLPs), which implement the BBB algorithm for its linear layers. We create a first iteration of BNNs for the classification and regression tasks at hand, which yield promising results. These results are then discussed and prospects are being made for Milestone 2.

2 Dataset overview

2.1 MNIST Dataset for Classification

The MNIST dataset contains 70.000 labelled images of handwritten digits ranging from class 0 to class 9.
60.000 are for training and 10.000 are for testing, but this split is adjustable depending on the user's preferences. For instance, the training set could be further broken down into a training set and a validation set. Or, a smaller number of datapoints could also be used in a cross-validation manner.
Technically, every data point is a (28x28)-tensor of pixels whose values are ranging from 0 to 256 (grayscale). Equivalently, one could interpret the samples as 784-dimensional vectors (flattened tensor), as in Figure 1.

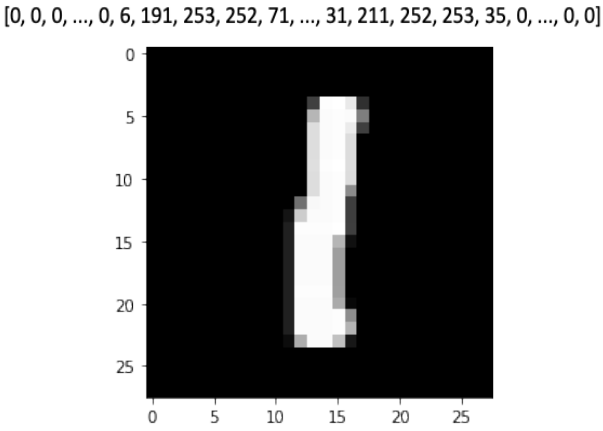
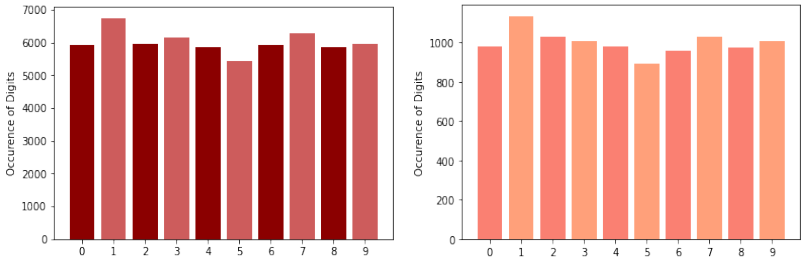


Figure 1. Vector representation and Graphic representation

In Figure 2, 36 data points were randomly picked from the training set. First of all, notice that this dataset comes with some irregularities. For instance, a seven can have no horizontal bar. There can be "noise" around the digit. Digits can appear to be very thick. Some classes are close to each other like class 5 and class 6 or class 1 and class 7. This brings challenges in regards to accurate classification.

Figure 2. 36 randomly chosen data points from the MNIST datasets.

Before training, one needs also to ensure that the classes are equally represented in the training set and the test set. As figure 3 shows, the distribution is relatively uniform across the different classes. However, class 3 is larger than class 2 in the training set while the opposite is true in the test set.



(a) (b)

Figure 3. Distribution of classes over the training (a) and testing (b) splits

A third aspect is dimensionality. As stated earlier, the data is living in a 784-dimensional space, which can be considered "high-dimensional". We ran a Principal Component Analysis (PCA) to gain insights into the dimensional structure of the dataset. It appears that the data would live in a substantially smaller submanifold as 95 % of the variance can be explained by the first 150 Principal Components (PC) (Figure 4.). As a result, we could reduce the data dimensionality from 784 to 150 and keep 95 % of the information. This would highly reduce the computational complexity of the model.

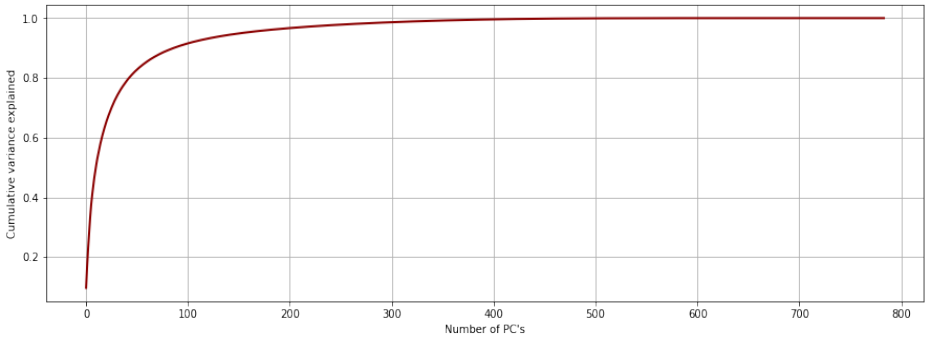


Figure 4. Cumulative variance explained in function of the number of PC used. The first PC explains 10 % of the total variance. The 10 first PC's 49 %. The 50 first PC's 82 %. The 75 first PC's 88 %. The 100 first PC's 91 %.

At last, we divide all pixels values by 126 as the pre-processing step, as is done in the original BBB paper. This could be seen as a grayscale-to-binary conversion, which allows the model to learn on only 3 different types of input values. (0, 1 or 2)

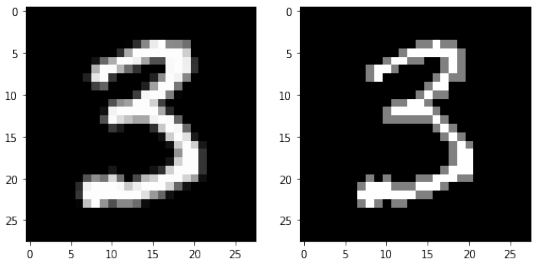


Figure 5. Pre-processing step : Divide all pixel values by 126. This transforms the input domain from $x \in \{0...256\}$ to $x \in \{0, 1, 2\}$. Similar to a grayscale-to-binary conversion.

2.2 Data generated from a curve for Regression

In order to replicate the regression experiments from the original paper, we reuse the data generating function as defined by the authors:

$$y = x + 0.3sin(2\pi(x + \epsilon)) + 0.3sin(4\pi(x + \epsilon)) + \epsilon \tag{1}$$

with $\epsilon \sim \mathcal{N}(0, 0.02)$. Furthermore, we randomly draw the x-values from a uniform distribution: $X \sim \mathcal{U}(0, 0.5)$ for the training and testing procedure.

We define a "lazy" data generator which draws new data batches from the above defined function when they are needed during training and validation of the artificial neural network. We initialize the data generators with random seeds to ensure our experiments are deterministic and thus reproducible.

For the evaluation and comparison of our baseline models as well as the implemented BBB algorithm in later milestones of the project, we generate 1000 datapoints from the same curve and its parameters as defined above, except we now draw our x-values from $X \sim \mathcal{U}(-0.2, 1.1)$ (see Figure 6).

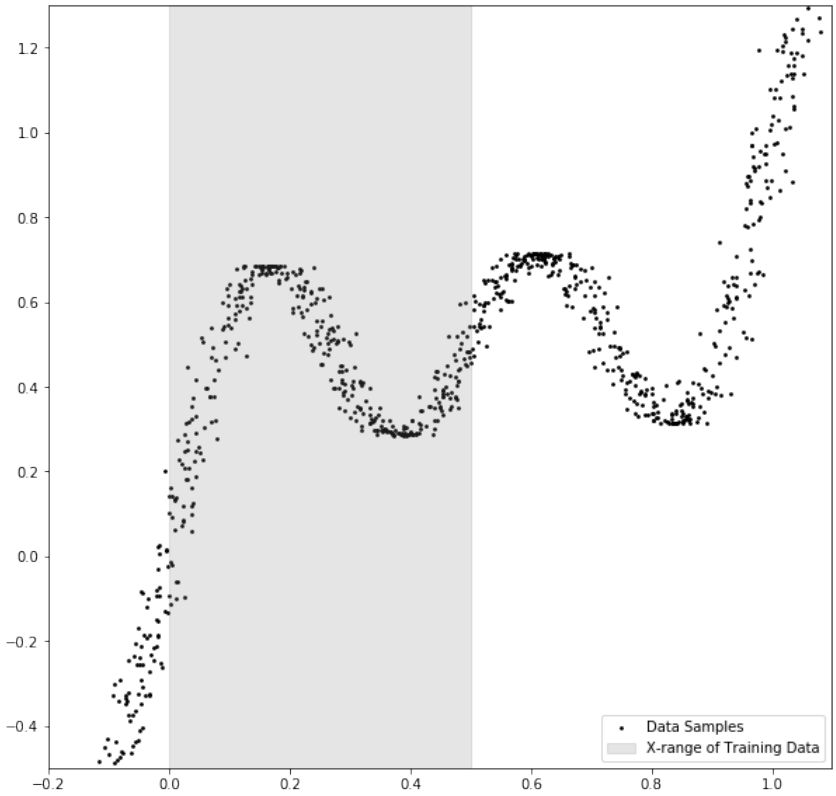


Figure 6. Randomly generated data from our curve function.

In summary, we now have a dataset with one randomly generated input feature, X, generated from a uniform distribution, and the target value y, generated by plugging our Xs into equation (1), subject to some random normal additive noise ϵ.

3 Comparison baselines

3.1 Classification

The paper suggests the following architecture as for the plain vanilla NN: A flattened (28x28)-neuron input layer, two hidden (dense) layers of each 1.200 Rectified Linear Units (ReLU), with a Softmax activation function at the output, applied on the ten output values per sample. We use the Cross Entropy Loss as the loss function, which will be the expression to minimize. Moreover, in order to progress along this objective function, we used the Stochastic Gradient Descent (SGD) optimizer as per the original BBB paper. We trained the above defined model first without dropout, and than with a dropout of 50% for its weights. Both models were trained with a learning rate of 10^{-1} [1] and with mini-batch-size of 128. The original MNIST training set is randomly split in a 50.000 datapoint-training set and a validation set consisting of 10.000 samples. In Figure 7, the test error in function of the epochs for both models is plotted. As expected, the test error of the dropout algorithm decreases in a significantly unsteadier manner than the mode without dropout, but it results in a lower test error in the end.

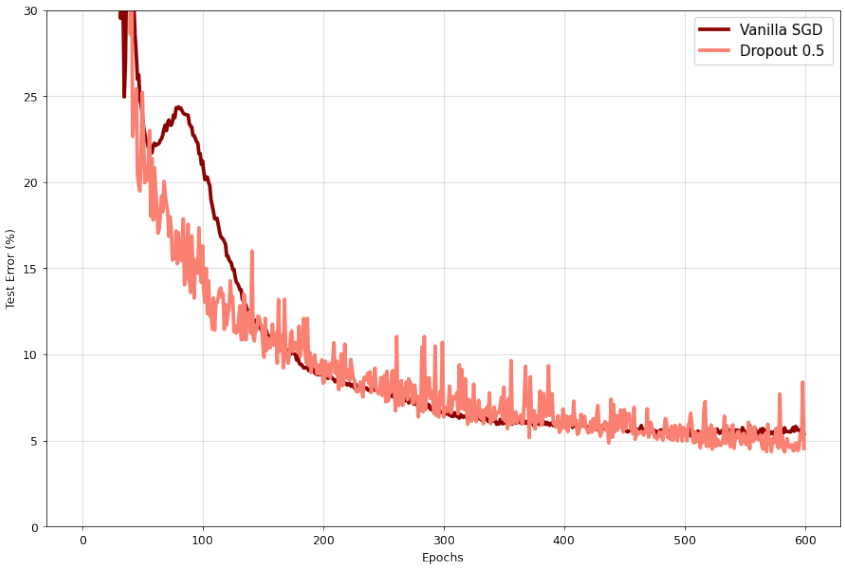


Figure 7. The test error decreases (accuracy increases) while training progresses. Finally, the Dropout algorithm converges to a lower test error than vanilla SGD.

The test error is computed as follows. The output of the previously described NN is a non-normalized [2] vector of 10 values. The index of the highest value ultimately give us the predicted class. Therefore, by comparing this index with the ground truth label for each data point in the test set, we were able to reconstruct a test error estimate at each epoch.
To continue, we verified the weights' distribution (2.392.800 weights) of both architectures and plotted them (see Figure 8.). Notice that the weight distribution resulting from dropout training has somewhat fatter tails and a lower peak. We expect the BBB-algorithm to showcase an even larger variance.

[1] This is higher than the learning rates specified by the authors, which are $10^{-3}, 10^{-4}$ or 10^{-5}

[2] The Softmax function is directly integrated in the the Cross Entropy Loss function. As a result, the immediate output of the NN is not yet normalized through Softmax.

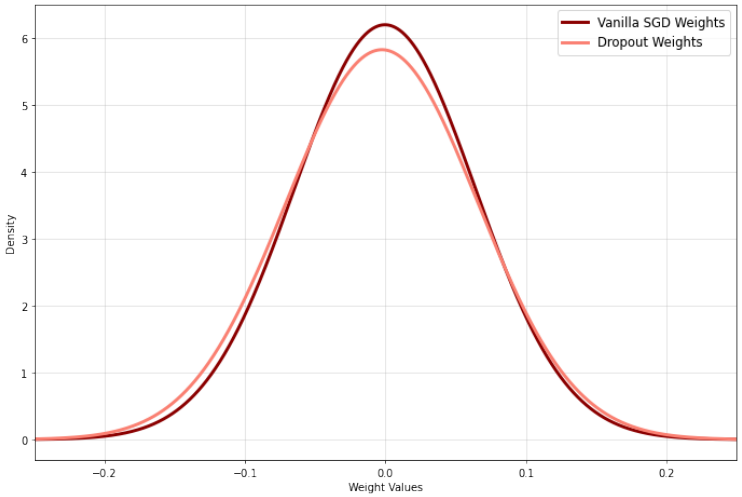


Figure 8. The dropout weights are slightly less densely distributed than the Vanilla SGD weights.

3.2 Regression

As our baseline models, analogously to the original paper, we use a standard multi-layer perceptron (MLP). Unfortunately the paper does not specify what parameters exactly were used to obtain their experimental results, we thus have to implement our own MLPs which show the best performance loss-wise, as well as computationally (training time and memory footprint).

Apart from the input and output layers, we defined one hidden layer of 400 neurons, connected from the input and to the output respectively with linearly rectified units (ReLU). We used Adam as our optimization algorithm due to a much faster observed convergence compared to Stochastic Gradient Descent. We have not seen any major differences when using different loss functions for regression implemented in the PyTorch machine learning framework, we thus opted for a simple L1-Loss, also known as the Mean Absolute Error (MAE).

As a second baseline model we use an MLP with added dropout, a method that is often addressed in the original paper, as it tackles a similar problem to BBB. For this we simply reuse the standard MLP defined above, adding a 50% dropout regularizer to its weights.

In the original paper the baseline model generates a prediction range with some interquantile ranges which indicate the certainty of the model in its predictions. This non-trivial fact is not explained any further in the original paper. Notoriously standard regression neural networks yield a point estimate, rather than a predictive distributions or confidence intervals. The most rational explanation that we were able to come up with is the authors used an ensemble of many randomly initialized MLPs with the same model architecture which are trained on the same training data (which also goes in line with one of the main proposals in the original BBB paper; to train an ensemble of neural nets). We ultimately trained 100 randomly initialized models for each of the two baseline models.

All MLPs in both ensemble models were trained until convergence, with all neural nets achieving approximately the same mean absolute error.

In the end, we would like our model to learn the underlying curve function (1), at least in the immediate vicinity of our training data, and in cases where the model prediction is wrong, we would like the model to at least be uncertain about its own prediction, so that we can infer that the predicted value has a higher likelihood of being false. Given the starting point of this experiment - where we trained the model only on x-values in the $[0.0, 0.5]$ interval - it is reasonable to expect the model to fail when predicting on data points outside of our training distribution. The reason being that one could hypothetically define any kind of function with the same data point distribution in the $[0.0, 0.5]$ interval, but with widely different behaviour outside of our training range. What is unreasonable though, and what the BBB paper is (among other things) trying to address, is the high confidence of standard artificial neural networks in their own prediction when the input data is (far) outside of the data distribution it has seen during

training.

This fact can be observed in Figure 9. The model has high confidence in the data range it was trained on ($[0.0, 0.5]$), but outside of it it is still very (over) confident in its false predictions: the predictions between 0.6 and 1.1 encompass not a single data point, yet the confidence interval stays very narrow, suggesting the ensemble is very confident in its prediction. As mentioned in the original BBB paper, dropout addresses some of the issues standard MLPs have - the confidence intervals become wider, but they still miss a lot of the ground-truth data outside of the training data range.

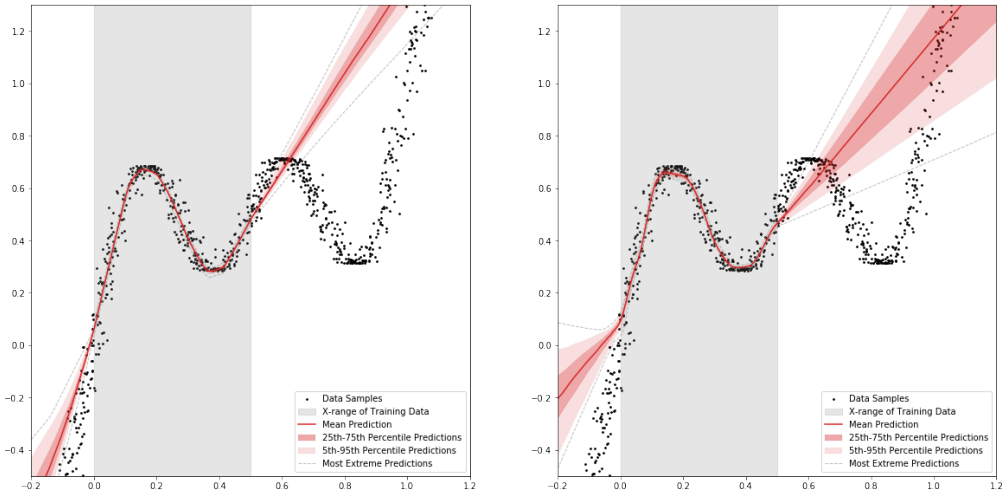


Figure 9. Median predictions and interquartile ranges for the predictions of two ensembles of 100 MLPs, without dropout (left) and with 50% dropout (right).

4 Bayesian Neural Nets : A first implementation

Lastly, we implemented a first baseline model based on the methods described by Blundel et al. in the original paper **Weight Uncertainty in Neural Networks** (2015). The paper addresses a central issue which artificial neural networks (ANNs) have been identified to have: namely, neural nets return point estimates, which may then be inspected for their accuracy, but they do not provide any information about the confidence they have in their own prediction - an issue which in sensitive domains like healthcare may be extremely dangerous. We thus would like an ANN to take a probabilistic approach to inference and reasoning, i.e. we would like to shift from Maximum Likelihood Estimation to a Bayesian Inference approach. Blundel et al. address this by treating the weights (and biases) of an ANN as data distributions, rather than single values, as is done classically. Purely working with distributions for ANNs becomes intractable very quickly, thus a surrogate method must be employed - this common issue is addressed by Variational Bayesian Inference techniques in machine learning.

In short, Variational Bayesian methods construct a simpler distribution which should be as close to the original intractable distribution as possible, by minimizing their respective KL-divergence. Using the resulting formula we can define a cost function which tries to find a prior close to the variational posterior, that at the same time explains the data well. This cost function is widely known as the Evidence Lower Bound (ELBO), and is referred to as the variational free energy in the original BBB paper.

Calculating the ELBO directly is once again computationally prohibitive for ANNs, which is why it needs to be approximated via repeated random sampling. For this, Blundel et al. define the so-called Gaussian Variational Posterior, which we implemented as a subclass of Pytorch's nn.Module superclass; the weight (or bias) is sampled from a "learned distribution" (a Gaussian where two scalar are learned: its mean and its variance).

As a prior, Blundel et al. propose the Scale Mixture Prior, which we implement as a subclass of Pytorch's nn.Module superclass. It consists of an affine combination of two Gaussians with the same mean; one with a heavy tail, the "slab", and the other with a high peek and low variance, the "spike". Formally: $\sigma_1 > \sigma_2$, $\sigma_2 \ll 1$.

Our "BayesianLinearLayer" class defines a linear layer which implements the BBB algorithm. For this, a ScaleMixturePrior object is initialized, the values ρ and μ are randomly initialized, which in turn initializes a GaussianVariationalPosterior object, using the ρ to calculate the variance and μ as the mean. The forward pass of the layer consists in sampling the weights and biases from their respective posterior distributions, which in turn are used to calculate the respective log-likelihoods for their priors and posteriors. The sum of the posterior log-likelihoods is subtracted from the sum of the prior log-likelihoods to calculate the KL-divergence between the distributions. The sampled weight and bias are passed on to a vanilla linear function together with the input to the layer, x. We would like to point out that this simple Bayesian Linear Layer does not yet implement the generalized Gaussian Reparameterisation Trick, as presented by Blundel et al..

Finally, the base Bayesian Neural Network (BNN) is defined. This PyTorch module defines the amount of hidden layers and neurons as well as the activation functions for a trainable model. It furthermore implements a function to sum up the KL-divergence terms of all bayesian linear layers in the model. The total KL-divergence is ultimately needed in the cost-function of the model: the ELBO. The self.elbo function specifies how many times to draw samples at each layer, and calculates the final loss of the model as π_i multiplied by the (previously calculated) total KL-divergence, plus the negative log-likelihood of the model prediction compared to the ground truth. π_i is a regularization term which is newly calculated at every single minibatch with index i, which is fed to the network. We have implemented this regularization function as the "minibatch_complexity_cost". Its ultimate purpose is to give more influence to the complexity loss of the first minibatches, while in later minibatches the learned parameters will be more influenced by the data. In short, this is done to keep down the influence of the data at the beginning of training, which may negatively impact the gradients of the yet fully randomly initialized model weights. Once again we would like to point out that this model builder class does not yet implement the generalized Gaussian Reparameterisation Trick.

The subsequent backpropagation and tweaking/learning of distribution parameters from which the weights and biases are drawn are taken care of by PyTorch's automatic differentiation engine.

For the following models and experiments we have partly reused the hyperparameters which Blundel et al. used for their MNIST classification, many other hyperparameters were not mentioned by the authors and we have defined ourselves. That being said, we have not conducted any hyperparameter search (e.g. grid search) in this first milestone of the project.

4.1 Classification

For the classification on the MNIST data set we define a BNN object with 784 input units, 10 output units and 2 hidden layers of 1200 units each. The biggest difference in training a BNN for multi-class classification as opposed to regression is the function used for calculating the negative log-likelihood: we use PyTorch's CrossEntropyLoss with sum-reduction. Using the Adam optimization algorithm we train our first proof-of-concept model for 8 epochs, resulting in an encouraging classification accuracy of 83.1% (see Figure 10.), without having done any hyperparameter optimization.

```
Training of a 2 hidden layers BNN for MNIST
epoch : 1, train loss : 2335209509.2271, test error : 28.1 %
epoch : 2, train loss : 2227849327.9331, test error : 20.1 %
epoch : 3, train loss : 2027745356.0058, test error : 21.5 %
epoch : 4, train loss : 1864104378.5453, test error : 21.4 %
epoch : 5, train loss : 1740842652.2081, test error : 18.2 %
epoch : 6, train loss : 1650773325.1786, test error : 20.6 %
epoch : 7, train loss : 1584624372.7423, test error : 21.3 %
epoch : 8, train loss : 1534337922.8213, test error : 16.9 %
```

Figure 10. Train loss and test error outputs for our first classification problem after 8 epochs of training.

4.2 Regression

For the regression on the "curve" data we define a BNN object with input and output layers with respectively 1 unit, as well as 2 hidden layers of 400 hidden units each. Here, for the negative log-likelihood function, we define a Gaussian with the model output $\hat{y}$ for μ, and a separate noise parameter σ_{noise} for its variance, from which the log-likelihood is calculated with respect to the ground truth y. This is our interpretation of the "conditional Gaussian loss" which Blundel et al. mentioned to have used for their regression experiments. The noise parameter σ_{noise} is a new hyperparameter which would need to be tuned in its own right. In this case we know the curve data generating function (1) draws its noise ϵ from a Gaussian with variance 0.02, which we reuse for the σ_{noise} parameter.

After 1000 training iterations using the Adam optimizer we receive encouraging results for the regression task as well (see Figure 11.). While the predictive median in the training data range leaves much to be desired, we receive much wider confidence intervals than with vanilla MLPs. Consequently much more datapoints are contained in the resulting "confidence area".

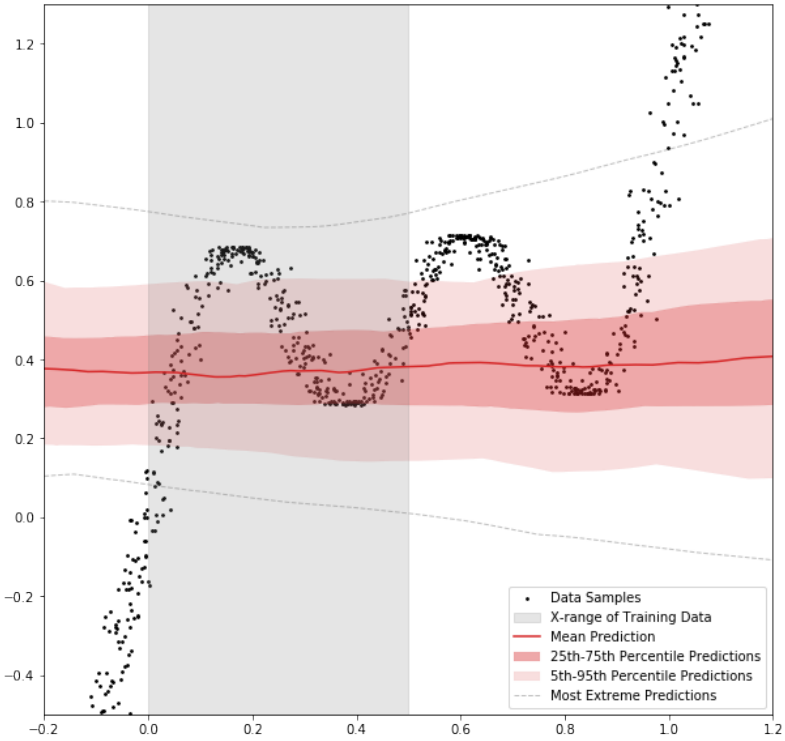


Figure 11. Median prediction and interquartile ranges for the predictions after sampling 100 individual neural nets from our first trained regression-BNN

5 Conclusion and Outlook

In Milestone 1, we were able to implement the baseline comparative models from the original paper, **Weight Uncertainty in Neural Networks**, and to (roughly) reproduce the baseline results by the authors. We were also able to implement a first version of the Bayes By Backprop algorithm and its associated Bayes linear layer. This provides a foundation on which we can build upon in Milestone 2; to finish implementing the BBB-algorithm, namely the Gaussian Reparameterization Trick part of it, as well as to do some larger-scale hyperparameter optimization, in order to receive similar experimental results as per the original BBB paper. Regarding regression, we were able to observe the overconfidence vanilla NNs have when predicting on data far from the distribution of the training data, and how dropout begins to address that issue, which is then further amplified when using a Bayesian Neural Net.

5.1 Classification

For classification, as per the original paper by Blundel et al., we would ultimately expect the BBB-algorithm (light orange line in Figure 12.) to yield a higher accuracy than vanilla MLPs with and without dropout. Secondly, we expect the fully-implemented and optimized BBB model to showcase a larger variance in the distribution of the weights, as is shown in Figure 12.

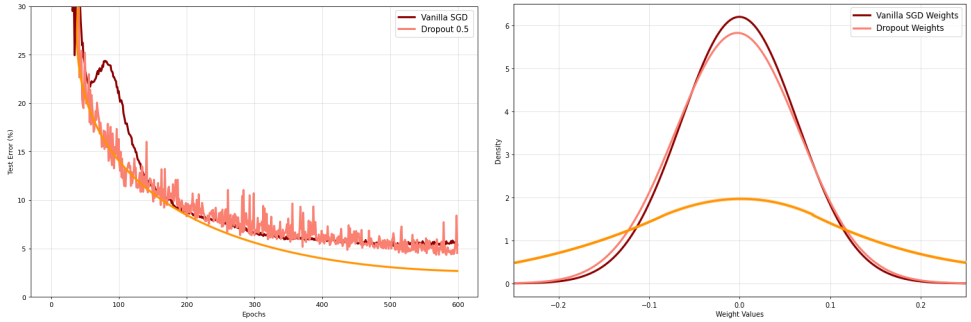


Figure 12. Forecasted results for the BBB-algorithm

5.2 Regression

Applying the BBB-algorithm to our regression task we expect a far wider confidence interval in data ranges which were not covered during the training process of our neural networks, contrary to what we had observed in Figure 9. We would like the 25-75th percentile predictions to cover the majority of data points outside our training set, but we would also like the confidence interval inside the training range $[0.0, 0.5]$ to reflect the variation induced by the additive noise ϵ. The hand-drawn example in Figure 13 reflects what we would ideally like to see at the end of Milestone 2, after implementing the BBB algorithm completely, and optimizing it for the regression problem at hand.

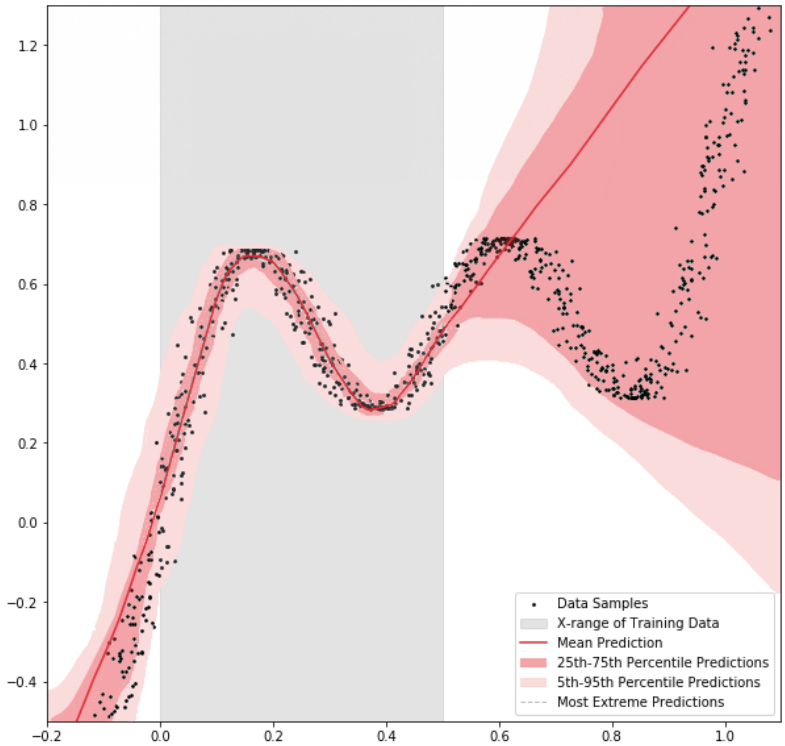


Figure 13. Something we would ideally like to observe when optimizing our training training procedure for the regression task at hand (hand-drawn example)