Replicating the model comparison figure of Implicit Weight Uncertainty in Neural Networks

Mozafar Shah (4717368, S.M.Shah-2@student.tudelft.nl), Siert Sebus (4710304, s.j.g.sebus@student

April 14, 2022

1 Introduction

Nowadays, neural networks are a popular solution to a wide array of machine learning problems such as classification, regression and feature mapping. A drawback of neural networks, however, is that, while they can often provide good performance if tuned correctly, they tend to be poor at assessing the confidence of their predictions. In fact, they are unreasonably overconfident most of the time, even on data points very dissimilar to those found in the training set. An example of this can be seen in Figure 1, where a simple convolutional network starts predicting other numbers with high confidence, the moment it can no longer identify the rotated three.

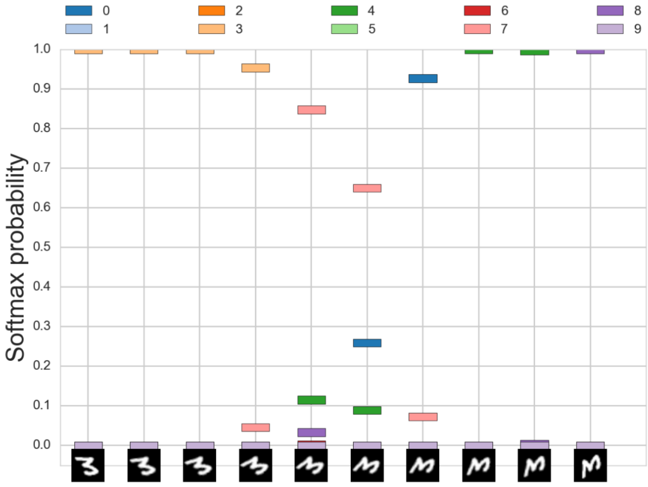


Figure 1: A figure from Louizos & Welling (2017) in which prediction certainty by a simple 'LeNet' architecture is shown on a progressively more rotated handwritten three from the MNIST data set.

To solve this problem, special neural network architectures have been developed that give a more reasonable indication of prediction uncertainty. An example of this is the Bayes by Hypernet architecture introduced by Pawlowski et al. (2017). In this architecture, the weights of the neural network do not take on a single value, but each have a small distribution describing the probabilities of the different values that the weight can take on. This way, a Bayes by Hypernet network does not encode a single parameterization of a network but a distribution over parameterizations. Running the network with different sampled weights then allows a variance estimate to be made on each of the outputs, giving a more reliable uncertainty measure.

1.1 Objective

The objective of this study is to replicate a specific figure from the Bayes by Hypernet paper (Pawlowski et al., 2017). This figure is also included here as Figure 2. In the figure it is shown how a toy regression problem is solved by six different machine learning implementations, one of which is Bayes by Hypernet. Importantly, for each of the methods the uncertainty is visualized.

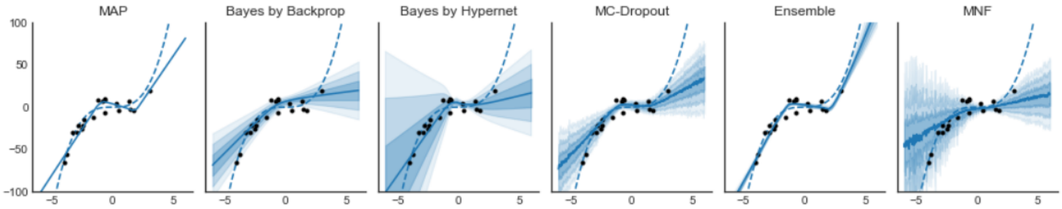


Figure 2: A figure from Pawlowski et al. (2017) in which six different models solve the same toy regression problem. The data points of this problem are represented by the black dots and are sampled from a real function, represented by the dashed line, with added noise. The blue dashed line represents the true data generating function. The continuous blue line denotes the regression fit made by the respective model. The partially transparent blue regions denote the model's predictive uncertainty. Specifically, the three different shades of blue denote 1, 2, and 3 standard deviations from the mean.

The original purpose in the paper of the image was to give a comparison between the newly introduced Bayes by Hypernet architecture and other, already existing architectures. To generate the figures, the authors of the Bayes by Hypernet paper implemented the various architectures themselves using the TensorFlow framework. We replicated those implementations in another deep learning framework, PyTorch, with the goal of reproducing four of the six figures in Figure 2. We did not implement Multiplicative Normalizing Flows (MNF) as it turned out to be outside the scope of the project and Maximum a Posteriori (MAP) was deemed not interesting enough.

2 The toy regression problem

The data set, that was used for the training and evaluation of the predictions of each model, is the same toy data set that was used in the paper Pawlowski et al. (2017). In this paper the toy data set was inspired by Hernandez-Lobato & Adams (2015). In order to accurately replicate the paper by Pawlowski et al. (2017) the same data set was used. Formally, the data $\mathcal{D}$ consists of $n = 20$ data points (x, y) with a feature space $x \in \mathbb{R}$ and a target space $y \in \mathbb{R}$. The targets are sampled from the normal distribution $\mathcal{N}(x^3, 9)$. So, effectively, the targets are values from the function $f(x) = x^3$ with added Gaussian noise of variance 9.

The following models have been selected to evaluate and compare their predictions in a regression experiment based on the toy data set. Bayes by Hypernet, Bayes by Backprop, MCDropout, and Ensemble. The implementation of these models will be discussed in sections 3, 4, 5, and 6, respectively.

3 Bayes by Backprop

Bayes by Backprop (BbB) is a neural network architecture first introduced by Blundell et al. (2015) that tries to approximate Bayesian inference over neural networks. Bayesian inference for neural networks calculates the posterior distribution of the network weights given the training data, $P(\theta|\mathcal{D})$. This distribution can then be used to gain an accurate distribution of target values given a data point, $P(\hat{y}|\hat{x}) = \mathbb{E}_{P(\theta|\mathcal{D})}[P(\hat{y}|\hat{x}, \theta)]$, and thus an accurate uncertainty estimate.

Since calculating $P(\theta|\mathcal{D})$ is intractable for networks of any practical size, Bayes by Backprop approximates it by constructing a new type of network in which every weight does not have a single value but describes a distribution of weight values. This is visualized in Figure 3.

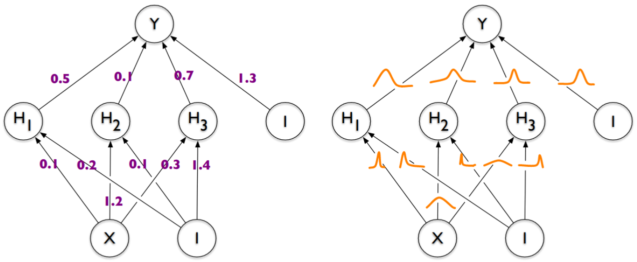


Figure 3: This is a figure by Blundell et al. (2015) that visualizes the concept of defining every weight in a network by a distribution of values instead of a single value.

In BbB, those weight distributions are assumed to be Gaussians. This assumption has the benefit that each weight distribution $\theta^{(i)}$ can be described by only two values, a mean $\theta_{\mu}^{(i)}$ and a variance $\theta_{\sigma^2}^{(i)}$, which are again just scalars. Effectively, a BbB implementation then only requires twice the amount parameters compared to a regular neural network of similar size.

To use a BbB for uncertainty estimation, one simply generates a number of networks from the BbB network by sampling a value for each weight. Then, uncertainty can be determined by calculating the empirical variance over the network outputs.

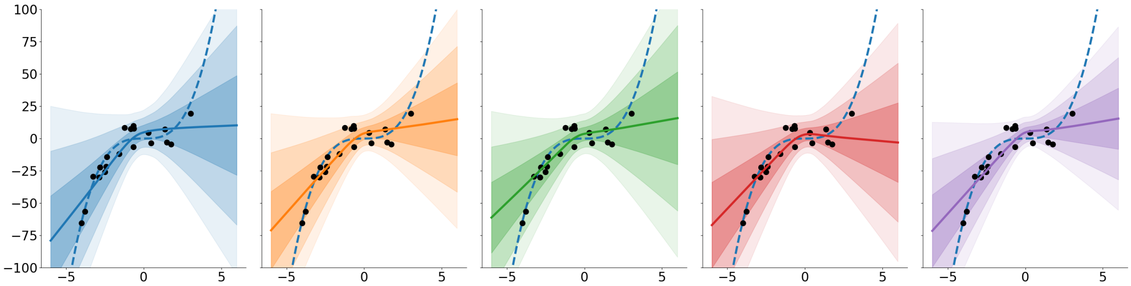


Figure 4: Results of running our BbB implementation with 5 different seedings for network initialization.

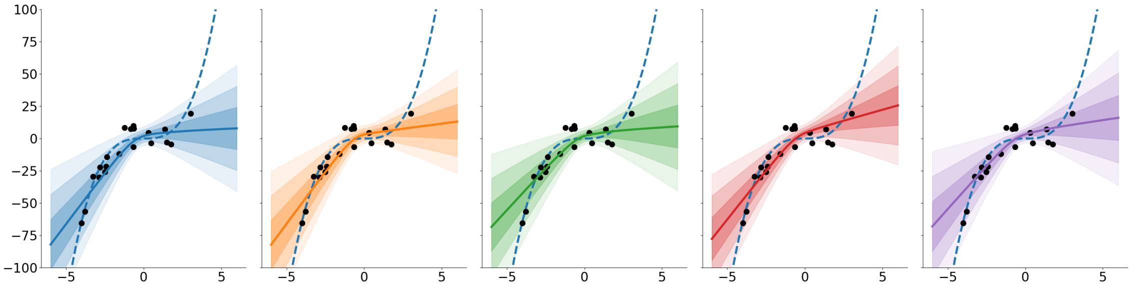


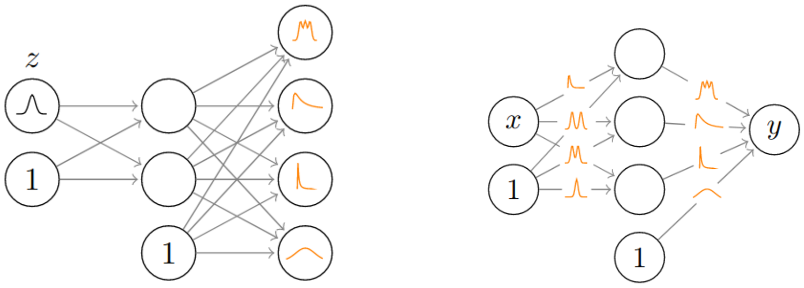
Figure 5: Results of running the BbB implementation of Pawlowski et al. (2017) with 5 different seedings for network initialization.

When comparing the plots generated from our implementation of BbB (Figure 4) and those from Pawlowski et al. (2017)'s implementation (Figure 5), it is immediately clear that the uncertainty is a lot higher in our implementation, across all seedings, compared to theirs. We are not sure why this is the case. It is likely due to a different interpretation of the BbB algorithm as we constructed our model mostly based on the original BbB paper by Blundell et al. (2015) and not so much on the TensorFlow implementation by Pawlowski et al. (2017).

Otherwise, the shape of the uncertainty and the regression mean are comparable. Both implementations of BbB are poor at getting a good fit for the data, compared to other methods, and both implementations give a good uncertainty estimate. For both models, the center of the feature space, where there is a lot of data, is a lot more certain than the sides, where there is little data.

4 Bayes by Hypernet

Bayes by Hypernet (BbH) is very similar to BbB in the sense that it also tries to approximate the intractable distribution $P(\theta|\mathcal{D})$. It does not however make the same assumption that weights are normally distributed. Instead, it aims to be more general by allowing any distribution over the weights. It achieves this by using a HyperNetwork as introduced by Ha et al. (2016) to estimate $P(\theta|\mathcal{D})$. A HyperNetwork is a neural network that outputs the weights of another network. In this case, a HyperNetwork is trained such that it takes as its input a vector of random values z and outputs network weights as if they were sampled from the posterior $P(\theta|\mathcal{D})$. This is illustrated in Figure 6. By running this network on different samples z, many different networks can be sampled through which the uncertainty defined by the posterior can be approximated by, again, calculating the empirical variance over the outputs. The fact that the weights are sampled from another neural network allows, unlike BbB, the distributions over to be any shape. This should, in theory, allow BbH to make more accurate uncertainty estimates.



(a) The HyperNetwork generating weights based on a random input vector z.

(b) The main network using generated weights to generate a target value y from an input x.

Figure 6: A simple visualization of how BbH works borrowed from Pawlowski et al. (2017)

When trying to replicate BbH in PyTorch, we noticed that the author of the paper had already published a PyTorch implementation of Bayes By Hypernet online on GitHub with a corresponding notebook containing an experiment on the same toy example.

Assuming the implementation was working correctly, instead of focusing on implementing our own BbH model, we focused on testing the approaches for multiple runs, and their consistency across different seedings. However, when performing the empirical tests for Bayes By Hypernet, comparing the pytorch implementation with the one in Tensorflow, we noticed huge discrepancies between them. We assume that this is most likely caused by bugs in the Pytorch implementaton, for which some of them are already fixed by us at the ending stage of the project.

With the addition of a new component, Hypernet, in the architecture, with it comes a lot of hyperparameters to tweak, and design choices to make. Similar to the Bayes By Backprop approach described in section 3, it samples from a noise distribution, for which its dimensionality can affect the fitting and predictive uncertainty of trained models. To demonstrate this, we performed experiments using a singleton dimension, and one with 200.

However, these are not the only considerations to take into account, since we are using a separate hypernetwork for each weight matrix, and one for each bias, we can also let them use an individually sampled noise vector from the distribution, further reducing the ... In addition, when calculating the KL divergence between the distribution generating the weights, and the prior, (is dit goed???), we can also decide to calculate the divergence for each dimension in the weight individually, instead of the entire weight space. Similarly, the KL divergence is an approximation, and the amount of weight vectors and prior instances we sample from their respective distribution, is also a factor to take into consideration.

5 MCDropout

Monte Carlo Dropout, MC Dropout, or MCD is used as a regularisation technique, i.e. a technique used to prevent overfitting. Regularisation is especially needed in small datasets such as the toy dataset that is being used, as the network is more likely to memorise the training data and consequentially provide excellent predictions solely on the data instances it has encountered during training. Dropout is one technique that helps avoid this problem. In order for dropout to be applied dropout layers have to be defined in the network. Dropout essentially ”turns off” some neurons in the dropout layers at each step in the training. Each neuron has some probability p (the dropout rate) of being ignored, this dropout rate is usually set between 0 and 0.5. When the dropout rate is set to 0 that means there is no dropout applied and when it is set to 0.5 each neutron in the dropout layer has a probability of 0.5 of being dropped. Since this process is probabilistic each training iteration another subset of the neutrons in the dropout layer(s) will be dropped. As a result the architecture of the model varies slightly. The outcome can then be seen as a sort of average of several networks.

In MCD, besides applying dropout during training, dropout is also applied at test time. This results in getting various predictions instead of a single prediction. The resulting distribution can then be analysed as wanted.

This subsection will provide the result of running our implementation in comparison to the implementation of Pawlowski et al. (2017).

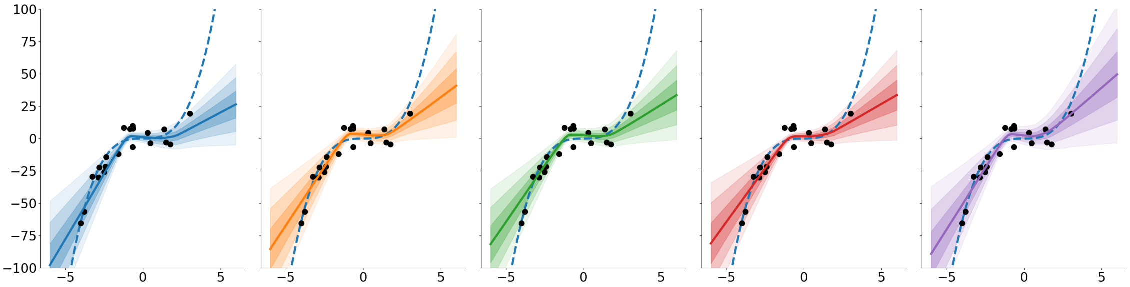


Figure 7: Results of running the MCD implementation of Pawlowski et al. (2017) with 5 different seeding points.

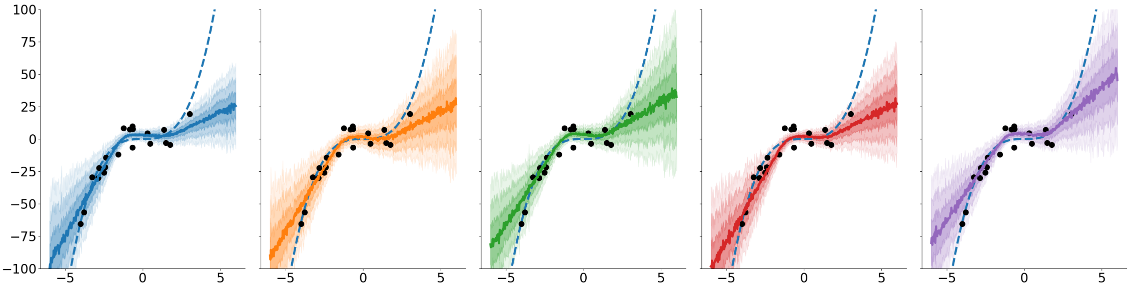


Figure 8: Results of running our MCD implementation with 5 different seeding points.

The resulting graphs look quite different. The graphs of the method by Pawlowski et al. (2017) look a lot smoother and have a lot less stochastic-seeming lines than our implementation. Both the uncertainty estimations and the resulting regression fits in our implementation seem to be more stochastic than the implementation of the paper. However, looking at figure 1 in the paper by Pawlowski et al. (2017), which is figure 2 in this document. The sub-figure corresponding to MC-Dropout included in that figure also has such stochastic-seeming lines for both the regression fit and the uncertainty estimations. Assuming that the authors chose to include the more stochastic-seeming graph because it is supposed to look like that. We believe that there must be some error in the implementation of MC-Dropout in the paper by Pawlowski et al. (2017) or the way we ran their implementation was erroneous, as the resulting graphs look very smooth in figure 7.

In both figures the graphs for the different seeding points look quite similar. However, some small differences can be seen with regards to both the uncertainty estimates and the regression fits, e.g. when looking at the leftmost graph and the rightmost graph. Thus, the seeding points do not play a significant role in the resulting regression fit and uncertainty estimates for this method.

6 Ensemble

Uncertainty estimation through an ensemble of neural networks as described by Lakshminarayanan et al. (2016) is a relatively simple method. One simply trains a number of differently initialized networks, in our case that number is 10, and then estimate uncertainty based on how much the networks agree or disagree with each other on the predicted target value for a particular data point. This uncertainty estimate is done by calculating the empirical variance over the different network outputs for each data point.

The results of running the ensemble can be seen in Figure 9. Compared to the other discussed methods, the ensemble underestimates the uncertainty. This is both the case with our implementation and the implementation of Pawlowski et al. (2017). To make it easier to compare the two implementations, we generated new graphs with boosted variance, namely, standard deviations of 5, 10 and 15 instead of 1, 2 and 3. These can be found in Figure 10 and Figure 11.

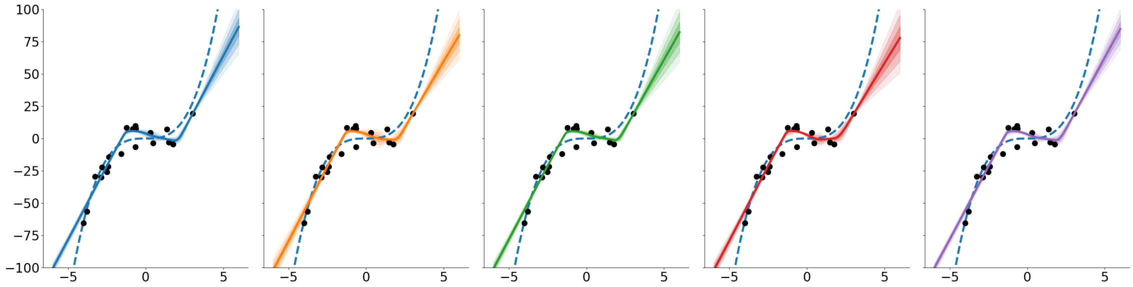


Figure 9: Results of running our ensemble implementation with 5 different seedings for network initialization. The transparent colored regions correspond to 1, 2 and 3 standard deviations.

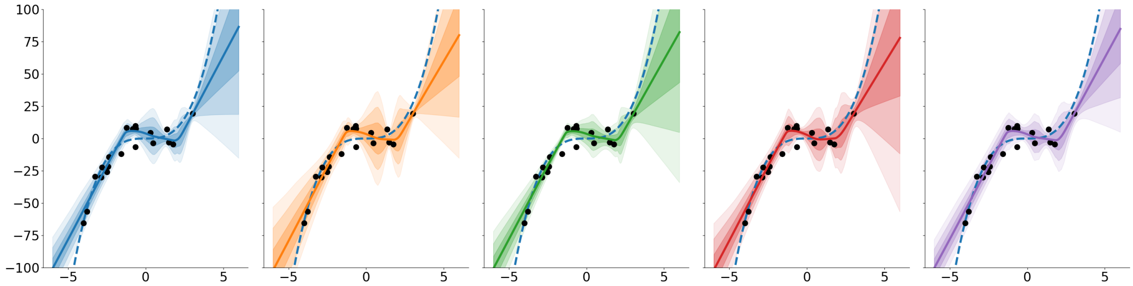


Figure 10: Results of running our ensemble implementation. The transparent colored regions correspond to 5, 10 and 15 standard deviations.

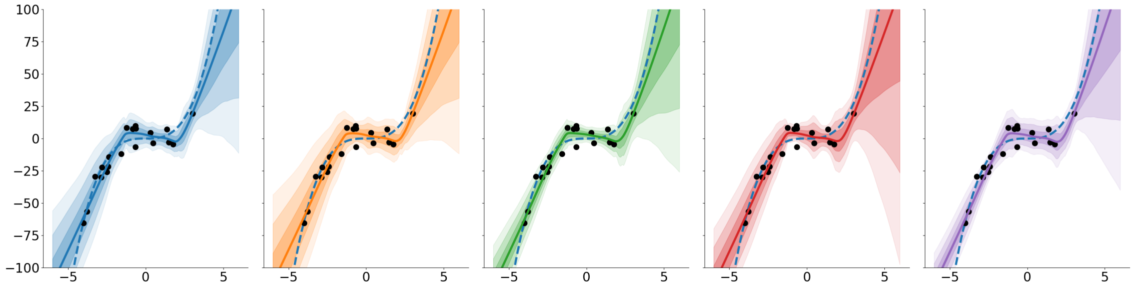


Figure 11: Results of running Pawlowski et al. (2017)'s ensemble implementation. The transparent colored regions correspond to 5, 10 and 15 standard deviations.

Both implementations seem to be resilient enough against changing the seeding. The regression mean stays roughly in the same place and follows the true distribution quite nicely. The variance, however, does fluctuate a lot across different seedings, especially for values outside the data set. The variance also appears to be slightly more pronounced in the paper's implementation compared to ours.

The largest difference between the two implementations seems to be smoothness of the variance. In our implementation, the variance is more sporadic in the center and straighter for values outside the bounds of the data set. We don't know for sure what causes this difference but our suspicion is that it has to do with the 'adversarial training for smooth predictive distributions' as described by Lakshminarayanan et al. (2016). Both our model and Pawlowski et al. (2017)'s model implemented this feature into the ensemble and there is probably a slight difference in implementation that explains the difference in smoothness.

7 Discussion

For the most part, we managed to reproduce all results within acceptable variability. The only reproduction that is a bit doubtful is the one of Bayes by Hypernet. This is the architecture for which we did not build our own model but for which we modified the PyTorch model provided by the authors and ran it against their Tensorflow model. Not only did we have problems aligning the outputs of the two different implementations, we also did not manage to reproduce the plot in the original image with either implementation. The following figures provide an overview of the various models for both the tensorflow implementation of the authors and our pytorch implementations of the corresponding methods. As can be seen the results for the respective methods look quite similar, with the exception of Bayes by Hypernet.

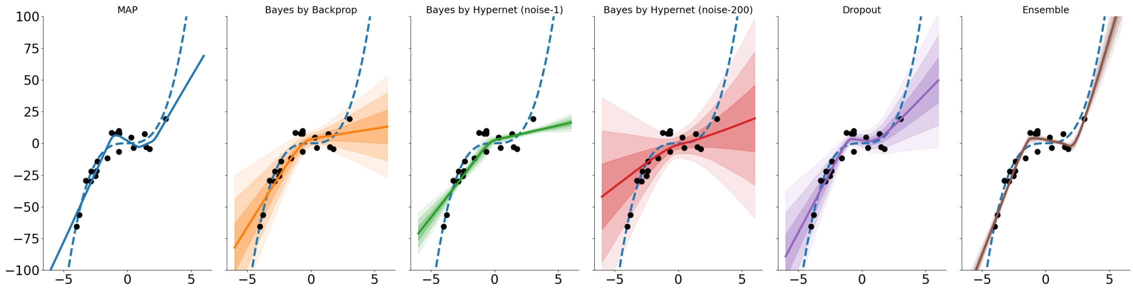


Figure 12: Overview of the models using the tensorflow implementations of the paper by Pawlowski et al. (2017).

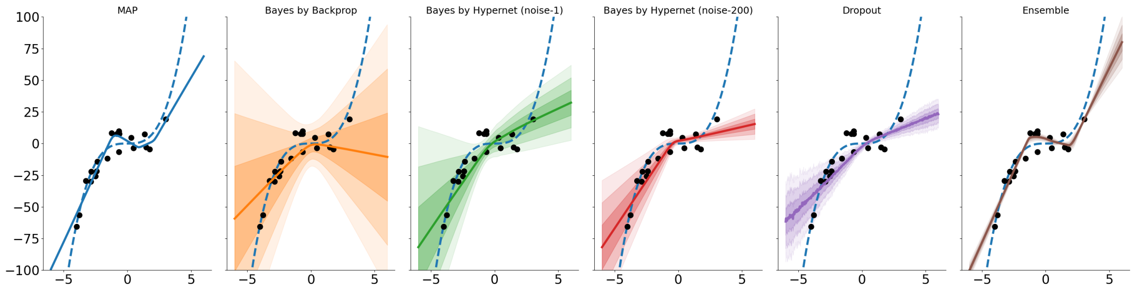


Figure 13: Overview of the models using our pytorch implementations of the models (with the exception of Bayes by Hypernet.

Also, the result for the MCDropout method seems a bit off. However, when looking at the results in section 5 the results for the different implementations are quite similar. The differences between the implementations for Bayes by Backprop have been highlighted in section 3.

8 Conclusion

All in all, the results for most methods were reproducible with acceptable differences. However, the method introduced by the authors of the paper, Bayes by Hypernet, is the only method for which we did not achieve a similar result as the paper. Therefore, we would argue that the paper is not clear enough to be properly reproduced or their pytorch implementation of the introduced method is faulty. Possible solutions could be to adjust their current pytorch solution for their model. Another option would have been for us to try and implement this introduced method from scratch like we did with the other methods, but we did not expect any errors in the implementation until we ran the comparisons of all models and by that time it was too late to implement the entire introduced model from scratch.

9 Task Overview

We wrote the report/ blog together and the individual tasks of each group member are listed in the table below.

Group member	**Tasks**
Chris	Code structuring and integration of implemented methods to generate plots
Mozafar	MCDropout and MAP implementation
Siert	Bayes by Backprop and Ensemble implementation

Table 1: Task overview per team member

References

Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015). Weight uncertainty in neural networks. In *Proceedings of the 32nd international conference on international conference on machine learning - volume 37* (p. 1613–1622). JMLR.org.

Ha, D., Dai, A., & Le, Q. V. (2016). *Hypernetworks.* arXiv. Retrieved from https://arxiv.org/abs/1609.09106 doi: 10.48550/ARXIV.1609.09106

Hernandez-Lobato, J. M., & Adams, R. (2015, 07–09 Jul). Probabilistic backpropagation for scalable learning of bayesian neural networks. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (Vol. 37, pp. 1861–1869). Lille, France: PMLR. Retrieved from https://proceedings.mlr.press/v37/hernandez-lobatoc15.html

Lakshminarayanan, B., Pritzel, A., & Blundell, C. (2016). *Simple and scalable predictive uncertainty estimation using deep ensembles.* arXiv. Retrieved from https://arxiv.org/abs/1612.01474 doi: 10.48550/ARXIV.1612.01474

Louizos, C., & Welling, M. (2017). Multiplicative normalizing flows for variational bayesian neural networks. In *Proceedings of the 34th international conference on machine learning - volume 70* (p. 2218–2227). JMLR.org.

Pawlowski, N., Rajchl, M., & Glocker, B. (2017). Implicit weight uncertainty in neural networks. *ArXiv*, *abs/1711.01297*.