Weight Uncertainty in Neural Networks

Assessment of Reproducibility

Sam Clarke

1 Introduction

This report assesses the reproducibility of the machine learning paper 'Weight Uncertainty in Neural Networks' [1]. The key contribution of the paper is the Bayes by Backprop algorithm (BBB), which the authors claim alleviates two common defects of plain feedforward neural networks: overfitting and their inability to assess the uncertainty of their predictions. Furthermore, they claim BBB provides a solution to the exploration vs. exploitation trade-off in reinforcement learning. The authors give theoretical and empirical evidence for these three claims. This report can be framed as an assessment of the reproducibility of that empirical evidence, conditional on a certain amount of compute, time and development effort.

This report is organised as follows: Section 2 gives our rationale for selecting this paper and the particular experiments. Section 3 summarises aspects of the paper relevant to our work. Section 4 describes the experiments we tried to replicate, compares them to those in the original paper, and draws conclusions about their reproducibility. Section 5 concludes, with a critical evaluation and suggestion of further work.

Contents

2 Rationale for selecting paper and experiments

We chose to replicate the paper 'Weight Uncertainty in Neural Networks' (WUNN) for three reasons. Firstly, group members were more curious and excited about Bayesian Deep Learning (BDL) than Natural Language Processing. Secondly, our interests are more theoretical than applied. Thirdly, we felt not to yet have a rigorous understanding of BDL, suggesting that replicating a seminal BDL paper would be more appropriate than one that goes beyond the basic methodology. Therefore, within the possible BDL papers, we were drawn to WUNN due to its theoretical nature and its being one of the first papers to explore a basic BDL technique.

As for the particular experiments, we felt it necessary to thoroughly all three experiments in the paper, including an attempted hyperparameter search, because we borrowed some of our code from online sources (see Section 6 for details). For the same reason, we extended the paper by (i) comparing BBB to an alternative method of Bayesian approximation for neural networks, Dropout (see Section 4.2), and (ii) exploring the idea that BBB can be used to guide decisions on which additional data to request in an Active Learning setting (see Section 4.4). Note that our group had only 3 members.

3 Summary of aspects of paper relevant to the experiment

The original paper is a terse eight pages. We summarise Section 3 (Being Bayesian by Backpropagation) which presents the key theoretical contribution of the paper. We omit a summary of Section 1 (Introduction) and Section 2 (a summary of standard learning in neural networks). Section 5 gives the experimental results, which are discussed in conjunction with our experimental results below. Section 6 is a short discussion with no new key information. Section 4 contextualises the application of BBB to a reinforcement learning problem, and Section 3.4 describes how BBB is amenable to minibatching. These are not directly relevant to the experiments, and so their summaries are deferred to the Appendix.

3.1 Being Bayesian by Backpropagation

Typically, when we train a neural network, it learns single point estimates for each of its weights, to minimise some loss function, given some data. To do bayesian inference on a neural network, we calculate the posterior distribution of the weights given the training data, $P(w|D)$. Figure 1 illustrates the difference.

Then, given the posterior distribution over weights, to answer predictive queries we take expectations under this distribution:

$$P(\hat{y} \mid \hat{x}) = \mathbb{E}_{P(w|D)}[P(\hat{y} \mid \hat{x}, w)]$$

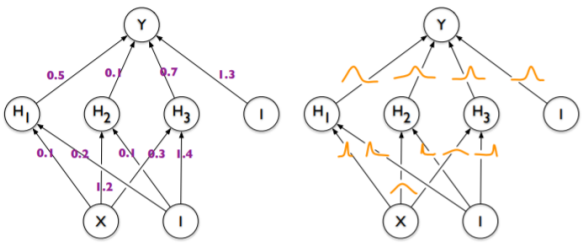

Figure 1: Left: learning point estimates of weights in a neural network. Right: learning distributions over weights.

However, this proper 'dogmatic' Bayes approach is intractable: taking an expectation under the posterior distribution on weights effectively means computing a forward pass with an infinite ensemble of neural networks, each with one possible configuration of the weights, then weighting their outputs according to the posterior distribution.

When we cannot do exact Bayesian inference, there are two common approximation methods: Markov Chain Monte Carlo and Variational Inference (VI). The paper proposes to use the latter.

VI works by approximating the true Bayesian posterior distribution $P(w|D)$ with another distribution $q(w|\theta)$. Its parameters θ are then optimised to minimise the Kullback-Leibler (KL) divergence with the true posterior:

$$\begin{aligned}
\theta^* &= \arg\min_\theta \mathrm{KL}\left[\, q(\mathbf{w}|\theta) || \, P(\mathbf{w}|\mathcal{D})\right] \\
&= \arg\min_\theta \int q(\mathbf{w}|\theta) \log \frac{q(\mathbf{w}|\theta)}{P(\mathbf{w}|\mathcal{D})} \\
&= \arg\min_\theta \int q(\mathbf{w}|\theta) \log \frac{q(\mathbf{w}|\theta)}{P(\mathbf{w})P(\mathcal{D}|\mathbf{w})} \\
&= \arg\min_\theta \mathrm{KL}\left[\, q(\mathbf{w}|\theta) || \, P(\mathbf{w})\right] - \mathbb{E}_{q(\mathbf{w}|\theta)}[\log P(\mathcal{D}|\mathbf{w})]
\end{aligned}$$

From this, we get the following loss function, $\mathcal{F}(\mathcal{D}, \theta)$, known as the variational free energy [2, 3, 4] or the expected lower bound [2, 5, 6]. $-\mathcal{F}$ is referred to as the evidence lower bound (ELBO). It is helpful to view it in two forms:

$$\begin{aligned}
\mathcal{F}(\mathcal{D}, \theta) &= \mathrm{KL}\left[\, q(\mathbf{w}|\theta) || \, P(\mathbf{w})\right] - \mathbb{E}_{q(\mathbf{w}|\theta)}[\log P(\mathcal{D}|\mathbf{w})] &(1)\\
&= \mathbb{E}_{q(\mathbf{w}|\theta)}[\log q(\mathbf{w}|\theta) - \log P(\mathbf{w}) - \log P(\mathcal{D}|\mathbf{w})] &(2)
\end{aligned}$$

Equation 1 shows that the loss function is the sum of a prior-dependent part, called the complexity cost, and a data-dependent part, called the likelihood cost. This elegantly captures the trade-off between fitting the data $\mathcal{D}$ and remaining close to a simplicity prior $P(\mathbf{w})$.

Minimising this loss analytically is computationally prohibitive. Equation 2 suggests an approximation to deal with this: estimate the expectation by sampling from $q(\mathbf{w}|\theta)$. We now explore how to combine this with gradient descent to get BBB, the key contribution of this paper.

Aside: Implementation

Before formally presenting BBB, it is worth noting that we already know enough to make a simple implementation! It remains to define a neural network architecture and an appropriate loss function, choose a type of probability distribution for the variational posterior (parameterised by θ) and the prior, find some data, specify an optimizer, and initialise parameters and hyperparameters.

The algorithm then proceeds by sampling from the variational posterior (possibly several times), computing a forward pass, and backpropagating the loss through the model parameters. This procedure is made precise in Algorithm 1.

Algorithm 1 Bayes by Backprop, with enough detail to translate into PyTorch or TensorFlow

1: **for all** epochs **do**
2: $\quad \mathcal{F} \leftarrow 0$
3: $\quad$ **for** $i \in \{1..n\}$ **do**
4: $\quad\quad$ Sample $\mathbf{w} \sim q(\mathbf{w}|\theta)$
5: $\quad\quad$ Compute $f = \log q(\mathbf{w}|\theta) - \log P(\mathbf{w}) - \log P(\mathcal{D}|\mathbf{w})$
6: $\quad\quad \mathcal{F} \leftarrow \mathcal{F} + f$
7: $\quad$ **end for**
8: $\quad \mathcal{F} \leftarrow \mathcal{F}/n$
9: $\quad$ Backpropagate $\mathcal{F}$ through the model parameters θ
10: **end for**

On line 5, we compute $q(\mathbf{w}|\theta)$ and $P(\mathbf{w})$ by simply plugging the sample of $\mathbf{w}$ back into $q(.)$ and into the prior density function, respectively. $P(\mathcal{D}|\mathbf{w})$ is the likelihood of the data given the model, which we compute (as in a typical neural network) by doing a forward pass, for all the training data, and accumulating the total loss according to the loss function we defined (e.g. cross entropy for categorical data; squared loss for real data).

This illustrates the remarkable simplicity of learning a distribution, as opposed to just point estimates, over weights: all that is necessary is to do several forward passes, one for each sample from the current distribution over the weights, use a slightly modified loss function (with just one extra term, assuming we would use a regularising prior anyway), and average it over the samples. Furthermore, assuming the θ which parameterised our variational posterior has only two parameters per weight, we only need to store twice as many parameters.

3.1.1 Unbiased Monte Carlo Gradients

This section of the paper digs deeper into the loss function approximation. In particular, it justifies why the derivative of an expectation can, in this case, be expressed as the expectation of a derivative. This is good news, because one might wonder whether Algorithm 1 is really justified. Is it really possible to get a good enough approximation of $\mathcal{F}$ using a reasonable number n of samples? How could we know our approximation is good enough? What this section shows is that in the case $n = 1$, the gradients computed using Algorithm 1 are equal in expectation to gradient of the true loss function[1]. Training the network involves many gradient updates, so we can then expected the cumulative effect of all these gradient updates to closely approximate updates based on the true loss function.

To justify this, the authors prove Proposition 1. The basic idea is a generalisation of the Gaussian reparameterisation trick [7, 8, 9] used for learning latent variable models. We cannot immediately push the derivative with respect to θ inside of the expectation because the expectation is over $\mathbf{w}$ which depends on θ. Therefore, $\mathbf{w}$ is reparameterised to depend on noiseless parameters θ, plus some random noise ϵ. The randomness now comes only from the ϵ term, so we can take the expectation over just ϵ, and therefore push the derivative inside since θ is now independent of the variable over which we take expectations.

[1]The authors do not mention why one might then use $n > 1$, as they do for some experiments, but presumably this will just further decrease the variance of the gradient estimates.

Proposition 1. *Let $\boldsymbol{\epsilon}$ be a random variable with probability density given by $q(\boldsymbol{\epsilon})$ and let $\mathbf{w} = t(\theta, \boldsymbol{\epsilon})$ where $t(.)$ is a deterministic function. Suppose further that the marginal probability density of $\mathbf{w}$, $q(\mathbf{w}|\theta)$, is such that $q(\boldsymbol{\epsilon})d\boldsymbol{\epsilon} = q(\mathbf{w}|\theta)d\mathbf{w}$. Then for a function f with derivates in $\mathbf{w}$:*

$$\frac{\partial}{\partial\theta}\mathbb{E}_{q(\mathbf{w}|\theta)}[f(\mathbf{w},\theta)] = \mathbb{E}_{q(\boldsymbol{\epsilon})}\left[\frac{\partial f(\mathbf{w},\theta)}{\partial \mathbf{w}}\frac{\partial \mathbf{w}}{\partial\theta} + \frac{\partial f(\mathbf{w},\theta)}{\partial\theta}\right]$$

Proof.

$$\begin{aligned}
\frac{\partial}{\partial\theta}\mathbb{E}_{q(\mathbf{w}|\theta)}[f(\mathbf{w},\theta)] &= \frac{\partial}{\partial\theta}\int f(\mathbf{w},\theta)q(\mathbf{w}|\theta)d\mathbf{w} \\
&= \frac{\partial}{\partial\theta}\int f(\mathbf{w},\theta)q(\boldsymbol{\epsilon})d\boldsymbol{\epsilon} \\
&= \int q(\boldsymbol{\epsilon})\frac{\partial}{\partial\theta}f(\mathbf{w},\theta)d\boldsymbol{\epsilon} \\
&= \mathbb{E}_{q(\boldsymbol{\epsilon})}\left[\frac{\partial}{\partial\theta}f(\mathbf{w},\theta)\right] \\
&= \mathbb{E}_{q(\boldsymbol{\epsilon})}\left[\frac{\partial f(\mathbf{w},\theta)}{\partial \mathbf{w}}\frac{\partial \mathbf{w}}{\partial\theta} + \frac{\partial f(\mathbf{w},\theta)}{\partial\theta}\right]
\end{aligned}$$

□

The fact that the derivative of $\mathcal{F}$ can be expressed as the expectation of f when the variational posterior is a Gaussian follows from this proposition combined with Lemma 1 (I have made this explicit and added a proof; this lemma was assumed without statement in the paper).

Lemma 1. *For $q(\mathbf{w}|\theta)$ a Gaussian distribution parameterised by $\theta = (\mu, \sigma)$ such that $\mathbf{w} = \boldsymbol{\mu} + \boldsymbol{\sigma} \circ \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$, we have that $q(\boldsymbol{\epsilon})d\boldsymbol{\epsilon} = q(\mathbf{w}|\theta)d\mathbf{w}$, where the $q(.)$ are probability density functions and $\circ$ is pointwise multiplication of matrices.*

Proof. Firstly, from the definition of $\mathbf{w} = \boldsymbol{\mu} + \boldsymbol{\sigma} \circ \boldsymbol{\epsilon}$, we have that $\frac{d\mathbf{w}}{d\boldsymbol{\epsilon}} = \boldsymbol{\sigma}$. Secondly, from the definition of the Gaussian density function:

$$\begin{aligned}
q(\mathbf{w}|\theta) &= \frac{1}{\sqrt{2\pi\boldsymbol{\sigma}^2}}\exp\left(-\frac{(\mathbf{w}-\boldsymbol{\mu})^2}{2\boldsymbol{\sigma}^2}\right) \\
&= \frac{1}{\sqrt{2\pi\boldsymbol{\sigma}^2}}\exp\left(-\frac{(\boldsymbol{\mu}+\boldsymbol{\sigma}\circ\boldsymbol{\epsilon}-\boldsymbol{\mu})^2}{2\boldsymbol{\sigma}^2}\right) \\
&= \frac{1}{\sqrt{2\pi\boldsymbol{\sigma}^2}}\exp\left(-\frac{\boldsymbol{\epsilon}^2}{2}\right) \\
q(\boldsymbol{\epsilon}) &= \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{\boldsymbol{\epsilon}^2}{2}\right)
\end{aligned}$$

Therefore, $\frac{q(\boldsymbol{\epsilon})}{q(\mathbf{w}|\theta)} = \boldsymbol{\sigma}$. The conclusion follows. □

3.1.2 Gaussian variational posterior

This section presents the resulting algorithm when we assume the variational posterior follows a Gaussian distribution. It is very similar to Algorithm 1, with two differences. (i) They parametrise the standard deviation $\boldsymbol{\sigma}$ of the variational posterior pointwise with a further parameter $\boldsymbol{\rho}$ which is fed through the softplus function to get $\boldsymbol{\sigma}$: $\boldsymbol{\sigma} = \log(1 + \exp(\boldsymbol{\rho}))$. This is apparently to ensure that

"$\boldsymbol{\sigma}$ is always non-negative" [1, p.4]. (ii) They explicitly give the gradient of $f(\mathbf{w}, \theta)$ with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\rho}$; this comes from Proposition 1, and it is not necessary for the implementation of the algorithm.

3.1.3 Scale mixture prior

This section has a discussion of their choice of prior $P(\mathbf{w})$. They propose a scale mixture of two Gaussians, and give some intuition for why this is a good choice, as well as for why they they do not adjust its parameters during training.

The same prior is shared across all weights, and has the form:

$$P(\mathbf{w}) = \prod_j \pi \mathcal{N}(\mathbf{w}_j|0, \sigma_1^2) + (1-\pi)\mathcal{N}(\mathbf{w}_j|0, \sigma_2^2)$$

where the product is over all the weights in the neural network ($\mathbf{w}_j$ is the jth weight); $\mathcal{N}(x|\mu, \sigma^2)$ is the Gaussian density evaluated at x with mean μ and variance σ^2; σ_1^2 and σ_2^2 are the variances of the two components; and $\pi \in [0, 1]$ controls the scaling of the mixture. Ideally, the hyperparameters σ_1, σ_2 and π are chosen by cross-validation (see Section 4). Setting $\sigma_1 > \sigma_2$ and $\sigma_2 \ll 1$ provides two purported benefits: the component with large variance gives the prior density a heavier tail than a plain Gaussian, and the component with the small variance causes many of the weights to a priori concentrate tightly around zero. I discuss why the authors seem to take these properties to be beneficial in Section 5.

The authors found that optimising the parameters during training (by gradient descent) "yields worse results" [1, p.4] (a procedure known as empirical Bayes). The intuition they give is that since there are only three prior parameters but many ($2|\mathbf{w}|$) posterior parameters, it is easy for gradient descent to optimise the prior parameters to fit the initial distribution of weights (which is unlikely to be very good). Thus, the loss (Eq. 1) is unwilling to move away from the poor initial posterior parameters, an effect the authors dub "the introduction of strange local minima" [1, p. 4].

4 Experiments

4.1 Classification on MNIST

To support the claim that BBB prevents overfitting, the original paper trained various networks of various sizes on the MNIST digits dataset, which consists of 60,000 training and 10,000 testing pixel images of size 28 by 28. They withheld 10,000 of the training images as a validation set to pick hyperparameters (see below for details). They preprocessed pixels by dividing values by 126. We repeated the same procedure, with one difference: the number of hyperparameter combinations that they report to have trained and validated would have taken over 1 year on our GPU. Therefore we were only able to grid search a small subset (around 3%[2]) of the hyperparameter space compared to the original paper (even that took 12 days). Table 1 shows the final test errors for different methods achieved in the original paper and by us. Figure 2, taken from the original paper, has their learning curves produced on the MNIST test set, for three methods using two hidden layers of 1200 ReLUs. Figure 3 is our attempt to replicate this. Table 2 enumerates all hyperparameters, along with the values that were grid searched, in the original paper and by us.

[2] As Table 2 shows, they grid searched $2 \times 3 \times 3 + 1 \times 3 \times 3 \times 4 \times 2 \times (3 + 3^3) = 2178$ hyperparmater combinations; we could only search through 60.

Method	# Units/Layer	# Weights	Test error (Blundell et al.)	Test error (us)
SGD	400	500k	1.83%	3.1%
	800	1.3m	1.84%	2.97%
	1200	2.4m	1.88%	2.85%
SGD, dropout	400	500k	1.51%	3.16%
	800	1.3m	1.33%	2.96%
	1200	2.4m	1.36%	3.04%
Bayes by Backprop, Gaussian	400	500k	1.82%	1.45%
	800	1.3m	1.99%	1.48%
	1200	2.4m	2.04%	1.56%
Bayes by Backprop, Scale mixture	400	500k	1.36%	1.57%
	800	1.3m	1.34%	1.59%
	1200	2.4m	1.32%	1.52%

Table 1: Test error of different methods on MNIST task, according to the paper and as produced by us. # Units/Layer is the number of ReLUs used in the two hidden layers. We trained for 1200 epochs and report minimum test errors. It is unclear exactly how long the original paper trained for, but it was for at least 600 epochs. It is unclear whether they report final or minimum test errors.

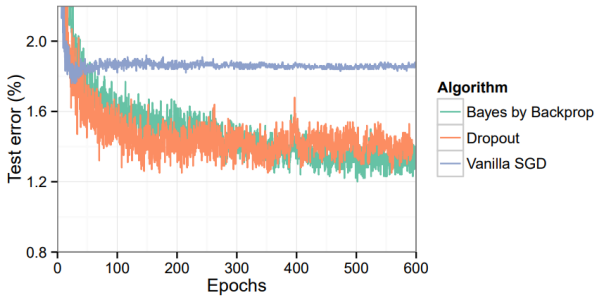

Figure 2: From the original paper: test error on MNIST as training progresses.

In summary, whilst we managed to replicate evidence for the important claim, namely that BBB prevents overfitting as well as Dropout (or, as we found, better), we did not achieve test errors as low as theirs.

4.2 Regression curves

Having provided evidence that BBB alleviates overfitting, giving performance comparable to dropout, the authors then explore the use of BBB to assess the uncertainty of a neural network in its output. To do so, they generate training data for the curve:

$$y = x + 0.3\sin(2\pi(x+\epsilon)) + 0.3\sin(4\pi(x+\epsilon)) + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, 0.02^2)$. Figure 4 features in the original paper, as evidence that BBB allows for assessment of uncertainty. An ordinary neural network reduces the variance to zero outside of the training data, choosing to fit a particular function, even though there are many possible extrapolations. However, with BBB, each network in the implicit infinite ensemble fits a different function. Each network in the ensemble comes from a different sample of weights from the variational posterior. The networks agree in the region where there is training data, but then diverge in their predictions outside that region, reflecting there being many possible extrapolations.

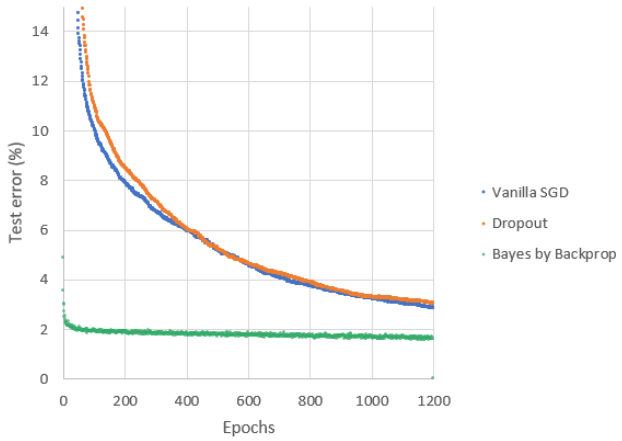

Figure 3: Our reproduction of Figure 2.

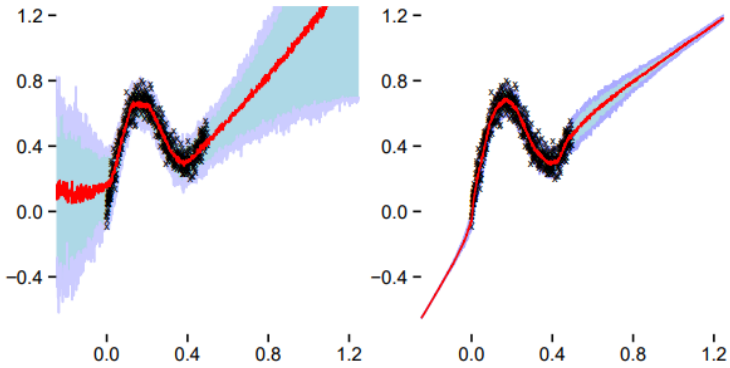

Figure 4: From the original paper: regression of noisy data with interquartile ranges. Black crosses are training samples. Red lines are median predictions. Blue/purple region is interquartile range. Left: Bayes by Backprop neural network, Right: standard neural network.

Frustratingly, the original paper did not give many implementational details. They mention that they minimised a conditional Gaussian loss, however the model architecture, hyperparameters and initialisation, as well as the optimizer and amount of training and training data are left unspecified. Figuring out these details came at a significant time cost (our team spent at least 24 focused person-hours). A key frustration was that the results were particularly sensitive to the initial μ and ρ of the weights. Eventually one team member found a suitable combination; the details can be found in our code. Using these, we could reproduce their regression curves, as in Figure 5. The training time for BBB is equally costly. It took around 450 epochs for MLP to converge, but 6000 and 20000 epochs for BBB to converge, with 400 and 800 hidden units respectively. In conclusion, this experiment was reproducible, but only with costly hyperparameter tuning, which was not mentioned in the paper.

In light of these difficulties, we compared BBB with another common method for Bayesian approximation in neural networks: Dropout [11]. With far less effort[3], we found this method to give equally legitimate results, as shown in Figure 6. The key difference is that Dropout does not appear to want to change the sign of the derivative of the extrapolated function, whereas BBB

[3]The model has fewer moving parts such that it worked out-of-the-box (c.f. 24 person-hours of hyperparameter tuning), and training took around 1000 epochs (c.f. up to 20000 epochs).

Hyperparameter	Blundell et al.	Us
ReLUs per hidden layer	400 800 1200	400 800 1200
Optimizer and its hyperparameters	SGD, $lr = 10^{-3}$ SGD, $lr = 10^{-4}$ SGD, $lr = 10^{-5}$	SGD, $lr = 10^{-3}$ SGD, $lr = 10^{-4}$ SGD, $lr = 10^{-5}$ Adam, $lr = 0.001, betas = (0.9, 0.999), eps = 10^{-8}$ Adam, as above, weight decay= 0.01 Adam, as above, weight decay= 0.1
Number of Monte Carlo samples from variational posterior	1 2 5 10	10
Do KL reweighting	True False	True False
Gaussian prior: $-\log\sigma$	0 1 2	0 1
Scale mixture prior: π	 1/4 1/2 3/4	1/2
$-\log\sigma_1$	0 1 2	0 1
$-\log\sigma_2$	6 7 8	6

Table 2: Hyperparameters considered in grid search in the original paper and by us. The top two hyperparameters are for all the networks; the bottom six are only for BBB. Note that we did not consider different optimizers and learning rates in the grid search for BBB hyperparameters. Due to limited time, we used only Adam with $lr = 10^{-3}$ because we found this to give good results without any tuning. At test time, in order to make a fair comparison, we used SGD with $lr = 10^{-3}$ for all algorithms, which was the learning rate which gave the best results in each case during validation (even though that was with Adam for BBB).

readily does so. Whilst it is clear that there are large practical benefits to using Dropout instead of BBB, it is unclear whether a willingness to change the derivative's sign is a desirable property, so our experiment leads to no conclusion on which method leads to better assessment of uncertainty. See Section 5 for a suggestion of how that might be done in future work.

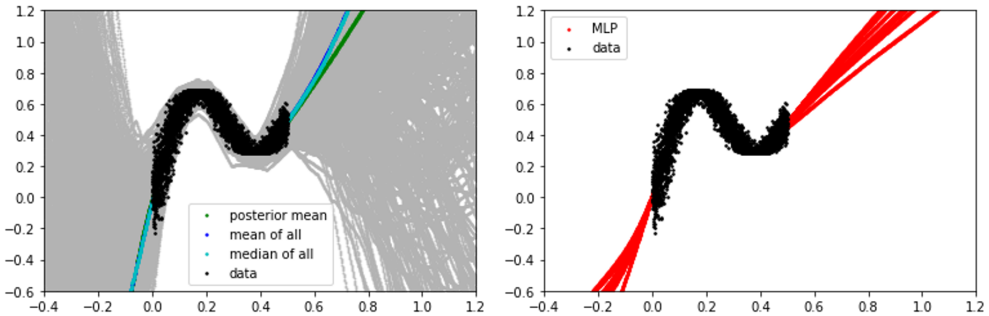

Figure 5: Our reproduction of Figure 4. Both networks use two hidden layers of 800 ReLUs. The MLP predictions are made by an ensemble of 10 networks.

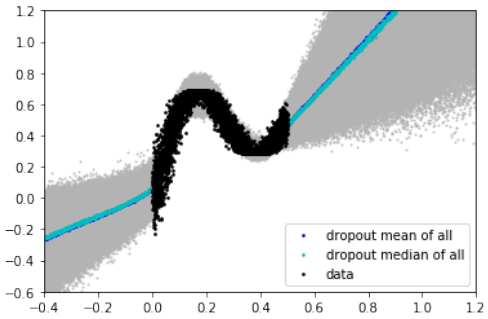

Figure 6: Left: Dropout, with $p = 0.5$ and the same network architecture as BBB and the MLP in Figure 5.

Interestingly, using fewer hidden units for BBB seemed to lead to lower variance estimates, whereas the opposite was true for Dropout, as shown in Figure 7.

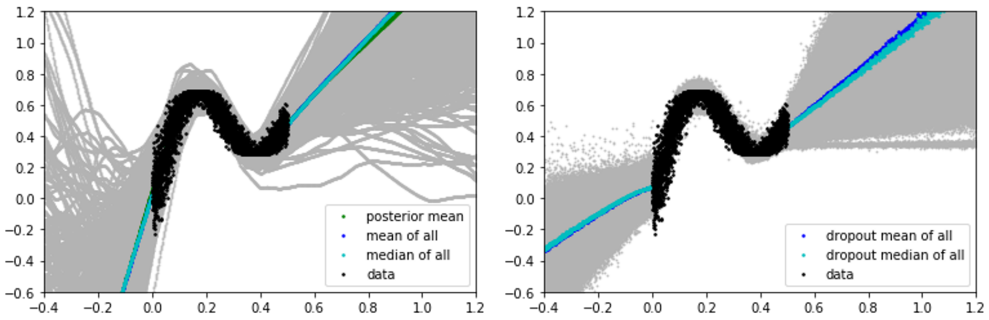

Figure 7: Left: BBB. Right: Dropout, with $p = 0.5$. Both networks use two hidden layers of 400 ReLUs.

4.3 Bandits on Mushroom Task

Having demonstrated that BBB helps with regularisation and uncertainty assessment, the authors perform a final experiment to give evidence for their claim that BBB adapts the idea of Thompson sampling for neural networks, and thus is an effective way to resolve the exploration vs. exploitation trade-off.

The setup is to take the UCI Mushrooms data set [12] and cast it as a bandit task, similar to [13, Ch. 6]. This is done as follows: on each step,

- The agent is presented with a mushroom which has 22 features that serve as the context x, and a choice to either eat or not eat the mushroom.
- Eating a non-poisonous mushroom gives a reward of 5.
- Eating a poisonous mushroom gives a reward of -35 with probability $\frac{1}{2}$, otherwise a reward of 5.
  - This makes the RL task more difficult. If poisonous mushrooms deterministically gave a reward of -35, then a simple agent could achieve zero regret on every action as soon as it had seen all the 8124 mushrooms.
- Not eating a mushroom gives a reward of 0.

The authors propose the following setup for instantiating the agent with a neural network. The network has 24 input units (22 for the features, which we convert from ASCII characters to their integer representations, and a one-hot encoding of the action), and 1 output unit (the expected reward). At each step, the agent does two forward passes, one for each action, and picks the action with the highest expected reward. It then trains on 64 minibatches of size 64. This is made possible by maintaining the most recent 4096 context, action and reward tuples in a buffer, and drawing minibatches uniformly at random from this buffer[4]. Thus, for every interaction with the mushroom bandit, the agent trains on 4096 examples. Figure 8 shows the results obtained in the original paper, for several agents. They plot cumulative sum of regret against number of interactions with the mushroom bandit. Regret on a single step is defined as the difference between the reward received by an oracle and by the agent. For this task, an oracle receives a reward of 5 and 0 respectively on a non-poisonous and poisonous mushroom. BBB is compared with three ϵ-greedy agents. These agents use a simple heuristic for trading-off exploration vs. exploitation: pick an action uniformly at random with probability ϵ, else maximise expected reward as per usual. Presumably, the authors select these agents for comparison because they are trying to assess the ability of BBB to make this trade-off.

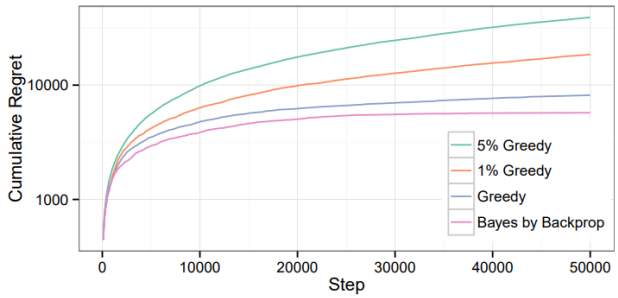

Figure 8: From the original paper: comparison of cumulative regret of various agents on the mushroom bandit task, averaged over five runs. BBB has the lowest cumulative regret after 50,000 steps.

Again, the authors do not fully specify the hyperparameters of their networks. They use two hidden layers of 100 ReLUs, and take two samples of weights from the variational posterior when training BBB. Apart from that, we were left to guess the other hyperparameters, of which there are 180 combinations[5], even without the weight initialisation. We presumed that they used a conditional Gaussian loss function, since the network output is scalar as in the regression task. We did not have time to do a grid search for the remaining hyperparameters, because a single BBB agent takes around 15 hours of GPU time to play for 50,000 steps. Instead, we could perform only 3/180 of the experiments that they presumably did. For the same reason, we were only able to compute cumulative regret over 1 run (as opposed to averaging over 5, as in the original paper). Figure 9 shows our attempt to replicate the results in Figure 8.

Whilst we were able to replicate the performance of the greedy agents, we did not find the BBB results to be replicable. In particular, the BBB agent did no better than guessing[6] for both the first and the final 1000 steps, for all three hyperparameter combinations we tried.

We suspect that this is due to not being able to perform the full hyperparameter search. In particular, weight initialisation is a likely to be a significant problem, since (i) this was what the

[4]Before training begins, the agent plays 4096 times to fill the buffer.

[5]This supposes they performed the same grid search as in the MNIST experiment.

[6]An agent which guesses actions uniformly at random for this task get expected regret per 1000 steps of about 5000, which is what BBB achieved.

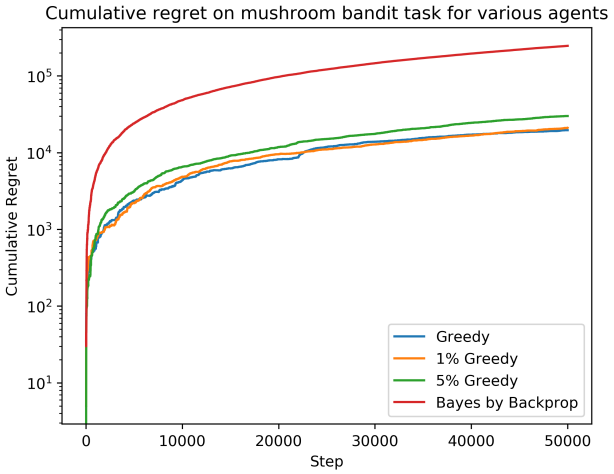

Figure 9: Reproduction of Figure 8 from the original paper

regression results so difficult to replicate, and (ii) the authors and we observed rather different behaviour in the pure greedy agent, at the beginning of training. They claim this agent chooses "to eat nothing ... for the first 1,000" steps, whereas we found the greedy agent to eat 456 of the first 1,000 mushrooms. However, this different behaviour at the beginning of training could equally be due to using a different optimizer. We used Adam; the authors did not specify their optimizer.

4.4 BBB to Drive Active Learning

Finally, we explored the use of BBB in an active learning framework. We were curious to explore the ability of weight uncertainty to help in settings where the labelling of data is expensive. The thought was that using the information entropy of the predictions made by BBB on a datum would indicate how surprising that datum is to the network, and therefore how much information it would gain by receiving its true label. Intuitively, if the network could then request the labels of data with the highest entropy, it may perform better than a network which receives the labels of random data. We tested this hypothesis with the following setup:

1. Train a "shared baseline" neural network on 1500 MNIST examples split over 150 batches for 5 epochs..
2. Duplicate the network to get `random_net` and `active_net`.
3. Train `random_net` on 25 further batches, chosen u.a.r.
4. Use `active_net` to classify 50 further batches of data, chosen u.a.r.
5. Train `active_net` on 25 batches worth of the highest entropy data.
6. Evaluate both nets on 2500 points from the test set.

Disappointingly, the results were ambiguous. On many examples, `random_net` performed better than `active_net`. Figure 10 is a histogram representing the entropy of the points chosen by `active_net` in step 4 above. We think it looks pretty reasonable, thus the basic method of discriminating between data on the basis of entropy appears not to be an unreasonable idea. One hypothesis explaining the failure is that just 25 further batches is not enough data for significant

differences in test error to emerge. Another hypothesis is that feeding the *most* uncertain examples can be counterproductive: these might just be very poor, that is to say, borderline, training examples. Instead, a good function for active learning might need to discriminate between two types of epistemic uncertainty: one that is high on examples that just are poor, and will always remain borderline cases, even for a close to perfect network (or human, for that matter); and another that is high on examples that have a clearly determinate label but nonetheless have a highly uncertain label, conditional on the current model parameters. Given more time, we would like to explore these hypotheses.

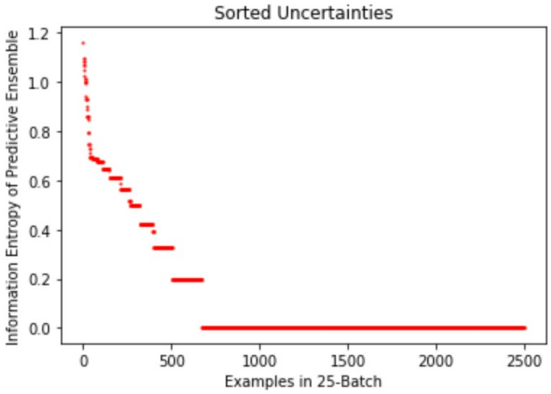

Figure 10: Histogram of information entropy of the 2500 examples selected by `active_net` for labelling.

5 Conclusion

5.1 Critical Evaluation

As a team, we managed to replicate the essential results of two from three experiments, and failed to replicate one. Of our two extensions, one was successful and the other was not. In my opinion, our biggest shortcoming was not helping each other more when our respective experiments ran into issues. In particular, debugging as a team is difficult because it requires reading and understanding someone else's code, however it may have been fruitful because when we did attempt this, we were sometimes able to quickly spot problems, once we had understood the code. On the other hand, it seems that more *skilled* debugging, rather than just more debugging eyes was our key bottleneck.

Personally, I struggled with this. Upon finding that each run of BBB on the mushroom bandit task took 15 hours, I thought it would be impossible to find a workable hyperparameter combination in the allotted time. However, the failure of each of the three combinations I tried would have been obvious within the first 5000 steps. If I had realised this, I could have tried many more combinations, and probably successfully replicated the result. This lack of meta-thinking when debugging prevented me from being a more successful team member.

5.2 Future Work

Given more time, I would like to run experiments which would aim to prove or disprove more general versions of the three key claims made by the authors. As it stands, I believe that they provided evidence for the following claims: after (rather extensive and costly) hyperparameter tuning, BBB appears to (i) provide regularisation comparable to dropout on the MNIST dataset, and an effective way to prune network weights; (ii) have appropriate epistemic uncertainty outside of the region

where there is training data, on non-linear regression of one sinusoidal function; and (iii) achieve a lower cumulative regret than ϵ-greedy agents for three values of ϵ on a contextual bandit problem. My curiosity is drawn to assessing whether these claims hold up more generally.

More concretely: does (i) hold up for different datasets, and in particular does BBB actually provide a scalable method of regularisation, in light of the difficulties we found with hyperparameter tuning? Further, it would be interesting to explore the effect of different priors on regularisation and weight redundancy. On this point, the authors claim that "the scale mixture prior used by Bayes by Backprop encourages a broad spread of the weights", such that "even when 95% of the weights are removed the network still performs well" [1, p. 7]. They give empirical evidence for this latter claim, but not for the assertion that it is the scale mixture prior which gives BBB this property. The claim is confusing, because it would seem that *any* prior with high variance would give BBB this property. Perhaps what they are claiming is that, given that the other main effect of the prior is to provide regularisation, and the presence of a lower variance prior is what has this regularising effect, the scale mixture of this prior with a higher variance prior encourages both regularisation and weight redundancy[7]. I would like to quantify and compare test error and weight redundancy in networks with various combinations of priors. This might help shed light on Figure 3 in the original paper, which appears to show the intriguing result that apparently regularised networks have a broader spread of weights that non-regularised ones. Is this only true when the networks also are optimised to have weight redundancy? Or is the intuition that regularised networks have smaller weights false for other reasons?

Does (ii) hold up for different non-linear regression problems? How about in other domains? For instance, domains where we can quantify the trade-off between calibration and informativeness of predictions (for example by using the logarithmic scoring rule) would be interesting, because this would give an indication of precisely how well BBB quantifies its uncertainty. The authors cite papers which show that "variational methods under-estimate uncertainty" [1, p. 6]; does BBB tend to be over-confident, too?

Finally, does (iii) hold up when BBB is compared against slightly more sophisticated strategies for trading-off exploration vs. exploitation, such as Decaying ϵ_t-Greedy, or Optimistic Initialisation? How does it fare in other domains? The authors claim that increasing the number of samples from the variational posterior reduces the variance of action selection, favouring exploitation. Can this be shown experimentally? Can even better performance be achieved with an agent which draws an increasing number of variational samples as training progresses?

6 Appendix

6.1 Code submission

Here is a link to an anonymous GitHub repo containing the code for our experiments: `tinyurl.com/y4ym3ncg`

6.2 Code contribution

- Classes `GaussianReparam`, `ScaleMixtureGaussian`, `BayesianLayer` and `BayesianNet` adapted from `tinyurl.com/y4ynt24z`.
- Functions `test()` and `test_ensemble()` also adapted from `tinyurl.com/y4ynt24z`.

[7] There is some evidence for this interpretation in the observation that MNIST test error increases with number of hidden units when BBB has a Gaussian prior, but decreases when BBB has a Scale Mixture prior.

- Helper function `get_train_valid_loader()` adapted from tinyurl.com/y5q87jsk.
- The remaining code is original.

6.3 Word count

5011 words (excluding Contents, Appendix, References and tables)

6.4 Minibatches and KL re-weighting

In Section 3.4 of the paper, the authors discuss how the algorithm is amenable to minibatch optimisation, whereby for each epoch, the training data $\mathcal{D}$ is randomly partitioned into M equally-sized subsets, $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_M$, and one gradient is computed per subset, resulting in M updates per epoch. Graves [10] proposes a slight modification of the loss function (Eq. 1) in the minibatch case, by minimising the following for minibatch $i = 1, 2, ..., M$:

$$\mathcal{F}_i^{EQ}(\mathcal{D}_i, \theta) = \frac{1}{M}\mathrm{KL}\left[\, q(\mathbf{w}|\theta) || \, P(\mathbf{w})\right] - \mathbb{E}_{q(\mathbf{w}|\theta)}[\log P(\mathcal{D}_i|\mathbf{w})]$$

As pointed out by the authors, this is equivalent to minimising the cost in Eq. 1 since $\sum_i \mathcal{F}_i^{EQ} = \mathcal{F}(\mathcal{D}, \theta)$. A further suggestion in this direction is to change at each minibatch the weighting of the complexity cost relative to the likelihood cost. If the minibatches are partitioned uniformly at random on each epoch, then for λ such that $\lambda \in [0, 1]^M$ and $\sum_{i=1}^M \lambda_i = 1$, we can define:

$$\mathcal{F}_i^{\lambda}(\mathcal{D}_i, \theta) = \lambda_i \mathrm{KL}\left[\, q(\mathbf{w}|\theta) || \, P(\mathbf{w})\right] - \mathbb{E}_{q(\mathbf{w}|\theta)}[\log P(\mathcal{D}_i|\mathbf{w})]$$

As with $\mathcal{F}_i^{EQ}$, since we have that $\mathbb{E}_M[\sum_{i=1}^M \mathcal{F}_i^{\lambda}(\mathcal{D}_i, \theta)] = \mathcal{F}(\mathcal{D}, \theta)$, where the expectation is over the partitioning of minibatches, this is again equivalent to minimising the cost in Eq. 1. The authors found the scheme $\lambda_i = \frac{2^{M-i}}{2^M - 1}$ to work well. The intuition for this is that it puts most of the weight on the complexity cost for the first minibatches in the epoch, such that when the algorithm has not yet seen much data, it is encouraged to keep the weights close to the prior. For later epochs, the data become exponentially more influential.

6.5 Contextual Bandits

In Section 4 of the paper, the authors explore the benefits of using BBB to solve a simple RL problem called Contextual Bandits. The setup is as follows: on each step,

- The agent is presented with a context x and a choice of one of K possible actions a.
- Different actions yield different unknown rewards r.
- The agent must pick an action with the highest expected reward.
- The context is assumed to be independent of any previous contexts, actions or rewards.
- The agent does not know what reward it could have received for an action that it did not pick (the "absence of the counterfactual").

The agent builds of model of the distribution of rewards conditioned on action and context: $P(r|x, a, \mathbf{w})$, and uses this model to pick an action.

6.5.1 Thompson Sampling for Neural Networks

The key challenge with this RL task is to trade-off between exploitation (picking the best known action) and exploration (picking an action that may be suboptimal, in order to learn more). Thompson sampling is one method to do so. In general, the procedure is:

1. Sample a new set of parameters $\mathbf{w}$ for the model $P(r|x, a, \mathbf{w})$.
2. Pick the action with the highest expected reward according to the model with these parameters.
3. Update the distribution from which $\mathbf{w}$ is sampled. Go to 1.

The intuition is that initially, with little information about which parameters are correct, the procedure will sample widely from the set of possible parameters (exploration). Then, with the correct updates, as play continues the distribution from which the parameters are sampled begins to converge, and so action selection becomes more deterministic, focusing on the known highest expected reward actions.

Now, the contribution made by the authors is that BBB provides a natural way to extend this methodology to neural networks:

1. Sample a new set of weights $\mathbf{w}$ from the variational posterior: $\mathbf{w} \sim q(\mathbf{w}|\theta)$.
2. Receive context x.
3. Pick the action a that maximises $\mathbb{E}_{P(r|x,a,\mathbf{w})}[r]$. Since we are modelling $P(r|x, a, \mathbf{w})$ with a neural network, this just means computing a forward pass for each possible action. Let $\hat{r}$ be the expected reward of the chosen action.
4. Receive true reward r.
5. Backpropagate the squared loss $(\hat{r} - r)^2$ (or some other loss function) through the network and update the variational parameters θ.

A final comment to make is that by using more than one Monte Carlo sample in step 1, and using the resulting networks as an ensemble (averaging their predictions for the reward of each action), we decrease the variance of action selection, trading off increased exploitation for reduced exploration.

References

[1] Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015). Weight uncertainty in neural networks. arXiv preprint arXiv:1505.05424.

[2] Radford M Neal and Geoffrey E Hinton. A view of the EM algorithm that justifies incremental, sparse, and other variants. In Learning in graphical models, pages 355–368. Springer, 1998.

[3] Jonathan S Yedidia, William T Freeman, and Yair Weiss. Generalized belief propagation. In Advances in Neural Information Processing Systems (NIPS), volume 13, pages 689–695, 2000.

[4] Karl Friston, Jeremie Mattout, Nelson Trujillo-Barreto, John Ashburner, and Will Penny. Variational free energy and the Laplace approximation. Neuroimage, 34(1):220–234, 2007.

[5] Lawrence K Saul, Tommi Jaakkola, and Michael I Jordan. Mean field theory for sigmoid belief networks. Journal of artificial intelligence research, 4(1):61–76, 1996.

[6] Tommi S. Jaakkola and Michael I. Jordan. Bayesian parameter estimation via variational methods. Statistics and Computing, 10(1):25–37, 2000.

[7] Manfred Opper and Cedric Archambeau. The variational Gaussian approximation revisited. Neural computation, 21(3):786–792, 2009.

[8] Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In Proceedings of the 2nd International Conference on Learning Representations (ICLR), 2014. arXiv: 1312.6114

[9] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In Proceedings of the 31st International Conference on Machine Learning (ICML), pages 1278–1286, 2014.

[10] Alex Graves. Practical variational inference for neural networks. In Advances in Neural Information Processing Systems (NIPS), pages 2348–2356, 2011.

[11] Gal, Y. and Ghahramani, Z., 2016, June. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In international conference on machine learning (pp. 1050-1059).

[12] Kevin Bache and Moshe Lichman. UCI Machine Learning Repository. University of California, Irvine, School of Information and Computer Sciences, 2013. URL http://archive.ics.uci.edu/ml.

[13] Arthur Guez. Sample-Based Search Methods For Bayes Adaptive Planning. PhD thesis, University College London, 2015.