The Neural Testbed: Evaluating Predictive Distributions

Ian Osband 1  Zheng Wen 1  Seyed Mohammad Asghari 1  Vikranth Dwaracherla 1  Xiuyuan Lu 1  Morteza Ibrahimi 1  Dieterich Lawson 1 2  Botao Hao 1  Brendan O’Donoghue 1  Benjamin Van Roy 1

Abstract
Posterior predictive distributions quantify uncertainties ignored by point estimates. This paper introduces The Neural Testbed: an opensource library for controlled and principled evaluation of agents that generate such predictions. Crucially, agents are assessed not only on the quality of their marginal predictions per input, but also on their joint predictions across many inputs. We evaluate a range of agents using a neural-network-based data generating process. Our results indicate that some well-known agents that produce accurate marginal predictions do not fare well with joint predictions. We show that the quality of joint predictions drives performance in downstream decision tasks, and highlight the importance of this observation to the community.

1. Introduction
We consider agents that are trained on data pairs \((X_t, Y_{t+1}) : t = 0, 1, \ldots, T-1\) and subsequently generate predictive distributions given new inputs. When presented with an input \(X_T\), such an agent can generate a predictive distribution of the outcome \(Y_{T+1}\) that is yet to be observed. This distribution characterizes the agent’s uncertainty about \(Y_{T+1}\). We refer to such a prediction as marginal to distinguish it from a joint predictive distribution over a sequence of prospective outcomes \((Y_{T+1}, \ldots, Y_{T+\tau})\) corresponding to inputs \((X_T, \ldots, X_{T+\tau-1})\).

Predictive distributions express uncertainty about future observations. The importance of such uncertainty estimation has motivated a great deal of research over recent years, much of which has been conducted by the Bayesian deep learning community (MacKay, 1992; Neal, 2012). This research has produced a variety of agents that generate predictive distributions. With this proliferation of alternatives, it is increasingly important to systematically study and improve their performance. In this paper, we introduce The Neural Testbed, which offers new tools to support this agenda.

1.1. A ‘unit test’ for uncertainty in deep learning
The Neural Testbed uses a synthetic data generating process to evaluate agents. Data is generated by random neural networks, designed to require agents to learn complex nonlinear relationships. Testing is carried out using newly generated independent samples to avoid bias due to the overfitting that arises with fixed data sets, as those commonly used in deep learning research. The Testbed thus allows for more systematic scientific analysis and validation of agent behavior.

Importantly, the Testbed assesses performance of not only marginal but also joint predictive distributions (Lu et al., 2021a). As we will see, some agents popular in the Bayesian deep learning literature produce highly inaccurate joint predictions even when their marginal predictions fare well.

The Testbed assesses accuracy via KL-divergence (equivalently, log-likelihood or cross-entropy loss) (Kullback & Leibler, 1951). Our results show that this metric clearly distinguishes joint predictive distributions produced by various agents. Further, as we will discuss, as the order of predictive distributions evaluated grows, this metric becomes universal in the sense that accuracy ensures effective decision making in any downstream task. This metric thus offers substantial benefits over measuring prediction quality based on particular downstream tasks. Agents tuned so that a particular decision algorithm performs well on a downstream task can fare poorly when the decision algorithm is improved because posterior predictives are inaccurate. Designing agents that address the root problem of generating accurate posterior predictives ensures robustness to such changes.

It is worth noting that the problem framed by the Testbed is a computational one. Optimal performance would be attained by carrying out exact Bayesian inference: given infinite compute time, an agent could calculate the posterior distribution, which maximizes performance. However, due to the complexity of the data generating process, this is infeasible. The agents we study serve as approximate inference algorithms. As we will see, stochastic gradient Markov chain Monte Carlo offers the strongest performance among these agents (Welling & Teh, 2011). However, that requires enormous compute time relative to the other agents we study. The modest compute time required by these alternatives make them viable approaches for practical applications.
1.2. Opensource code, reproducible research

As part of this project we release github.com/deepmind/neural_testbed, which instantiates all agents, environments and analysis included in this paper. This library serves to facilitate reproducible and accessible research on uncertainty in deep learning, without massive computation budgets (Sharir et al., 2020). It includes:

- Implementation of all experiments used in the paper.
- Tuned reference agents (+ sweeps) for several benchmark approaches to uncertainty modelling.
- Examples using TensorFlow, PyTorch and Jax.

We provide more details on code and usage in Section 3.

1.3. Related work

There is a rich literature around uncertainty estimation in deep learning. Much of this work has focused on agent development, with a wide variety of approaches including variational inference (Blundell et al., 2015), dropout (Gal & Ghahramani, 2016), ensembles (Osband & Van Roy, 2015; Lakshminarayanan et al., 2017), and MCMC (Welling & Teh, 2011; Hoffman et al., 2014). However, even when approaches become popular within particular research communities, there are still significant disagreements over the quality of the resultant uncertainty estimates (Osband, 2016; Hron et al., 2017; Le Folgoc et al., 2021).

Bayesian deep learning has largely relied on benchmark problems to guide agent development and measure agent progress. These typically include classic deep learning datasets but supplement the usual goal of classification accuracy to include an evaluation of the probabilistic predictions via negative log likelihood (NLL) and expected calibration error (ECE) (Nado et al., 2021). More recently, several efforts have been made to supplement these datasets with challenges tailored towards Bayesian deep learning (Band et al., 2021), and explicit Bayesian inference (Wilson et al., 2021).

This literature has largely focused on evaluating marginal predictions, paired with evaluation on downstream tasks (Riquelme et al., 2018). Our work is motivated by the importance of joint predictions in driving good performance in sequential decisions (Lu et al., 2021a). This shares much in common with the work of Wang et al. (2021), which highlights the importance of joint predictions. We build on this work to develop a streamlined and simplified evaluation which works for classification and not just regression.

1Agent evaluation costs less than 1 USD on Google cloud.
2Wang et al. (2021) say that “joint log-likelihood scores [are] determined almost entirely by the marginal log-likelihood scores...in practice, they provide little new information beyond marginal likelihoods.”. Our results, on the other hand, indicate that joint log-likelihoods can decisively differentiate methods that do or do not learn accurate joint predictions.

1.4. Key contributions

Our main contribution is to release The Neural Testbed, a clear and simple unit test for uncertainty estimation in deep learning. The testbed highlights the importance of joint predictions as well as marginals. Unlike many benchmarks, the neural testbed is accessible to researchers and students without large computing budgets. Together with the testbed, we tune several benchmark agents and provide these as reference to the community. This effort shares much of the motivation of bsuite (Osband et al., 2019), but with a focus on posterior inference rather than decision making.

The testbed presents an extremely simplified perspective on uncertainty estimation in deep learning. Despite this, we find that it can provide clear and novel signal in agent development, even for state-of-the-art agents. These results can help to reconcile some puzzling observations made in the literature. In particular, why some approaches developed by the Bayesian deep learning community, such as Bayes-by-backprop, dropout, and deep ensembles, perform poorly in sequential decision tasks despite faring well based on evaluation metrics of that community (Osband et al., 2018). Our results in Section 4 show that, while such methods produce accurate marginal predictions, they are no longer competitive when it comes to joint predictions.

In Section 5 we apply these approaches to decision problems; we show that the quality of joint predictions drives the quality of decision making. Our results contribute to a growing literature on ‘neural bandits’ (Lu & Van Roy, 2017; Riquelme et al., 2018). Our results provide clear evidence that the insights derived from the stylized and simple testbed can extend to complex and high-dimensional problems. This work can inform the design of efficient reinforcement learning agents; through quantifying their uncertainty about the world, and leveraging this to learn and act more effectively (Osband et al., 2016; Lu et al., 2021).

2. Evaluating predictive distributions

In this section, we introduce notation for the standard supervised learning we consider as well as our evaluation metric: KL-loss. We review the distinction between marginal and joint predictions, together with numerical schemes to estimate KL divergence via Monte Carlo sampling.

2.1. Environment and predictions

Consider a sequence of pairs \((X_t, Y_{t+1}) : t = 0, 1, 2, \ldots\), where each \(X_t\) is a feature vector and each \(Y_{t+1}\) is its target label. Each target label \(Y_{t+1}\) is produced by an environment \(\mathcal{E}\), which we formally take to be a conditional distribution \(\mathcal{E}(X_t)\). The environment \(\mathcal{E}\) is a random variable; this reflects the agent’s uncertainty about how labels are generated. Note that \(\mathbb{P}(Y_{t+1} \in \mathcal{E}(X_t)) = \mathcal{E}(X_t)\) and \(\mathbb{P}(Y_{t+1} \in \mathcal{E}(X_t)|X_t) = \mathbb{E}[\mathcal{E}(X_t)|X_t]\).
We consider an agent that learns about the environment from training data $D_T = \{(X_t,Y_{t+1}) : t = 0,1,\ldots,T-1\}$. After training, the agent predicts testing class labels $Y_{T+1:T+\tau} \equiv (Y_{T+1},\ldots,Y_{T+\tau})$ from unlabeled feature vectors $X_{T:T+\tau-1} \equiv (X_T,\ldots,X_{T+\tau-1})$.

We describe the agent’s predictions in terms of a generative model, parameterized by a vector $\theta_T$ that the agent learns from the training data $D_T$. For any inputs $X_{T:T+\tau-1}$, $\theta_T$ determines a predictive distribution, which could be used to sample imagined outcomes $Y_{T+1:T+\tau}$. Hence, the agents’ $T$th-order predictive distribution is given by

$$
\hat{P}_{T+1:T+\tau} = P(\hat{Y}_{T+1:T+\tau} \in \cdot | \theta_T, X_{T:T+\tau-1}),
$$

which represents an approximation to what would be obtained by conditioning on the environment:

$$
P_{T+1:T+\tau}^* = P(Y_{T+1:T+\tau} \in \cdot | \mathcal{E}, X_{T:T+\tau-1}).
$$

If $\tau = 1$, this represents a marginal prediction; that is a prediction of a single label. For $\tau > 1$, this is a joint prediction over labels for $\tau$ different inputs.

### 2.2. Marginal versus joint predictions

Evaluating an agent’s ability to estimate uncertainty on joint instead of marginal predictions can result in very different answers. We provide a simple example that illustrates the point. Suppose the data $((X_t,Y_{t+1}) : t = 0,1,\ldots)$ is generated by repeated tosses of a possibly biased coin with unknown probability $p$ of heads. Let $X_t = 0$, to indicate that there is no input. Outcomes $Y_{t+1}$ are binary-valued. Consider two agents that, without any training, predict outcomes. Agent 1 assumes $p = 2/3$ and models the outcome of each flip as independent conditioned on $p$. Agent 2 assumes that $p = 1$ with probability $2/3$ and $p = 0$ with probability $1/3$; that is, the coin either produces only heads or only tails.

Let $\hat{Y}^1_{t+1}$ and $\hat{Y}^2_{t+1}$ denote the outcomes imagined by the two agents. Despite their differing assumptions, the two agents generate identical marginal predictive distributions:

$$
P(\hat{Y}^1_{t+1} = 0) = P(\hat{Y}^2_{t+1} = 0) = 1/3.
$$

On the other hand, joint predictions greatly differ for large $\tau$:

$$
P(\hat{Y}^1_1,\ldots,\hat{Y}^1_\tau = 0) = 1/3^\tau \ll 1/3 = P(\hat{Y}^2_1,\ldots,\hat{Y}^2_\tau = 0).
$$

Evaluating marginal predictions cannot distinguish between the two possibilities, though for a specific prior distribution over $p$, one agent could be right and the other wrong. One must evaluate joint predictions to make this distinction.

When it comes to decision-making, this distinction can be critical (Lu et al., 2021a). In a casino, under the first agent’s assumption, there is large upside and little risk to repeatedly betting on heads. However, if there is a $1/3$ chance the coin will always land tails, as is the case in the second agent’s prediction, there is a ruinous risk to repeatedly betting heads. Evaluating joint predictions beyond marginals distinguishes these cases.

### 2.3. Kullback–Leibler loss

We use expected KL-loss to quantify the error between an agent’s predictive distribution $\hat{P}_{T+1:T+\tau}$ and the prescient prediction $P_{T+1:T+\tau}^*$ that would be made given full knowledge of the environment:

$$
d_{KL}^* = \mathbb{E}[d_{KL}(P_{T+1:T+\tau}^* \| \hat{P}_{T+1:T+\tau})].
$$

The expectation is taken over all random variables, including the environment $\mathcal{E}$, the parameters $\theta_T$, $X_{T:T+\tau-1}$, and $Y_{T+1:T+\tau}$. Note that $d_{KL}$ is equivalent to the widely used notion of cross-entropy loss, though offset by a quantity that is independent of $\theta_T$.

In contexts we will consider, it is not possible to compute $d_{KL}$ exactly. As such, we will approximate $d_{KL}$ via Monte Carlo simulation, as described by Algorithm 1. First, a set of environments is sampled. Then, for each sampled environment, a training dataset is sampled. For sampled environment and corresponding training data set, the agent is re-initialized, trained, and then tested on $N$ independent test data $\tau$-samples. Note that each test data $\tau$-sample includes $\tau$ data pairs. For each test data $\tau$-sample, the likelihood of the environment $p_{j,n}$ is computed exactly, but that of the agent’s predictive distribution is approximated via another Monte Carlo simulation, and we use $\hat{p}_{j,n}$ to denote this approximation. The estimate of $d_{KL}^*$ is taken to be the sample mean of these log-likelihood ratios.

### 3. The Neural Testbed

In this section we introduce the Neural Testbed. We believe that a simple, clear and accessible testbed can provide significant value to community. We provide a high-level overview of the open source code which we release at github.com/deepmind/neural_testbed. We
then provide more details on the underlying generative model, together with an extensive selection of benchmark agents that we have tuned to perform well in this setting.

![Diagram of the Neural Testbed](image)

**Figure 1. Overview of the Neural Testbed.** For each random seed the testbed samples an environment realization with training and testing data. The testbed then compares the estimated log-likelihood of the true environment and that of a trained agent to estimate KL divergence.

### 3.1. Synthetic data generating processes

By data generating process, we do not mean only the conditional distribution of data pairs $(X_t, Y_{t+1})|E$ but also the distribution of the environment $E$. The Testbed considers 2-dimensional inputs and binary classification problems, although the generating processes can be easily extended to any input dimension and number of classes. The Testbed offers three data generating processes distinguished by a “temperature” setting, which signifies the signal-to-noise ratio (SNR) of the generated data. The agent can be tuned separately for each setting. This reflects prior knowledge of whether the agent is operating in a high SNR regime such as image recognition or a low SNR regime such as weather forecasting.

To generate a model, the Testbed samples a 2-hidden-layer ReLU MLP with 2 output units, which are scaled by 1/temperature and passed through a softmax function to produce class probabilities. The MLP is sampled according to standard Xavier initialization (Glorot & Bengio, 2010), with the exception that biases in the first layer are drawn from $N(0, \frac{4}{N})$. The inputs $(X_t : t = 0, 1, \ldots)$ are drawn i.i.d. from $N(0, I)$. The agent is provided with the data generating process as prior knowledge.

In Section 2.3, we described KL-loss as a metric for evaluating performance of an agent. The Neural Testbed estimates KL-loss, with $\tau \in \{1, 10\}$, for three temperature settings and several training dataset sizes. For each value of $\tau$, the KL-losses are averaged to produce an aggregate performance measure. Further details concerning data generation and agent evaluation are offered in Appendix A.

### 3.2. Why do we need a synthetic testbed?

The Neural Testbed is designed to be a maximally simple problem that investigates the key properties of uncertainty modeling in deep learning. Progress in deep learning has been driven both by challenge datasets that stretch agent capabilities (Deng et al., 2009; Krizhevsky et al., 2012), together with foundational work that builds understanding (Bartlett et al., 2021). In this work, we provide a benchmark designed to improve our understanding of probabilistic predictions beyond marginals. Doing well in the testbed is not necessarily an impressive grand success in AI, although doing poorly in such a simple setting may reveal fundamental flaws in algorithm design.

A key property of the testbed is that it is specified by a probabilistic model, rather than a finite collection of datasets. Benchmarks that rank performance on datasets are vulnerable to overfitting through iterative hill-climbing on the data included in the benchmark (Russo & Zou, 2016), which may not generalize to data outside of the benchmark (Recht et al., 2018). In contrast, access to a generative model means that we can produce an unlimited amount of testing data from our problem of interest. We can avoid the dangers of overfitting to any specific choices of benchmark dataset simply by generating more samples.

### 3.3. Benchmark agents

Table 3.3 lists agents that we study and compare as well as hyperparameters that we tune. In our experiments, we optimize these hyperparameters via grid search. Our implementations, which aim to match ‘canonical’ versions, are available at [github.com/deepmind/neural_testbed](https://github.com/deepmind/neural_testbed). Further detail on these agents is provided in Appendix B.

In addition to these agent implementations, our opensource offerings include all the evaluation code to reproduce the results of this paper. Our experiments make extensive use of parallel computation to facilitate hyperparameter sweeps. Nevertheless, the overall computational cost is relatively low by modern deep learning standards and relies only on standard CPUs. For reference, evaluating the mlp agent across all the problems in our testbed requires less than 3 CPU-hours. We view our opensource effort as a substantial contribution of this work.

### 3.4. How do I use the Neural Testbed?

The benchmark agents in Section 3.3 are written in Python and makes use of JAX internally (Bradbury et al., 2018). However, our testbed as a whole is framework agnostic, and can equally be used with any Python package including TensorFlow, PyTorch or even SKlearn. The high-level interface simply asks that an agent can take data and prior information to produce posterior probabilities at evaluat-
4. Results

We evaluate the benchmark agents of Section 3.3 across the Neural Testbed. We begin with an analysis of marginal predictions where, after agent tuning, all approaches are able to make reasonably good predictions. However, when we examine joint predictions we find that agent performance can vary drastically, even for well-tuned agents. If an agent cannot output accurate joint predictions in the testbed, we should question if we expect that same agent to perform better other settings. These results provide significant new insights to the the design of effective learning agents, and are a major contribution of this paper.

Figure 2. Most Bayesian deep learning approaches do not significantly outperform a single MLP in marginal predictions (τ = 1). Once we examine predictive distributions beyond marginals we see a clear difference in performance among our benchmark agents (τ = 10). For each τ, the KL estimates are normalized by the KL of the MLP agent.

4.1. Performance in marginal predictions

We begin our evaluation of benchmark approaches to Bayesian deep learning in marginal predictions (τ = 1). One of the first questions one might consider is whether the generative model as outlined in Section 3.1 represents a meaningful challenge for deep learning systems. Figure 3 compares the performance of naive uniform class probabilities, logistic regression, and a tuned 2-layer MLP. This simple comparison demonstrates that the Neural Testbed is not trivially solved by agents without deep learning architectures.

Figure 3. MLP agent outperforms naive baseline approaches in marginal predictions. This difference becomes more significant as the number of training points grows.
Marginal predictions have been the focus of the Bayesian deep learning literature. Despite this focus, Figure 2 shows that none of the benchmark methods significantly outperform a well-tuned MLP baseline in terms of $d_{\text{KL}}$. This observation is mirrored when we examine the average classification accuracy and expected calibration error (ECE) across the testbed (Table 2). These results are different from the empirical observations in other challenge datasets, where much agent development has focused on improving ECE, and present an interesting new observation in the Bayesian deep learning literature (Nado et al., 2021). We have two main hypotheses for this discrepancy. First, our agents are tuned for $d_{\text{KL}} = d_{\text{KL}}^1 + \frac{1}{10} d_{\text{KL}}^{10}$, not ECE (see Appendix B). Second, the generative model of Section 3.1 matches the agent architecture, with inputs sampled i.i.d. $\mathcal{N}(0, I)$. Investigating the conditions in which these results hold more generally is an exciting area for future research.

4.2. Performance beyond marginals

One of the key contributions of this paper is to evaluate predictive distributions beyond marginals. In Figure 2, the red bars show the results of benchmark agents evaluated on joint predictive distributions with $\tau = 10$. Unlike when evaluating on marginal predictions, where no method significantly outperforms a well-tuned MLP, the potential benefits afforded by Bayesian deep learning become clear when examining higher-order predictive distributions. Our results provide a stimulating counterpoint to prior works’ claims that examining $d_{\text{KL}}^\tau$ beyond marginals provides little new information (Wang et al., 2021).

Figure 2 shows that sgmcmc is the top-performing agent overall. This should be reassuring to the Bayesian deep learning community and beyond. In the limit of large compute this agent should recover the ‘gold-standard’ of Bayesian inference, and it does indeed perform best (Welling & Teh, 2011). However, some of the most popular approaches in this field (ensemble, dropout) do not actually provide good approximations to the predictive distributions of order $\tau = 10$. In fact, we even see that ensemble+ and hypermodels can provide much better approximations to the Bayesian posterior than ‘fully Bayesian’ VI approaches like bbb (Wilson & Izmailov, 2020). We note too that while sgmcmc performs best, it also requires orders of magnitude more computation than competitive methods even in this toy setting (see Appendix C.2). As we scale to more complex environments, it may therefore be worthwhile to consider alternative approaches to approximate Bayesian inference.

For insight into where our top agents are able to outperform, we compare ensemble and ensemble+ under the medium SNR regime in Figures 5 and 4. These methods are identical, except for the addition of a randomized prior function (Osband et al., 2018). Figure 5 shows that, although these methods perform similarly in the quality of their marginal predictions ($\tau = 1$), the addition of a prior function greatly improves the quality of joint predictive distributions ($\tau = 10$) in the low data regime. Note that, since the testbed considers 2D inputs, 100 training points may already be considered as in the high data regime. These results are quite intuitive: the effects of a prior function diminish as the agent gathers more data from the environment, so that the methods perform similarly in high data regimes.

Figure 6 provides some insight for how this benefit scales with the order $\tau$ of the predictive distribution. We can see a clear trend that as $\tau$ increases so does the separation between agents ensemble and ensemble+.

Figure 4 provides additional intuition into how the randomized prior functions are able to drive improved performance. Figure 4a shows a sampled generative model from our Testbed, with the training data shown in red and blue circles. Figure 4b shows the mean predictions and 4 randomly sampled ensemble members from each agent (top=ensemble, bottom=ensemble+). We see that, although the agents mostly agree in their mean predictions, ensemble+ produces more diverse sampled outcomes enabled by the addition of randomized prior functions. In contrast, ensemble produces similar samples, which may explain why its performance is close to baseline mlp.

5. Sequential decisions

Section 4 outlines agent performance on the Neural Testbed. We see that several state-of-the-art agents that perform similarly in marginal prediction have markedly different quality in terms of joint distributions. In this section, we will relate this problem to a sequential decision problem, and show that it is the quality in joint predictions that is essential to driving good performance in sequential decision making. Further, we see that the insights gained from the simple 2D Neural Testbed can extend to high-dimensional decision problems.
5.1. Problem formulation

One of the main motivations for the Neural Testbed is to design agents suitable for sequential decision problems. In this section, we outline an empirical evaluation of predictive distributions for sequential decisions through a Thompson sampling (TS) agent (Thompson, 1933).

To do this, we introduce a class of bandit problems derived from our Testbed formulation (Gittins, 1979). First, we sample a set of $N$ actions $\mathcal{X} = \{x_1, \ldots, x_N\}$ i.i.d. from a $d$-dimensional standard normal distribution. We then sample an environment $\mathcal{E}$, which specifies the conditional probability $\mathcal{E}(Y_{t+1} \in \cdot | X_t)$, according to the class of generative models described in Section 3.1. We pick the temperature, which controls the SNR, to be 0.1. At each timestep $t$, an agent selects an action $X_t \in \mathcal{X}$ and receives a reward $R_{t+1} = Y_{t+1}$. Let $\bar{R}_x = \mathbb{E}[R_{t+1} | \mathcal{E}, X_t = x]$ denote the expected reward of action $x$ conditioned on the environment, and let $X_* = \arg \max_{x \in \mathcal{X}} \bar{R}_x$ denote the optimal action. We assess agent performance through regret $\text{regret}(T) := \sum_{t=0}^{T-1} \mathbb{E}[\bar{R}_{X_*} - \bar{R}_x]$, which measures the shortfall in expected cumulative rewards relative to an optimal decision maker.

We evaluate the testbed agents on these bandit problems through actions selected by Thompson sampling, varying only the posterior predictive distributions that TS samples from. A TS agent requires an approximate posterior distribution over the environment, which is supplied by the testbed agents. At each timestep, TS samples an environment from the approximate posterior and selects an action that optimizes for the sampled environment (Thompson, 1933; Russo et al., 2018). This simple heuristic can effectively balance the needs of exploration and exploitation, and allows us to evaluate the ability of an approximate posterior to drive decision making. A complete algorithm and the details are presented in Appendix D.
5.2. Agent performance

We present empirical results of testbed agents on these random bandit problems with $N = 1000$ actions drawn from a $d = 50$ dimensional space. Figure 7 shows the average regret through time for each of the agents as selected by the Neural Testbed, averaged over 20 random seeds.\footnote{We omit sgmcmc as the computational demands are several orders of magnitude too large to consider in online learning.} We can clearly see that for each learning agent, the quality of decisions improves through time. However, the quality of decisions is greatly affected by the choice of agent. As expected, the agents that perform best in sequential decision problems seem to correlate with those that make accurate joint predictions on the testbed (Lu et al., 2021a).

![Figure 7. Learning agent impacts TS regret in neural bandits.](image)

To investigate the relationship between predictions and decisions we repeat the experiment of Figure 7 with 10 independent random initializations over all the testbed and bandit problems. We then empirically investigate the correlation between $d^\tau_{KL}$ and total regret at $T = 50,000$ for both $\tau = 1$ and $\tau = 10$. We use bootstrap sampling to estimate confidence intervals on the correlation coefficient on a logarithmic scale at the 5th and 95th percentiles. Figures 8 and 9 support our claim that performance in $d^{10}_{KL}$ is highly correlated with performance in sequential decision problems, whereas correlation to marginals is not significant. We would not expect a perfect correlation as the particular TS action selection strategy may introduce confounding factors, together with natural variability in seeds.

These results are significant in two ways. First, they provide empirical evidence that practical deep learning approaches separated by the quality of their joint predictions, but not their marginals, can lead to differing performance in downstream tasks. Second, we show that our simple 2D testbed can provide insights that scale to much higher dimension problems. In Appendix D.3 we show that these results are largely insensitive to input dimension. As such, we believe that sanity-checking approaches to probabilistic inference on the Neural Testbed offers a valuable complement to existing high-dimensional challenges in Bayesian deep learning.

![Figure 8. Testbed marginal performance $\tau = 1$ is not significantly correlated with sequential decision performance.](image)

![Figure 9. Testbed joint performance $\tau = 10$ is highly correlated with sequential decision performance.](image)

6. Conclusion

The Neural Testbed investigates the quality of predictive uncertainty in joint predictions, as well as marginals. With this simple and clear 2D challenge we aim to build understanding that can inform the field’s wider efforts in deep learning. We have shown that results on the testbed can offer new insights to agent development, and reveal the importance of joint predictions beyond marginals in the design of practical algorithms. Further, we establish that the insights gained in the testbed can scale up to complex and high-dimensional decision problems.

Beyond the results in this paper, we believe that this work can provide an exciting base for future research:

- Can we design better learning algorithms for joint predictions, as well as marginals?
- Is there a similar result for Figure 2 on challenge datasets?
- How can effective joint predictions drive better decisions?

We believe that studying these simple testbed problems can help foster interplay between theory and practice, improve accessibility in the field, and complement existing research. We hope that this will accelerate the growth of understanding in the field and, ultimately, drive forward the design of better learning agents.
The Neural Testbed

References

Band, N., Rudner, T. G., Feng, Q., Filos, A., Nado, Z., Dusenberry, M. W., Jerfel, G., Tran, D., and Gal, Y. Benchmarking Bayesian deep learning on diabetic retinopathy detection tasks. In NeurIPS 2021 Workshop on Distribution Shifts: Connecting Methods and Applications, 2021.

Bartlett, P. L., Montanari, A., and Rakhlin, A. Deep learning: a statistical viewpoint. arXiv preprint arXiv:2103.09177, 2021.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. Weight uncertainty in neural network. In International Conference on Machine Learning, pp. 1613–1622. PMLR, 2015.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Cover, T. and Hart, P. Nearest neighbor pattern classification. IEEE transactions on information theory, 13(1):21–27, 1967.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition, pp. 248–255. Ieee, 2009.

Dwaracherla, V., Lu, X., Ibrahimi, M., Osband, I., Wen, Z., and Van Roy, B. Hypermodels for exploration. In International Conference on Learning Representations, 2020. URL https://openreview.net/forum?id=ryx6WgStPB.

Friedman, J. H. The elements of statistical learning: Data mining, inference, and prediction. springer open, 2017.

Gal, Y. and Ghahramani, Z. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In International Conference on Machine Learning, 2016.

Gittins, J. C. Bandit processes and dynamic allocation indices. Journal of the Royal Statistical Society: Series B (Methodological), 41(2):148–164, 1979.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the 13th international conference on artificial intelligence and statistics, pp. 249–256, 2010.

He, B., Lakshminarayanan, B., and Teh, Y. W. Bayesian deep ensembles via the neural tangent kernel. arXiv preprint arXiv:2007.05864, 2020.

Hoffman, M. D., Gelman, A., et al. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. J. Mach. Learn. Res., 15(1):1593–1623, 2014.

Hron, J., Matthews, A. G. d. G., and Ghahramani, Z. Variational gaussian dropout is not Bayesian. arXiv preprint arXiv:1711.02989, 2017.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25, pp. 1097–1105, 2012.

Kullback, S. and Leibler, R. A. On information and sufficiency. The annals of mathematical statistics, 22(1):79–86, 1951.

Lakshminarayanan, B., Pritzel, A., and Blundell, C. Simple and scalable predictive uncertainty estimation using deep ensembles. In Advances in Neural Information Processing Systems, pp. 6405–6416, 2017.

Le Folgoc, L., Baltatzis, V., Desai, S., Devaraj, A., Ellis, S., Manzanera, O. E. M., Nair, A., Qiu, H., Schnabel, J., and Glocker, B. Is MC dropout Bayesian? arXiv preprint arXiv:2110.04286, 2021.

Lu, X. and Van Roy, B. Ensemble sampling. In Advances in Neural Information Processing Systems, pp. 3260–3268, 2017.

Lu, X., Osband, I., Van Roy, B., and Wen, Z. Evaluating probabilistic inference in deep learning: Beyond marginal predictions. CoRR, abs/2107.09224, 2021a. URL https://arxiv.org/abs/2107.09224.

Lu, X., Van Roy, B., Dwaracherla, V., Ibrahimi, M., Osband, I., and Wen, Z. Reinforcement learning, bit by bit. arXiv preprint arXiv:2103.04047, 2021b.

MacKay, D. J. A practical Bayesian framework for backpropagation networks. Neural computation, 4(3):448–472, 1992.

Murphy, K. P. Machine Learning: A Probabilistic Perspective. MIT Press, 2012.

Nado, Z., Band, N., Collier, M., Djolonga, J., Dusenberry, M., Farquhar, S., Filos, A., Havasi, M., Jenatton, R., Jerfel, G., Liu, J., Mariet, Z., Nixon, J., Padhy, S., Ren, J., Rudner, T., Wen, Y., Wenzel, F., Murphy, K., Sculley, D., Lakshminarayanan, B., Snoek, J., Gal, Y., and Tran, D. Uncertainty Baselines: Benchmarks for uncertainty & robustness in deep learning. arXiv preprint arXiv:2106.04015, 2021.

Neal, R. M. Bayesian learning for neural networks, volume 118. Springer Science & Business Media, 2012.
Osband, I. Risk versus uncertainty in deep learning: Bayes, bootstrap and the dangers of dropout. In NIPS Workshop on Bayesian Deep Learning, volume 192, 2016.

Osband, I. and Van Roy, B. Bootstrapped Thompson sampling and deep exploration. arXiv preprint arXiv:1507.00300, 2015.

Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. Deep exploration via bootstrapped DQN. In Advances in Neural Information Processing Systems 29, pp. 4026–4034, 2016.

Osband, I., Aslanides, J., and Cassirer, A. Randomized prior functions for deep reinforcement learning. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), Advances in Neural Information Processing Systems 31, pp. 8617–8629. Curran Associates, Inc., 2018. URL https://bit.ly/rpf_neurips.

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepesvari, C., Singh, S., et al. Behaviour suite for reinforcement learning. arXiv preprint arXiv:1908.03568, 2019.

Osband, I., Wen, Z., Asghari, M., Ibrahimi, M., Lu, X., and Van Roy, B. Epistemic neural networks. arXiv preprint arXiv:2107.08924, 2021.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

Rasmussen, C. E. Gaussian processes in machine learning. In Summer school on machine learning, pp. 63–71. Springer, 2003.

Recht, B., Roelofs, R., Schmidt, L., and Shankar, V. Do cifar-10 classifiers generalize to cifar-10? arXiv preprint arXiv:1806.00451, 2018.

Rezende, D. and Mohamed, S. Variational inference with normalizing flows. In International conference on machine learning, pp. 1530–1538. PMLR, 2015.

Riquelme, C., Tucker, G., and Snoek, J. Deep Bayesian bandits showdown: An empirical comparison of Bayesian deep networks for thompson sampling. arXiv preprint arXiv:1802.09127, 2018.

Russo, D. and Zou, J. Controlling bias in adaptive data analysis using information theory. In Artificial Intelligence and Statistics, pp. 1232–1240. PMLR, 2016.

Russo, D. J., Van Roy, B., Kazeroni, A., Osband, I., and Wen, Z. A tutorial on thompson sampling. Found. Trends Mach. Learn., 11(1):1–96, July 2018. ISSN 1935-8237.

Schölkopf, B. and Smola, A. J. Learning with kernels: Support vector machines, regularization, optimization, and beyond. MIT press, 2018.

Sharir, O., Peleg, B., and Shoham, Y. The cost of training nlp models: A concise overview. arXiv preprint arXiv:2004.08900, 2020.

Sun, S., Zhang, G., Shi, J., and Grosse, R. Functional variational Bayesian neural networks. arXiv preprint arXiv:1903.05779, 2019.

Thompson, W. R. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. Biometrika, 25(3/4):285–294, 1933.

Wang, C., Sun, S., and Grosse, R. Beyond marginal uncertainty: How accurately can Bayesian regression models estimate posterior predictive correlations? In International Conference on Artificial Intelligence and Statistics, pp. 2476–2484. PMLR, 2021.

Welling, M. and Teh, Y. W. Bayesian learning via stochastic gradient Langevin dynamics. In Proceedings of the 28th international conference on machine learning (ICML-11), pp. 681–688. Citeseer, 2011.

Wilson, A. G. and Izmailov, P. Bayesian deep learning and a probabilistic perspective of generalization. arXiv preprint arXiv:2002.08791, 2020.

Wilson, A. G., Izmailov, P., Hoffman, M. D., Gal, Y., Li, Y., Pradier, M. F., Vikram, S., Foong, A., Lotfi, S., and Farquhar, S. Evaluating approximate inference in Bayesian deep learning. 2021.

Woodbury, M. A. Inverting modified matrices. Statistical Research Group, 1950.
A. Testbed Pseudocode

We present the testbed pseudocode in this section. Specifically, Algorithm 2 is the pseudocode for our neural testbed, and Algorithm 3 is an approach to estimate the likelihood of a test data \( \tau \)-sample conditioned on an agent’s belief, based on the standard Monte-Carlo estimation. The presented testbed pseudocode works for any prior \( \mathbb{P}(\mathcal{E} \in \cdot) \) over the environment and any input distribution \( \mathbb{P}_X \), including the ones described in Section 3.1. We also release full code and implementations at github.com/deepmind/neural_testbed.

In addition to presenting the testbed pseudocode, we also explain our choices of experiment parameters in Appendix B. To apply Algorithm 2, we need to specify an input distribution \( \mathbb{P}_X \) and a prior distribution on the environment \( \mathbb{P}(\mathcal{E} \in \cdot) \). Recall from Section 3.1 that we consider binary classification problems with input dimension 2. We choose \( \mathbb{P}_X = \mathcal{N}(0, I) \), and we consider three environment priors distinguished by a temperature parameter that controls the signal-to-noise ratio (SNR) regime. We sweep over temperatures in \( \{0.01, 0.1, 0.5\} \). The prior distribution \( \mathbb{P}(\mathcal{E} \in \cdot) \) is induced by a distribution over MLPs with 2 hidden layers and ReLU activation. The MLP is distributed according to standard Xavier initialization, except that biases in the first layer are drawn from \( \mathcal{N}(0, \frac{1}{2}) \). The MLP outputs two units, which are divided by the temperature parameter and passed through the softmax function to produce class probabilities. The implementation of this generative model is in our open source code under the path /generative/factories.py.

We now describe the other parameters we use in the Testbed. In Algorithm 2, we pick the order of predictive distributions \( \tau \in \{1, 10, 30, 100, 300, 1000\} \), training dataset size \( T \in \{1, 3, 10, 30, 100, 300, 1000\} \), number of sampled problems \( J = 10 \), and number of testing data \( \tau \)-samples \( N = 1000 \). To apply Algorithm 3, we sample \( M = 1000 \) models from the agent.

B. Agents

In this section, we describe the benchmark agents in Section 3.3 and the choice of various hyperparameters used in the implementation of these agents. The list of agents include MLP, ensemble, dropout, Bayes by backprop, stochastic Langevin MCMC, ensemble+ and hypermodel. We will also include other agents such as KNN, random forest, and deep kernel, but the performance of these agents was worse than the other benchmark agents, so we chose not to include them in the comparison in Section 4. In each case, we attempt to match the “canonical” implementation. The complete implementation of these agents including the hyperparameter sweeps used for the Testbed are available at github.com/deepmind/neural_testbed. We make use of the Epistemic Neural Networks notation from (Osband et al., 2021) in our code. We set the default hyperparameters of each agent to be the ones that minimize the aggregated KL score \( d^1_{\text{KL}} = d^1_{\text{KL}} + \frac{1}{10} d^{10}_{\text{KL}} \).

B.1. MLP

The mlp agent learns a 2-layer MLP with 50 hidden units in each layer by minimizing the cross-entropy loss with \( L_2 \) weight regularization. The \( L_2 \) weight decay scale is chosen either to be \( \lambda^\frac{1}{T} \) or \( \lambda^\frac{d}{T} \), where \( d \) is the input dimension, \( \beta \) is the temperature of the generative process and \( T \) is the size of the training dataset. We sweep over \( \lambda \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\} \). We implement the MLP agent as a special case of a deep ensemble (B.2). The implementation and hyperparameter sweeps for the mlp agent can be found in our open source code, as a special case of the ensemble agent, under the path /agents/factories/ensemble.py.

B.2. Ensemble

We implement the basic “deep ensembles” approach for posterior approximation (Lakshminarayanan et al., 2017). The ensemble agent learns an ensemble of MLPs by minimizing the cross-entropy loss with \( L_2 \) weight regularization. The only difference between the ensemble members is their independently initialized network weights. We chose the \( L_2 \) weight scale to be either \( \lambda^\frac{1}{M^2} \) or \( \lambda^\frac{d}{M^2} \), where \( M \) is the ensemble size, \( d \) is the input dimension, \( \beta \) is the temperature of the generative process, and \( T \) is the size of the training dataset. We sweep over ensemble size \( M \in \{1, 3, 10, 30, 100\} \) and \( \lambda \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\} \). We find that larger ensembles work better, but this effect is within margin of error after 10 elements. The implementation and hyperparameter sweeps for the ensemble agent can be found in our open source code under the path /agents/factories/ensemble.py.
Algorithm 2 Neural Testbed

Require: the testbed requires the following inputs

1. prior distribution over the environment $P(E \in \cdot)$, input distribution $P_X$
2. agent $f_\theta$
3. number of training data $T$, test distribution order $\tau$
4. number of sampled problems $J$, number of test data samples $N$
5. parameters for agent likelihood estimation, as is specified in Algorithm 3

for $j = 1, 2, \ldots, J$ do

Step 1: sample environment and training data
1. sample environment $E \sim P(E \in \cdot)$
2. sample $T$ inputs $X_0, X_1, \ldots, X_{T-1}$ i.i.d. from $P_X$
3. sample the training labels $Y_1, \ldots, Y_T$ conditionally i.i.d. as $Y_{t+1} \sim P(Y \in \cdot | E, X_t = X_t)$ $\forall t = 0, 1, \ldots, T - 1$
4. choose the training dataset as $D_T = \{(X_t, Y_{t+1}), t = 0, \ldots, T - 1\}$

Step 2: train agent
train agent $f_\theta_T$ based on training dataset $D_T$

Step 3: compute likelihoods
for $n = 1, 2, \ldots, N$ do
1. sample $X^{(n)}_{T}, \ldots, X^{(n)}_{T+\tau-1}$ i.i.d. from $P_X$
2. generate $Y^{(n)}_{T+1}, \ldots, Y^{(n)}_{T+\tau}$ conditionally independently as $Y^{(n)}_{t+1} \sim P(Y \in \cdot | E, X_t = X^{(n)}_t)$ $\forall t = T, T+1, \ldots, T+\tau - 1$
3. compute the likelihood under the environment $E$ as $p_{j,n} = P\left(Y^{(n)}_{T+1:T+\tau} | E, X^{(n)}_{T:T+\tau-1}\right) = \prod_{t=T}^{T+\tau-1} \Pr\left(Y^{(n)}_{t+1} | E, X^{(n)}_t\right)$
4. estimate the likelihood conditioned on the agent’s belief $\hat{p}_{j,n} \approx P\left(\hat{Y}_{T+1:T+\tau} = Y^{(n)}_{T+1:T+\tau} | \theta_T, X^{(n)}_{T:T+\tau-1}, Y^{(n)}_{T+1:T+\tau}\right)$ based on Algorithm 3 with test data $\tau$-sample $\left(X^{(n)}_{T:T+\tau-1}, Y^{(n)}_{T+1:T+\tau}\right)$.

end for

return $\frac{1}{JN} \sum_{j=1}^{J} \sum_{n=1}^{N} \log \left(p_{j,n}/\hat{p}_{j,n}\right)$
The Neural Testbed

Algorithm 3 Monte Carlo Estimation of Likelihood of Agent’s Belief

Require: the Monte-Carlo estimation requires the following inputs

1. trained agent \( f_{\theta_T} \), and number of Monte Carlo samples \( M \)
2. test data \( \tau \)-sample \( (X_{T:T+\tau-1}, Y_{T+1:T+\tau}) \)

Step 1: sample \( M \) models \( \hat{\epsilon}_1, \ldots, \hat{\epsilon}_M \) conditionally i.i.d. from \( P \left( \hat{\epsilon} \in \cdot | \theta_T \right) \)

Step 2: estimate \( \hat{p} \) as

\[
\hat{p} = \frac{1}{M} \sum_{m=1}^{M} P \left( \hat{Y}_{T+1:T+\tau} = Y_{T+1:T+\tau} | \hat{\epsilon}_m, X_{T:T+\tau-1}, Y_{T+1:T+\tau} \right)
\]

return \( \hat{p} \)

B.3. Dropout

We follow (Gal & Ghahramani, 2016) to build a dropout agent for posterior approximation. The agent applies dropout on each layer of a fully connected MLP with ReLU activation and optimizes the network using the cross-entropy loss combined with \( L_2 \) weight decay. The \( L_2 \) weight decay scale is chosen to be either \( \frac{\lambda^2}{T} (1 - p_{\text{drop}}) \) or \( \frac{\lambda^2 p_{\text{drop}}}{T} \) where \( p_{\text{drop}} \) is the dropping probability, \( d \) is the input dimension, \( \beta \) is the temperature of the data generating process, and \( T \) is the size of the training dataset. We sweep over dropout rate \( p_{\text{drop}} \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\} \), length scale (used for \( L_2 \) weight decay) \( l \in \{1, 3, 10\} \), number of neural network layers \( \in \{2, 3\} \), and hidden layer size \( \in \{50, 100\} \). The implementation and hyperparameter sweeps for the dropout agent can be found in our open source code under the path /agents/factories/dropout.py.

B.4. Bayes-by-backprop

We follow (Blundell et al., 2015) to build a bbb agent for posterior approximation. We consider a scale mixture of two zero-mean Gaussian densities as the prior. The Gaussian densities have standard deviations \( \sigma_1 \) and \( \sigma_2 \), and they are mixed with probabilities \( p \) and \( 1-p \), respectively. We sweep over \( \sigma_1 \in \{0.3, 0.5, 0.7, 1, 2, 4\} \), \( \sigma_2 \in \{0.3, 0.5, 0.7\} \), \( p \in \{0, 0.5, 1\} \), learning rate \( \in \{10^{-3}, 3 \times 10^{-3}\} \), number of training steps \( \in \{1000, 2000\} \), number of neural network layers \( \in \{2, 3\} \), hidden layer size \( \in \{50, 100\} \), and the ratio of the complexity cost to the likelihood cost \( \in \{1, d \sqrt{\beta}\} \), where \( d \) is the input dimension and \( \beta \) is the temperature of the data generating process. The implementation and hyperparameter sweeps for the bbb agent can be found in our open source code under the path /agents/factories/bbb.py.

B.5. Stochastic gradient Langevin dynamics

We follow (Welling & Teh, 2011) to implement a sgmcmc agent using stochastic gradient Langevin dynamics (SGLD). We consider two versions of SGLD, one with momentum and other without the momentum. We consider independent Gaussian prior on the neural network parameters where the prior variance is set to be

\[
\sigma^2 = \lambda \frac{T}{d \sqrt{\beta}}
\]

where \( \lambda \) is a hyperparameter that is swept over \( \{0.0025, 0.01, 0.04\} \), \( d \) is the input dimension, \( \beta \) is the temperature of the data generating process, and \( T \) is the size of the training dataset. We consider a constant learning rate that is swept over \( \{10^{-3}, 5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}\} \). For SGLD with momentum, the momentum decay term is always set to be 0.9. The number of training batches is \( 5 \times 10^3 \) with burn-in time of \( 10^3 \) training batches. We save a model every 1000 steps after the burn-in time and use these models as an ensemble during the evaluation. The implementation and hyperparameter sweeps for the sgmcmc agent can be found in our open source code under the path /agents/factories/sgmcmc.py.

B.6. Ensemble+

We implement the ensemble+ agent using deep ensembles with randomized prior functions (Osband et al., 2018) and bootstrap sampling (Osband & Van Roy, 2015). Similar to the vanilla ensemble agent in Section B.2, we consider \( L_2 \) weight...
We use an additive prior which is a linear hypermodel prior over an MLP base model, which is similar to the generating hypermodel. We follow (Dwaracherla et al., 2020) to build a model.

A neural network takes input \( X_t \in \mathcal{X} \) and produces output \( Z_{t+1} = W \phi_b(X_t) + b \in \mathbb{R}^K \), where \( W \in \mathbb{R}^{K \times m} \) is a matrix, \( b \in \mathbb{R}^K \) is a bias vector, and \( \phi_b : \mathcal{X} \to \mathbb{R}^m \) is the output of the penultimate layer of the neural network. In the case of classification the output \( Z_{t+1} \) corresponds to the logits over the class labels, i.e., \( Y_{t+1} \propto \exp(Z_{t+1}) \). The neural network should learn a function that maps the input into a space where the classes are linearly distinguishable. In other words, the mapping that the neural network is learning can be considered a form of kernel (Schölkopf & Smola, 2018), where the kernel function \( k : \mathcal{X} \times \mathcal{X} \to \mathbb{R} \) is simply \( k(X, X') = \phi_b(X)^\top \phi_b(X') \). With this in mind, we can take a trained neural network and consider the learned mapping to be the kernel in a Gaussian process (GP) (Rasmussen, 2003), from which we can obtain approximate uncertainty estimates. Concretely, let \( \Phi_{0:t-1} \in \mathbb{R}^{T \times m} \) be the matrix corresponding to the \( \phi_b(X_t) \), \( t = 0, \ldots, T-1 \), vectors stacked row-wise and let \( \Phi_{T:T+t-1} \in \mathbb{R}^{T \times m} \) denote the same quantity for the test set. Fix index \( i \in \{0, \ldots, K-1\} \) to be a particular class index. A GP models the joint distribution over the dataset to be a multi-variate Gaussian, i.e.,

\[
\begin{bmatrix}
Z^{(i)}_{1:T} \\
Z^{(i)}_{T+1:T+t}
\end{bmatrix} \sim \mathcal{N}
\begin{bmatrix}
\mu^{(i)}_{1:T} \\
\mu^{(i)}_{T+1:T+t}
\end{bmatrix},
\begin{bmatrix}
\Sigma + \sigma^2 I & \Phi_{0:t-1} \Phi_{T:T+t-1}^\top \\
\Phi_{0:t-1} \Phi_{T:T+t-1} & \Phi_{T:T+t-1} \Phi_{T:T+t-1}^\top
\end{bmatrix}
\]

where \( \sigma > 0 \) models the noise in the training data measurement and \( \mu^{(i)}_{1:T}, \mu^{(i)}_{T+1:T+t} \) are the means under the GP. The conditional distribution is given by

\[
P(Z^{(i)}_{T+1:T+t} \mid Z^{(i)}_{1:T}, X_{0:t-1}) = \mathcal{N}
\begin{bmatrix}
\mu^{(i)}_{T+1:T+t} \\
\Sigma_{T+1:T+t|1:T}
\end{bmatrix}
\]
where
\[ \Sigma_{T+1:T+\tau\mid 0:T} = \Phi_{T:T+\tau-1}^T \Phi_{T:T+\tau-1} - \Phi_{T:T+\tau-1}^T \Phi_{0:T-1}^T (\sigma^2 I + \Phi_{0:T-1}^T \Phi_{0:T-1})^{-1} \Phi_{0:T-1} \Phi_{T:T+\tau-1}. \]

and rather than use the GP to compute \( \mu_{T+1:T+\tau\mid 0:T}^{(i)} \) (which would not be possible since we do not observe the true logits) we just take it to be the output of the neural network when evaluated on the test dataset. The matrix being inverted in the expression for \( \Sigma_{T+1:T+\tau\mid 0:T} \) has dimension \( T \times T \), which may be quite large. We use the Sherman-Morrison-Woodbury identity to rewrite it as follows (Woodbury, 1950)
\[
\Sigma_{T+1:T+\tau\mid 0:T} = \Phi_{T:T+\tau-1}^T (I - \Phi_{0:T-1}^T (\sigma^2 I + \Phi_{0:T-1}^T \Phi_{0:T-1})^{-1} \Phi_{0:T-1}) \Phi_{T:T+\tau-1}^{-1}
= \sigma^2 \Phi_{T:T+\tau-1}^T (\sigma^2 I + \Phi_{0:T-1}^T \Phi_{0:T-1})^{-1} \Phi_{T:T+\tau-1}^{-1},
\]

which instead involves the inverse of an \( m \times m \) matrix, which may be much smaller. If we perform a Cholesky factorization of positive definite matrix \( (\sigma^2 I + \Phi_{0:T-1}^T \Phi_{0:T-1}) = LL^\top \) then the samples for all logits simultaneously can be drawn by first sampling \( \zeta \in \mathbb{R}^{m \times K} \), with each entry drawn IID from \( \mathcal{N}(0, 1) \), then forming
\[
\hat{Y}_{T+1:T+\tau} \propto \exp(\mu_{T+1:T+\tau\mid 1:T} + \sigma \Phi_{T:T+\tau-1} L^{-\top} \zeta).
\]

The implementation and hyperparameter sweeps for the deep_kernel agent can be found in our open source code under the path /agents/factories/deep_kernel.py.

### B.10. Other agents

In our paper we have made a concerted effort to include representative and canonical agents across different families of Bayesian deep learning and adjacent research. In addition to these implementations, we performed extensive tuning to make sure that each agent was given a fair shot. However, with the proliferation of research in this area, it was not possible for us to evaluate all competing approaches. We hope that, by opensourcing the Neural Testbed, we can allow researchers in the field to easily assess and compare their agents to these baselines.

For example, we highlight a few recent pieces of research that might be interesting to evaluate in our setting. Of course, there are many more methods to compare and benchmark. We leave this open as an exciting area for future research.

- **Neural Tangent Kernel Prior Functions** (He et al., 2020). Proposes a specific type of prior function in ensemble+ inspired by connections to the neural tangent kernel.
- **Functional Variational Bayesian Neural Networks** (Sun et al., 2019). Applies variational inference directly to the function outputs, rather than weights like bbb.
- **Variational normalizing flows** (Rezende & Mohamed, 2015). Applies variational inference over a more expressive family than bbb.
- **No U-Turn Sampler** (Hoffman et al., 2014). Another approach to sgmcmc that attempts to compute the posterior directly, computational costs can grow large.

### C. Testbed results

In this section, we provide the complete results of the performance of benchmark agents on the Testbed, broken down by the temperature setting, which controls the SNR, and the size of the training dataset. We select the best performing agent, based on aggregated score \( d_{KL}^{10} + d_{KL}^{10}/10 \), within each agent family and plot \( d_{KL}^{1} \) and \( d_{KL}^{10} \) with the performance of an MLP agent as a reference. We also provide a plot comparing the training time of different agents.

#### C.1. Performance breakdown

Figures 10 and 11 show the KL estimates evaluated on \( \tau = 1 \) and \( \tau = 10 \), respectively. For each agent, for each SNR regime, for each number of training points we plot the average KL estimate from the Testbed. In each plot, we include the “baseline” mlp agent as a black dashed line to allow for easy comparison across agents. A detailed description of these benchmark agents can be found in Appendix B.
C.2. Training time

Figure 12 shows a plot comparing the $d_{10}^{KL}$ and training time of different agents normalized with that of an MLP. The parameters of each agent are selected to maximize the $d_{10}^{KL}$. We can see that sgmcmc agent has the best performance, but at the cost of more training time (computation). Both ensemble+ and hypermodel agents have similar performance as sgmcmc with lower training time. We trained our agents on CPU only systems.

D. Sequential Decision Problems

This section provides supplementary information for the sequential decision problems in Section 5. All of the code necessary to reproduce the experiments is opensourced in the /bandit/ directory.

D.1. Problem formulation

We consider bandit problems derived from the testbed and evaluate the agents using Algorithm 4 for which we need to specify prior on the environment $P(E \in \cdot)$, input distribution $P_X$, and the number of actions $N$. We choose input distribution $P_X = \mathcal{N}(0, I_d)$, where $d$ is the input dimension. We sweep over $d \in \{2, 10, 50\}$ and choose the number of actions to be $N = 20d$, i.e., for input dimensions $\{2, 10, 50\}$ we have $\{40, 200, 1000\}$ actions respectively. We use the same prior distribution of environments as in Appendix A with a fixed temperature of $0.1$. For each setting, we run for 50,000 time steps ($T = 50,000$) and with 20 random seeds ($J = 20$).

D.2. Agent definition

In Appendix B, we described benchmark agents in our testbed. Among these agents, we use mlp, ensemble, dropout, bbb, ensemble+, and hypermodel agents for sequential decision problems. For all the agents we use the hyperparameters specified by default, in the source code, at the path /agents/factories/. The default hyperparameters of each agent correspond to be the ones that minimize the aggregated KL score $d_{agg}^{KL} = d_{KL}^1 + d_{KL}^{10}/10$. As the agent interacts with the environment, the amount of data the agent has observed keeps growing. Due to this we tune the regularization term based on the number of time steps agent has interacted with the environment. For mlp, ensemble, ensemble+, and hypermodel agents we use an $L_2$ weight decay of $\lambda \frac{2 \sqrt{\beta t}}{t}$, where $\beta$ is the temperature, $t$ is the number of the time steps the agent has interacted with the environment, and $\lambda$ is the default weight scale of the agent. For dropout we choose the $L_2$ weight decay as $\frac{\sqrt{\beta l}}{l}$, where $l$ is the default length scale used in the dropout agent. For bbb we scale the prior term by $\frac{1}{\tau}$. As described above, all hyperparameters are chosen to be the ones which minimize the aggregated KL score $d_{agg}^{KL} = d_{KL}^1 + \frac{1}{10}d_{KL}^{10}$.

D.3. Results

Figures 8 and 9 shows the correlation between performance on testbed performance and sequential decision problems with an input dimension of 50. Different points of an agent in these figures corresponds to different random seeds for the testbed and sequential problems. We can see that performance on sequential decision problems is strongly correlated with testbed joint performance $\tau = 10$ and not correlated with the testbed marginal performance. In Figures 13 and 14 we show a similar correlation plots between testbed performance and sequential decision problems across different input dimensions for sequential decision problems. We can see that performance on sequential decision problems has clear correlation with testbed joint performance $\tau = 10$, and no correlation with testbed marginal performance $\tau = 1$, across all the input dimensions considered.

These results offer empirical evidence that practical deep learning approaches separated by the quality of their joint predictions, but not their marginals, can lead to differing performance in downstream tasks. In addition, we show that our simple 2D testbed can provide insights that scale to much higher dimension problems.
Algorithm 4 Evaluation on Bandit Problem

Require: Evaluation on bandit problem requires the following inputs

1. Distribution over the environment $P(E \in \cdot)$, input distribution $P_X$, and the number of actions $N$.
2. Agent $f_\theta$
3. Number of time steps $T$
4. Number of sampled problems $J$

for $j = 1, 2, \ldots, J$ do

Step 1: Sample environment and action set
1. Sample environment $\hat{E} \sim P(\hat{E} \in \cdot)$
2. Sample a set $\mathcal{X}$ of $N$ actions $x_1, x_2, \ldots, x_N$ i.i.d. from $P_X$
3. Obtain the mean rewards corresponding to actions in $\mathcal{X}$ conditioned on the environment
   \[
   \overline{R}_x = P(Y_{t+1} = 1|\hat{E}, X_t = x), \quad \forall x \in \mathcal{X}
   \]
4. Compute the optimal expected reward $\overline{R}_* = \max_{x \in \mathcal{X}} \overline{R}_x$

Step 2: Agent interaction with the environment
Initialize the data buffer $D_0 = \emptyset$

for $t = 1, 2, \ldots, T$ do
1. Update agent $f_{\theta_t}$ belief distribution based on the data in the buffer $D_{t-1}$
2. TS action selection scheme:
   i. Sample $\hat{E}_t$ from the agent belief distribution
      \[
      \hat{E}_t \sim P(\hat{E} \in \cdot|\theta_t)
      \]
   ii. Act greedily based on $\hat{E}_t$
      \[
      X_t \in \arg \max_{x \in \mathcal{X}} P(Y_{t+1} = 1|\hat{E}_t, X_t = x)
      \]
   iii. Generate observation $Y_{t+1}$ based on action $X_t$
      \[
      Y_{t+1} \sim P(Y_{t+1} \in \cdot|\hat{E}, X_t = X_t)
      \]
3. Update the buffer $D_t = D_0 \cup (X_t, Y_{t+1})$
end for

Compute the total regret incurred in $T$ time steps
\[
\text{Regret}_j(T) = \sum_{t=1}^{T} (\overline{R}_* - \overline{R}_{X_t})
\]

end for

return $\frac{1}{J} \sum_{j=1}^{J} \text{Regret}_j(T)$
Figure 10. Performance of benchmark agents on the Testbed evaluated on $\tau = 1$, compared against the MLP baseline.
Figure 11. Performance of benchmark agents on the Testbed evaluated on $\tau = 10$, compared against the MLP baseline.
The Neural Testbed

**Figure 12.** Normalized $d_{KL}^{10}$ vs training time of different agents

**Figure 13.** Testbed marginal performance $d_{KL}^{1}$ is not significantly positively correlated with sequential decision performance. This result is robust across input dimensions 2, 10, and 50.

**Figure 14.** Testbed joint performance $d_{KL}^{10}$ is significantly positively correlated with sequential decision performance. This result is robust across input dimensions 2, 10, and 50.