Efficient Online Bayesian Inference for Neural Bandits

Gerardo Duran-Martin, Aleyna Kara and Kevin Murphy
Queen Mary University Boğaziçi University Google Research

Abstract

In this paper we present a new algorithm for online (sequential) inference in Bayesian neural networks, and show its suitability for tackling contextual bandit problems. The key idea is to combine the extended Kalman filter (which locally linearizes the likelihood function at each time step) with a (learned or random) low-dimensional affine subspace for the parameters; the use of a subspace enables us to scale our algorithm to models with $\sim 1M$ parameters. While most other neural bandit methods need to store the entire past dataset in order to avoid the problem of "catastrophic forgetting", our approach uses constant memory. This is possible because we represent uncertainty about all the parameters in the model, not just the final linear layer. We show good results on the "Deep Bayesian Bandit Showdown" benchmark, as well as MNIST and a recommender system.

1 Introduction

Contextual bandit problems (see e.g., [LS19; Sli19]) are a special case of reinforcement learning, in which the state (context) at each time step is chosen independently, rather than being dependent on the past history of states and actions. Despite this limitation, contextual bandits are widely used in real-world applications, such as recommender systems [Li+10; Guo+20], advertising [McM+13; Du+21], healthcare [Gre+17; AKR21], etc. The goal is to maximize the sequence of rewards y_t obtained by picking actions a_t in response to each input context or state $\boldsymbol{s}_t$. To do this, the decision making agent must learn a reward model $\mathbb{E}\left[y_t|\boldsymbol{s}_t, a_t, \boldsymbol{\theta}\right] = f(\boldsymbol{s}_t, a_t; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the unknown model parameters. Unlike supervised learning, the agent does not get to see the "correct" output, but instead only gets feedback on whether the choice it made was good or bad (in the form of the reward signal). If the agent knew $\boldsymbol{\theta}$, it could pick the optimal action using $a_t^* = \text{argmax}_{a \in \mathcal{A}} f(\boldsymbol{s}_t, a; \boldsymbol{\theta})$. However, since $\boldsymbol{\theta}$ is unknown, the agent must "explore", so it can gather information about the reward function, before it can "exploit" its model.

In the bandit literature, the two most common solutions to solving the explore-exploit dilemma are based on the upper confidence bound (UCB) method (see e.g., [Li+10; KCG12]) and the Thompson Sampling (TS) method (see e.g., [AG13; Rus+18]). The key bottleneck in both UBC and TS is efficiently computing the posterior $p(\boldsymbol{\theta}|\mathcal{D}_{1:t})$ in an online fashion, where $\mathcal{D}_{1:t} = \{(\boldsymbol{s}_i, a_i, y_i) : i = 1 : t\}$ is all the data seen so far. This can be done in closed form for linear-Gaussian models, but for nonlinear models, such as deep neural networks (DNNs), it is computationally infeasible.

In this paper, we propose to use a version of the extended Kalman filter to recursively approximate the parameter posterior $p(\boldsymbol{\theta}|\mathcal{D}_{1:t})$ using constant time and memory (i.e., independent of T). The main novelty of our approach is that we show how to scale the EKF to large neural networks by leveraging recent results that show that deep neural networks often have very few "degrees of freedom" (see e.g., [Li+18; Izm+19; Lar+21]). Thus we can compute a low-dimensional subspace and perform Bayesian filtering in the subspace rather than the original parameter space. We therefore call our method "Bayesian subspace bandits".

Although Bayesian inference in DNN subspaces has previously been explored (see related work in Section 2), it has not been done in an online or bandit setting, as far as we know. Since we are using approximate inference, we lose the well-known optimality of Thompson sampling [PAYD19]; we leave proving regret bounds for our method to future work. In this paper, we restrict attention to an empirical comparison. We show that our method works well in practice on various datasets, including the "Deep Bayesian Bandits Showdown" benchmark [RTS18], the MNIST dataset, and a recommender

Proceedings of the 25th International Conference on Artificial Intelligence and Statistics (AISTATS) 2022, Valencia, Spain. PMLR: Volume 151.

system dataset. In addition, our method uses much less memory and time than most other methods.

Our algorithm is not specific to bandits, and can be applied to any situation that requires efficient online computation of the posterior. This includes tasks such as life long learning, Bayesian optimization, active learning, reinforcement learning, etc.[1]. However, we leave such extensions to future work.

2 Related work

In this section, we briefly review related work. We divide the prior work into several groups: Bayesian neural networks, neural net subspaces, and neural contextual bandits.

Most work on Bayesian inference for neural networks has focused on the offline (batch) setting. Common approaches include the Laplace approximation [Mac92; Mac95; Dax+21a]; Hamiltonian MCMC [Nea95; Izm+21]; variational inference, such as the "Bayes by backprop" method of [Blu+15], and the "variational online Gauss-Newton" method of [Osa+19]; expectation propagation, such as the "probabilistic backpropagation" method of [HLA15]; and many others. (For more details and references, see e.g., [PS17; Wil20; WI20; Kha20].)

There are several techniques for online or sequential Bayesian inference for neural networks. [RBB18] propose an online version of the Laplace approximation, [Ngu+18] propose an online version of variational inference, and [GDFY16] propose to use assumed density filtering (an online version of expectation propagation). However, in [RTS18], they showed that these methods do not work very well for bandit problems. In this paper, we build on older work, specifically [SW89; FNG00], which used the extended Kalman filter (EKF) to perform approximate online inference for DNNs. We combine this with subspace methods to scale to high dimensions, as we discuss below.

There are several techniques for scaling Bayesian inference to neural networks with many parameters. A simple approach is to use variational inference with a diagonal Gaussian posterior, but this ignores important correlations between the weights. It is also possible to use low-rank factorizations of the posterior covariance matrix. In [Dax+21b], they propose to use a MAP estimate for some parameters and a Laplace approximation for others. However, their computation of the MAP estimate relies on standard offline SGD (stochastic gradient descent), whereas we perform online Bayesian inference without using SGD. In [Izm+19], they compute a linear subspace of dimension d by applying PCA to the last L iterates of stochastic weight averaging [Izm+18]; they then perform slice sampling in this low-dimensional subspace. In this paper, we also leverage subspace inference, but we do so in the online setting, which is necessary when solving bandit problems.

The literature on contextual bandits is vast (see e.g., [LS19; Sli19]). Here we just discuss recent work which utilizes DNNs to model the reward function, combined with Thompson sampling as the policy for choosing the action. In [RTS18], they evaluated many different approximate inference methods for Bayesian neural networks on a set of benchmark contextual bandit problems; they called this the "Deep Bayesian Bandits Showdown". The best performing method in their showdown is what they call the "neural linear" method, which we discuss in Section 3.3.

Unfortunately the neural linear method is not a fully online algorithm, since it needs to keep all the past data to avoid the problem of "catastrophic forgetting" [Rob95; Fre99; Kir+17]. This means that the memory complexity is $O(T)$, and the computational complexity can be as large as $O(T^2)$. This makes the method impractical for applications where the data is high dimensional, and/or the agent is running for a long time. In [NZM21], they make an online version of the neural linear method which they call "Lim2", which stands for "Limited Memory Neural-Linear with Likelihood Matching". We discuss this in more detail in Section 3.3.

More recently, several methods based on neural tangent kernels (NTK) have been developed [JGH18], including neural Thompson sampling [Zha+21] and neural UCB [ZLG20]. We discuss these methods in more detail in Section 3.3. Although Neural-TS and Neural-UCB in principle achieve a regret of $O(\sqrt{T})$, in practice there are some disadvantages. First, these algorithms perform multiple gradient steps, based on all the past data, at each step of the algorithm. Thus these are full memory algorithms that take $O(T)$ space and $O(T^2)$ time. Second, it can be shown [AZL19; Gho+20] that NTKs are less data efficient learners than (finite width) hierarchical DNNs, both in theory and in practice. Indeed we will show that our approach, that uses constant memory and finite width DNNs, does significantly better in practice.

[1] These problems are all very closely related. For example, BayesOpt is a kind of (non-contextual) bandit problem with an infinite number of arms; the goal is to identify the action (input to the reward function $f : \mathbb{R}^D \to \mathbb{R}$) that maximizes the output. Active learning is closely related to BayesOpt, but now the actions correspond to choosing data points $\boldsymbol{x} \in \mathbb{R}^n$ that we want to label, and our objective is to minimize uncertainty about the underlying function f, rather than find the location of its maximum.

3 Methods

In this section, we discuss various methods for tackling bandit problems, including our proposed new method.

3.1 Algorithmic framework

Algorithm 1: Online-Eval(Agent, Env, T, τ)

$\mathcal{D}_\tau$ = Environment.Warmup(τ);
$\boldsymbol{b}_\tau$ = Agent.InitBelief($\mathcal{D}_\tau$) ;
$R = 0$ // cumulative reward ;
for $t = (\tau + 1) : T$ **do**
  $\boldsymbol{s}_t$ = Environment.GetState(t) ;
  a_t = Agent.ChooseAction($\boldsymbol{b}_{t-1}, \boldsymbol{s}_t$) ;
  y_t = Environment.GetReward($\boldsymbol{s}_t, a_t$) ;
  $R+ = y_t$;
  $\mathcal{D}_t = (\boldsymbol{s}_t, a_t, y_t)$;
  $\boldsymbol{b}_t$ = Agent.UpdateBelief($\boldsymbol{b}_{t-1}, \mathcal{D}_t$);
Return R

In Algorithm 1, we give the pseudocode for a way to estimate the expected reward for a bandit policy (agent), given access to an environment or simulator. In the case of a Thompson sampling agent, the action selection is usually implemented by first sampling a parameter vector from the posterior (belief state), $\tilde{\boldsymbol{\theta}}_t \sim p(\boldsymbol{\theta}_t|\mathcal{D}_{1:t-1})$, and then predicting the reward for each action and greedily picking the best, $a_t = \text{argmax}_{a\in\mathcal{A}} \mathbb{E}\left[y|\boldsymbol{s}_t, a_t, \tilde{\boldsymbol{\theta}}_t\right]$. In the case of a UCB agent, the action is chosen by first computing the posterior predicted mean and variance, and then picking the action with the highest optimistic estimate of reward:

$$p_{t|t-1}(y|\boldsymbol{s}, a) \triangleq \int p(y|\boldsymbol{s}, a, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{D}_{1:t-1}) d\boldsymbol{\theta} \quad (1)$$

$$\mu_a = \mathbb{E}_{p_{t|t-1}}[y|\boldsymbol{s}_t, a] \quad (2)$$

$$\sigma_a = \sqrt{\mathbb{V}_{p_{t|t-1}}[y|\boldsymbol{s}_t, a]} \quad (3)$$

$$a_t = \underset{a\in\mathcal{A}}{\text{argmax}}\, \mu_a + \alpha\sigma_a \quad (4)$$

where $\alpha > 0$ is a tuning parameter that controls the degree of exploration. In this paper, we focus on Thompson sampling, but our methods can be extended to UCB in a straightforward way.

Since the prior on the parameters is usually uninformative, the initial actions are effectively random. Consequently we let the agent have a "warmup period", in which we systematically try each action N_w times, in a round robin fashion, for a total of $\tau = N_a \times N_w$ steps. We then use this warmup data to initialize the belief state to get an informative prior. If we have a long warmup period, then we will have a better initial estimate, but we may incur high regret during this period, since we are choosing actions "blindly". Thus we can view τ as a hyperparameter of the algorithm. The optimal value will depend on the expected lifetime T of the agent (if T is large, we can more easily amortize the cost of a long warmup period).

3.2 Modeling assumptions

We will assume a Gaussian bandit setting, in which the observation model for the reward is a Gaussian with a fixed or inferred observation variance: $p(y_t|\boldsymbol{s}_t, a_t) = \mathcal{N}(y_t|f(\boldsymbol{s}_t, a_t; \boldsymbol{\theta}_t), \sigma^2)$. (We discuss extensions to the Bernoulli bandit case in Section 5.)

Many current bandit algorithms assume the reward function is a linear model applied to a set of learned features. That is, it has the form $f(\boldsymbol{s}, a; \boldsymbol{\theta}) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{s}; \mathbf{V})$, where $\boldsymbol{\phi}(\boldsymbol{s}; \mathbf{V}) \in \mathbb{R}^{N_z}$ is the hidden state computed by a feature extractor, $\mathbf{V} \in \mathbb{R}^{D_b}$ are the parameters of this feature extractor "body", and $\mathbf{W} \in \mathbb{R}^{N_z \times N_a}$ is the final linear layer, with one output "head" per action. For example, in Figure 1a, we show a 2 layer model where $\boldsymbol{\phi}(\boldsymbol{s}; \mathbf{V}) = \text{ReLU}(\mathbf{V}_2\, \text{ReLU}(\mathbf{V}_1 \boldsymbol{s}))$ is the feature vector, and $\mathbf{V}_1 \in \mathbb{R}^{N_h^{(1)} \times N_s}$ and $\mathbf{V}_2 \in \mathbb{R}^{N_h^{(2)} \times N_h^{(1)}}$ are the first and second layer weights. (We ignore the bias terms for simplicity.) Thus $N_z = N_h^{(2)}$ is the size of the feature vector that is passed to the final linear layer. If the feature vector is fixed (i.e., is not learned), so $\boldsymbol{\phi}(\boldsymbol{s}) = \boldsymbol{s}$, we get a linear model of the form $f(\boldsymbol{s}, a; \boldsymbol{w}) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{s}$.

An alternative model structure is to concatentate the state vector, $\boldsymbol{\phi}(\boldsymbol{s}_t)$, with the action vetcor, $\boldsymbol{\phi}(a_t)$ to get an input of the form $\boldsymbol{x}_t = (\boldsymbol{\phi}(\boldsymbol{s}_t), \boldsymbol{\phi}(a_t))$. This is shown in Figure 1b. This can be useful if we have many possible actions; in this case, we can represent arms in terms of their features instead of their indices, just as we represent states in terms of their features. In this formulation, the linear output layer returns the predicted reward for the specified $(\boldsymbol{s}, \boldsymbol{a})$ input combination, and we require N_a forwards passes to evaluate the reward vector for each possible action.

Instead of concatenating the state and action vectors, we can compute their outer product and then flatten the result, to get $\boldsymbol{x}_t = \text{flatten}(\boldsymbol{\phi}(\boldsymbol{s}_t)\boldsymbol{\phi}(a_t)^\mathsf{T})$. This can model interaction effects, an proposed in [Li+10]. If $\boldsymbol{\phi}(a_t)$ is a one-hot encoding, we get the block-structured input $\boldsymbol{x}_t = (\mathbf{0}, \cdots, \mathbf{0}, \boldsymbol{\phi}(\boldsymbol{s}_t), \mathbf{0}, \cdots, \mathbf{0})$, where we insert the state feature vector into the block corresponding to the chosen action (see Figure 1c). This approach is used by recent NTK methods. If we assume $\boldsymbol{\phi}(\boldsymbol{s}) = \boldsymbol{s}$, so the state features are fixed, and we assume that the MLP has no hidden layers, then this model becomes equivalent to the linear model, since $\boldsymbol{w}^\mathsf{T} \boldsymbol{x}_t = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{s}_t$.

3.3 Existing methods

In this section, we briefly describe existing inference methods that we will compare to. More details on all

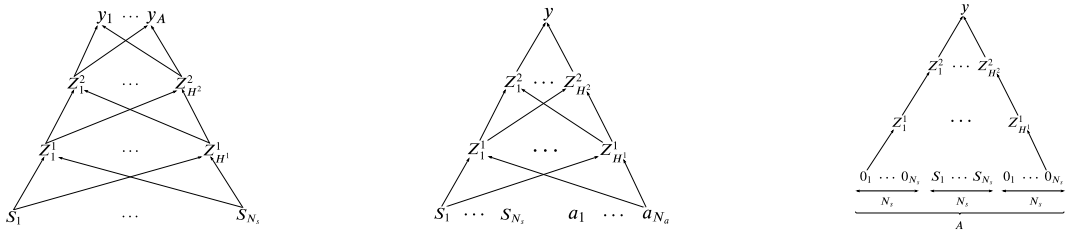


Figure 1: Illustration of some common MLP architectures used in bandit problems. $\boldsymbol{s}$ represents the state (context) vector, $\boldsymbol{a}$ represents the action vector, $\boldsymbol{y}$ represents the reward vector (for each possible action), and z_i^l is the i'th hidden node in layer l. (a) The input is $\boldsymbol{s}$, and there are A output "heads", $y_1, \ldots, y_A$, one per action. (b) The input is a concatentation of $\boldsymbol{s}$ and $\boldsymbol{a}$; the output is the predicted reward for this $(\boldsymbol{s}, \boldsymbol{a})$ combination. (c) The input is a block structured vector, where we insert $\boldsymbol{s}$ into the a'th block (when evaluating action a), and the remaining input blocks are zero.

methods can be found in the Supplementary Information. These methods differ in the kind of belief state they use to represent uncertainty about the model parameters, and in their mechanism for updating this belief state. See Table 1 for a summary.

Linear method The most common approach to bandit problems is to assume a linear model for the expected reward, $f(\boldsymbol{s}, a; \boldsymbol{\theta}) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{s}$. If we use a Gaussian prior, and assume a Gaussian likelihood, then we can represent the belief state as a Gaussian, $\boldsymbol{b}_t = \{(\boldsymbol{\mu}_{t,a}, \boldsymbol{\Sigma}_{t,a}) : a = 1 : N_a\}$. This can be efficiently updated online using the recursive least squares algorithm, which is a special case of the Kalman filter (see Appendix A.1 for details).

Neural linear method In [RTS18], they proposed a method called "neural linear", which they showed outperformed many other more sophisticated approaches, such as variational inference, on their bandit showdown benchmark. It assumes that the reward model has the form $f(\boldsymbol{s}, a; \boldsymbol{\theta}) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{s}; \mathbf{V}) \in \mathbb{R}^{N_z}$ is the hidden state computed by a feature extractor (see Figure 1a for an illustration). The neural linear method computes a point estimate of $\mathbf{V}$ by using SGD, and uses Bayesian linear regression to update the posterior over each $\boldsymbol{w}_a$, and optionally σ^2.

If we just update $\mathbf{V}$ at each step using $\mathcal{D}_t$, we run the risk of "catastrophic forgetting" (see Section 2). The standard solution to this is to store all the past data, and to re-run (minibatch) SGD on all the data at each step. Thus the belief state is represented as $\boldsymbol{b}_t = (\boldsymbol{\theta}_t, \mathcal{D}_{1:t})$. See Appendix A.2 for details.

The time cost is $O(T^2 N_e C_f)$, where N_e is the number of epochs (passes over the data) at each step, and C_f is the cost of a single forwards-backwards pass through the network (needed to compute the objective and its gradient).[2] Since it is typically too expensive to run SGD on each step, we can just perform updating every T_u steps. The total time then becomes $O(T' T N_e C_f)$, where $T' = T/T_u$ is the total number of times we invoke SGD.

The memory cost is $O(D + TN_x)$, where N_x is the size of each input example, $\boldsymbol{x}_t = (\boldsymbol{s}_t, a_t)$. If we limit the memory to the last M observations (also called a "replay buffer"), the memory reduces to $O(D + MN_x)$, and the time reduces to $O(T' M N_e C_f)$. However, naively limiting the memory in this way can hurt (statistical) performance, as we will see.

LiM2 In [NZM21], they propose a method called "LiM2", which stands for "Limited Memory Neural-Linear with Likelihood Matching". This is an extension of the neural linear method designed to solve the "catastrophic forgetting" that occurs when using a fixed memory buffer. The basic idea is to approximate the covariance of the old features in the memory buffer before replacing them with the new features, computed after updating the network parameters. This old covariance can be used as a prior during the Bayesian linear regression step.

Computing the updated prior covariance requires solving a semi-definite program (SDP) after each SGD step. In practice, the SDP can be solved using an inner loop of projected gradient descent (PGD), which involves solving an eigendecomposition at each step. This takes $O(T' M N_e N_p (C_f + N_z^3))$ time, where N_p is the number of PGD steps per SGD step. See Appendix A.3 for details.

NTK methods In [Zha+21], they propose a method called "Neural Thompson Sampling", and in [ZLG20], the propose a related method called "neural UCB". Both methods are based on approximating the MLP with a neural tangent kernel or NTK [JGH18]. Specifically, the feature vector at time t is defined to be $\boldsymbol{\phi}_t(\boldsymbol{s}, a) =$

[2] The reason for the quadratic cost is that each epoch passes over $O(T)$ examples, even if we use minibatching.

Method	Belief state	Memory	Time
Linear	$(\boldsymbol{\mu}_{t,a}, \boldsymbol{\Sigma}_{t,a})$	$O(N_a N_z^2)$	$O(T(C_f + N_a N_x^3))$
Neural-Greedy	$\boldsymbol{\theta}_t = (\mathbf{V}_t, \mathbf{W}_t)$	$O(D_b + N_a N_z + T N_x)$	$O(T' T N_e C_f)$
Neural-Linear	$(\mathbf{V}_t, \boldsymbol{\mu}_{t,a}, \boldsymbol{\Sigma}_{t,a}, \mathcal{D}_{1:t})$	$O(D_b + N_a N_z^2 + T N_x)$	$O(T' T N_e C_f + T N_a N_z^3)$
LiM2	$(\mathbf{V}_t, \boldsymbol{\mu}_{t,a}, \boldsymbol{\Sigma}_{t,a}, \mathcal{D}_{t-M:t})$	$O(D_b + N_a N_z^2 + M N_x)$	$O(T' M N_e N_p (C_f + N_z^3) + T N_a N_z^3)$
Neural-Thompson	$(\boldsymbol{\theta}_t, \boldsymbol{\Sigma}_t, \mathcal{D}_{1:t})$	$O(D + D^2 + T N_x)$	$O(T(T N_e C_f + D^3))$
EKF	$(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$	$O(D^2)$	$O(T(C_f + D^3))$
EKF-Subspace	$(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t, \boldsymbol{\theta}_*, \mathbf{A})$	$O(d^2 + Dd)$	$O(T(C_f + d^3 + Dd))$

Table 1: Summary of the methods for Bayesian inference considered in this paper. Notation: T: num steps taken by the agent in the environment; T_u: update frequency for SGD; $T' = T/T_u$: total num. times that we invoke SGD; N_e: num. epochs over the training data for each run of SGD; C_f: cost of to evaluate gradient of the network on one example, N_a: num. actions; N_x: size of input feature vector for state and action. N_z: num. features in penultimate (feature) layer; D_b: num. parameters in the body (feature extractor); $D_h = N_a N_z$: num. parameters in final layer linear; $D = D_b + D_h$: total num. parameters; d: size of subspace; M: size of memory buffer;

$(1/\sqrt{N_h})\nabla_{\boldsymbol{\theta}} f(\boldsymbol{s}, a)|_{\boldsymbol{\theta}_{t-1}}$, where N_h is the width of each hidden layer, and the gradient is evaluated at the most recent parameter estimate. They use a linear Gaussian model on top of these features. The network parameters are re-estimated at each step based on all the past data, and then the method effectively performs Bayesian linear regression on the output layer (see Appendix A.4 for details).

3.4 Our method: Subspace EKF

A natural alternative to just modeling uncertainty in the final layer weights is to "be Bayesian" about *all* the network parameters. Since our model is nonlinear, we must use approximate Bayesian inference. In this paper we choose to use the Extended Kalman Filter (EKF), which is a popular deterministic inference scheme for nonlinear state-space models based on linearizing the model (see Appendix A.5 for details). It was first applied to inferring the parameters of an MLP in [SW89], although it has not been applied to bandit problems, as far as we know. In more detail, we define the latent variable to be the unknown parameters $\boldsymbol{\theta}_t$. The (non-stationary) observation model is given by $p_t(y_t|\boldsymbol{\theta}_t) = \mathcal{N}(y_t|f(\boldsymbol{s}_t, a_t; \boldsymbol{\theta}_t, \sigma^2)$, where $\boldsymbol{s}_t$ and a_t are inputs to the model, and the dynamics model for the parameters is given by $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \mathcal{N}(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, \tau^2\mathbf{I})$. We can set $\tau^2 = 0$ to encode the assumption that the parameters of the reward function are constant over time. However in practice we use a small non-zero value for τ, for numerical stability.

The belief state of an EKF has the form $\boldsymbol{b}_t = (\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$. This takes $O(D^2)$ space and $O(TD^3)$ time to compute. Modern neural networks often have millions of parameters, which makes direct application of the EKF intractable. We can reduce the memory from $O(D^2)$ to $O(D)$ and the time from $O(TD^3)$ to $O(TD^2)$ by using a diagonal approximation to $\boldsymbol{\Sigma}_t$. However, this ignores correlations between the parameters, which is important for good performance (as we show empirically in Section 4). We can improve the approximation by using a block structured approximation, with one block per layer of the MLP, but this still ignores correlations between layers.

In this paper, we explore a different approach to scaling the EKF to large neural networks. Our key insight is to exploit the fact that the DNN pararameters are not independent "degrees of freedom". Indeed, [Li+18] showed empirically that we can replace the original neural network weights $\boldsymbol{\theta} \in \mathbb{R}^D$ with a lower dimensional version, $\boldsymbol{z} \in \mathbb{R}^d$, by defining the affine mapping $\boldsymbol{\theta}(\boldsymbol{z}) = \mathbf{A}\boldsymbol{z} + \boldsymbol{\theta}_*$, and then optimizing the low-dimensional parameters $\boldsymbol{z}$. Here $\mathbf{A} \in \mathbb{R}^{D \times d}$ is a fixed but random Gaussian matrix with columns normalized to 1, and $\boldsymbol{\theta}_* \in \mathbb{R}^D$ is a random initial guess of the parameters (which we call an "offset"). In [Li+18], they show that optimizing in the $\boldsymbol{z}$ subspace gives good results on standard classification and RL benchmarks, even when $d \ll D$, provided that $d > d_{\text{min}}$, where d_{min} is a critical threshold. In [Lar+21], they provide a theoretical explanation for why such a threshold exists, based on geometric properties of the high dimensional loss landscape.

Instead of using a random offset $\boldsymbol{\theta}_*$, we can optimize it by performing SGD in the original $\boldsymbol{\theta}$ space during a warmup period. Similarly, instead of using a random basis matrix $\mathbf{A}$, we can optimize it by applying SVD to the iterates of SGD during the warmup period, as proposed in [Izm+19; Lar+21]. (If we wish, we can just keep a subset of the iterates, since consecutive samples are correlated.) These two changes reduce the dimensionality of the subspace d that we need to use in order to get good performance. (We can use cross-validation on the data from the warmup phase to find a good value for d.)

Once we have computed the subspace, we can perform Bayesian inference for the embedded parameters $\boldsymbol{z} \in \mathbb{R}^d$ instead of the original parameters $\boldsymbol{\theta} \in \mathbb{R}^D$. We do this by applying the EKF to the a state-space

model with a (non-stationary) observation model of the form $p_t(y_t|\boldsymbol{z}_t) = \mathcal{N}(y_t|f(\boldsymbol{s}_t, a_t; \mathbf{A}\boldsymbol{z}_t + \boldsymbol{\theta}_*), \sigma^2)$, and a deterministic transition model of the form $p(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}) = \mathcal{N}(\boldsymbol{z}_t|\boldsymbol{z}_{t-1}, \tau^2\mathbf{I})$.

The overall algorithm is summarized in Algorithm 2. (If we use a random subspace, we can skip the warmup phase, but results are worse, as we show in Section 4.) The algorithm takes $O(d^3)$ time per step. Empirically we find that we can reduce models with $D \sim 10^6$ down to $d \sim 10^2$ while getting the same (or sometimes better) performance, as we show in Section 4. We can further reduce the time to $O(d)$ by using a diagonal covariance matrix, with little change to the performance, as we shown in Section 4. The time cost of the warmup phase is dominated by SVD. If we have τ samples, the time complexity for exact SVD is $O(\min(\tau^2 D, D^2\tau))$. However, if we use randomized SVD [HMT11] this reduces the time to $O(\tau D \log d + (\tau + D)d^2)$.

The memory cost is $O(d^2 + Dd)$, since we need to store the belief state, $\boldsymbol{b}_t = (\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, as well as the offset $\boldsymbol{\theta}_*$ and the $D \times d$ basis matrix $\mathbf{A}$. We have succesfully scaled this to models with $\sim 1M$ parameters, but going beyond this may require the use of a sparse random orthogonal matrix to represent $\mathbf{A}$ [CRW17]. We leave this to future work.

Note that our method can be applied to any kind of DNN, not just MLPs. The low dimensional vector $\boldsymbol{z}$ depends on all of the parameters in the model. By contrast, the neural linear and Lim2 methods assume that the model has a linear final layer, and they only capture parameter uncertainty in this final layer. Thus these methods cannot be combined with the subspace trick.

Algorithm 2: Neural Subspace Bandits

$\mathcal{D}_\tau = \text{Environment.Warmup}(\tau)$;
$\boldsymbol{\theta}_{1:\tau} = \text{SGD}(\mathcal{D}_\tau)$;
$\boldsymbol{\theta}_* = \boldsymbol{\theta}_\tau$;
$\mathbf{A} = \text{SVD}(\boldsymbol{\theta}_{1:\tau})$;
$(\boldsymbol{\mu}_\tau, \boldsymbol{\Sigma}_\tau) = \text{EKF}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0, \mathcal{D}_{1:\tau})$;
for $t = (\tau + 1) : T$ **do**
  $\boldsymbol{s}_t = \text{Environment.GetState}(t)$;
  $\tilde{\boldsymbol{z}}_t \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$
  $a_t = \text{argmax}_a\, f(\boldsymbol{s}_t, a_t; \mathbf{A}\tilde{\boldsymbol{z}}_t + \boldsymbol{\theta}_*)$;
  $y_t = \text{Environment.GetReward}(\boldsymbol{s}_t, a_t)$;
  $\mathcal{D}_t = (\boldsymbol{s}_t, a_t, y_t)$;
  $(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) = \text{EKF}(\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \mathcal{D}_t)$;

4 Results

In this section, we present empirical results in which we evaluate the performance (reward) and speed (time) of our method compared to other methods on various bandit problems. We also study the effects of various hyper-parameters of our algorithm, such as how we choose the subspace.

4.1 Tabular datasets

To compare ourselves to prior works, we consider a subset of the datasets used in the "Deep Bayesian Bandits Showdown" [RTS18]. These are small tabular datasets, where the goal is to predict the class label given the features.[3] We turn this into a bandit problem by defining the actions to be the class labels, and the reward is 1 if the correct label is predicted, and is 0 otherwise. Thus the cumulative reward is the number of correct classifications, and the regret is the number of incorrect classifications.

Following prior work, we use the multi-headed MLP in Figure 1a, with one hidden layer with $N_h = 50$ units and ReLU activations. (The Neural-TS results are based on the multi-input model in Figure 1c.) We use $N_w = 20$ "pulls" per arm during the warmup phase and run for $T = 5000$ steps. We run 10 random trials and report the mean reward, together with the standard deviation.

We compare the following 11 methods: EKF in a learned (SVD) subspace (with full or diagonal covariance), EKF in a random subspace (with full or diagonal covariance), EKF in the original parameter space (with full or diagonal covariance), Linear, Neural-Linear (with unlimited or limited memory), LiM2, and Neural-TS. For the 6 EKF methods, we use our own code.[4] For LiM2 and Neural-TS, we use the original code from the authors.[5] For Linear and Neural-Linear methods, we reproduced the original code from the authors in our own codebase. All the hyperparameters are the same as in the original papers/code (namely [NZM21] for Linear, Neural-Linear and Lim2, and [Zha+21] for Neural-TS).

We show the average reward for each method on each dataset in Figure 2. (We use $d = 200$ for all experiments, which we found to work well.) On the Adult dataset, all methods have similar performance, showng that this is an easy problem. On the Covertype dataset, we find that the best method is EKF in a learned (SVD)

[3] The datasets are from the UCI ML repository https://archive.ics.uci.edu/ml/datasets. Statlog (shuttle) has 9 features, 7 classes. Coverype has 54 features, 7 classes. Adult has 89 features, 2 classes. We use $T = 5000$ samples for all datasets.

[4] Our code is available (in JAX) at https://github.com/probml/bandits.

[5] LiM2 is available (in TF1) at https://github.com/ofirnabati/Neural-Linear-Bandits-with-Likelihood-Matching. Neural-TS is available (in PyTorch) at https://github.com/ZeroWeight/NeuralTS.

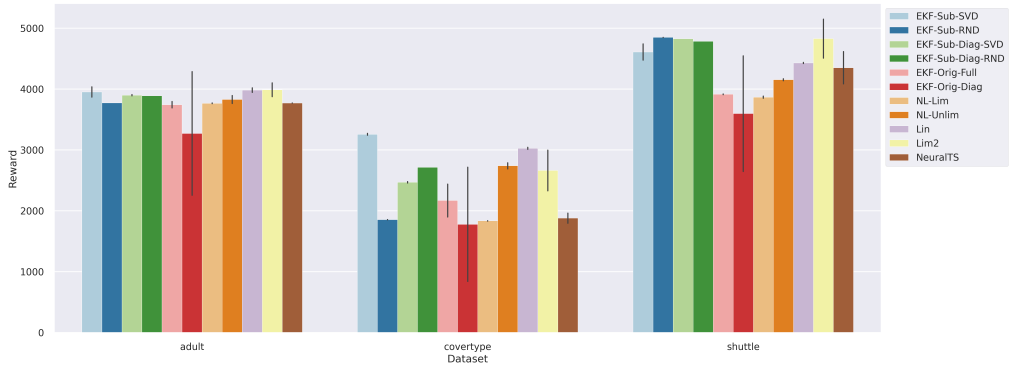


Figure 2: Reward for various methods on 3 tabular datasets. The maximum possible reward for each dataset is 5000.

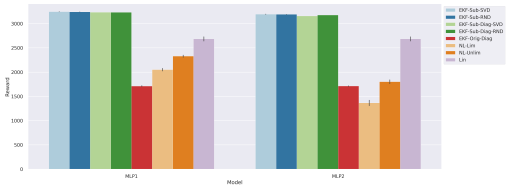


Figure 3: Reward for various methods on the Movielens dataset.

subspace with full covariance (light blue bar). This is the only method to beat the linear baseline (purple). On the Shuttle (Statlog) dataset, we see that all the EKF subspace variants work well, and match the accuracy of Lim2 while being much faster. (We discuss speed in Section 4.5.) We see that EKF in the original parameter space peforms worse, especially when we use a diagonal approximation (red). We also see that limited memory version of neural linear (light orange) is worse than unlimited memory (dark orange).

However, we also see that differences between most methods are often rather small, and are often within the error bars. We also noticed this with other examples from the Bandit Showdown benchmark (results not shown). We therefore believe this benchmark is too simple to be a reliable way of measuring performance differences of neural bandit algorithms (despite its popularity in the literature). In the sections below, we consider more challenging benchmarks, where the relative performance differences are clearer.

4.2 Recommender systems

One of the main applications of bandits is to recommender systems (see e.g., [Li+10; Guo+20]). Unfortunately, evaluating bandit policies in such systems requires running a live experiment, unless we have a simulator or we use off-policy evaluation methods such as those in [Li+11]. In this section, we build a simple simulator by applying SVD to the MovieLens-100k dataset, following the example in the TF-Agents library.[6]

In more detail, we start with the MovieLens-100k dataset, which has 100,000 ratings on a scale of 1–5 from 943 users on 1682 movies. This defines a sparse 943×1682 ratings matrix, where 0s correspond to missing entries. We extract a subset of this matrix corresponding to the first 20 movies to get a 943×20 matrix $\mathbf{X}$. We then compute the SVD of this matrix, $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\mathsf{T}$, and compute a dense low-rank approximation to it $\hat{\mathbf{X}} = \mathbf{U}_K\mathbf{S}_K\mathbf{V}_K^\mathsf{T}$. (This is a standard approach to matrix imputation, see e.g., [SJ03; BK07]). We treat each user i as a context, represented by $\boldsymbol{u}_i$, and treat each movie j as an action; the reward for taking action j in in context i is $X_{ij} \in \mathbb{R}$. We follow the TF-Agents example and use $K = 20$, so the context has 20 features, and there are also 20 actions (movies).

Having created this simulator, we can use it to evaluate various bandit algorithms. We use MLPs with 1 or 2 hidden layers, with 50 hidden units per layer. Since the Lim2 and NeuralTS code was not designed for this environment, we restrict ourselves to the 9 methods we have implemented ourselves. We show the results in Figure 3. On this dataset we see that the EKF subspace methods perform the best (by a large margin), followed by linear, and then neural-linear, and finally EKF in the original space (diagonal approximation). We also see that the deeper model (MLP2) performs worse than the shallower model (MLP1) when using the neural linear approximation; we attribute this to overfitting, due to not being Bayesian about the parameters of the feature extractor. By contrast, our fully Bayesian approach is robust to using overparameterized models, even in the small sample setting.

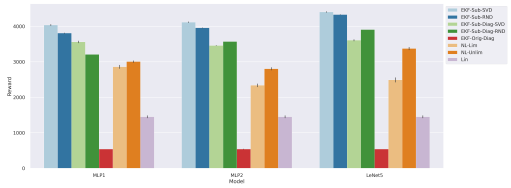


Figure 4: Reward for various methods on MNIST. The maximum possible reward is 5000.

4.3 MNIST

So far we have only considered low dimensional problems. To check the scalability of our method, we applied it to MNIST, which has 784 input features and 10 classes (actions). In addition to a baseline linear model, we consider three different kinds of deep neural network: an MLP with 50 hidden units and 10 linear outputs (MLP1, with $D = 39,760$ parameters), an MLP with two layers of 200 hidden units each and 10 linear outputs (MLP2 with $D = 48,420$ parameters), and a small convolutional neural network (CNN) known as LeNet5 [LeC+98] with $D = 61,706$ parameters.

Not surprisingly, we find that the CNN works better than MLP2, which works better than MLP1 (see Figure 4). Furthermore, for any given model, we see that our EKF-subspace method outperforms the widely used neural-linear method, even though the latter has unlimited memory (and therefore potentially takes $O(T^2)$ time).

For this experiment, we use a subspace dimensionality of $d = 470$ (chosen using a validation set). With this size of subspace, there is not a big difference between using an SVD subspace and a random subspace. However, using a full covariance in the subspace works better than a diagonal covariance (compare blue bars with the green bars). We see that all subspace methods work better than the neural linear baseline. In the original parameter space, a full covariance is intractable, and EKF with a diagonal approximation (red bar) works very poorly.

4.4 Varying the subspace

A critical component of our approach is how we estimating the parameter subspace matrix $\mathbf{A} \in \mathbb{R}^{D \times d}$. As we explained in Section 3.4, we have two different approaches for computing this: randomly or based on SVD applied to the parameter iterates computing by gradient descent during the warmup phase. We show the performance vs d for these two approaches in Figure 5 for a one-layer MLP with $D \sim 40k$ parameters on some tabular datasets. We see two main trends: SVD is usually much better than random, especially in low dimensions; and performance usually increases with d, and then either plateaus or even drops. The drop in performance with increasing dimensionality is odd, but is consistent with the results in [Lar+21], who noticed exactly the same effect. We leave investigating the causes of this to future work.

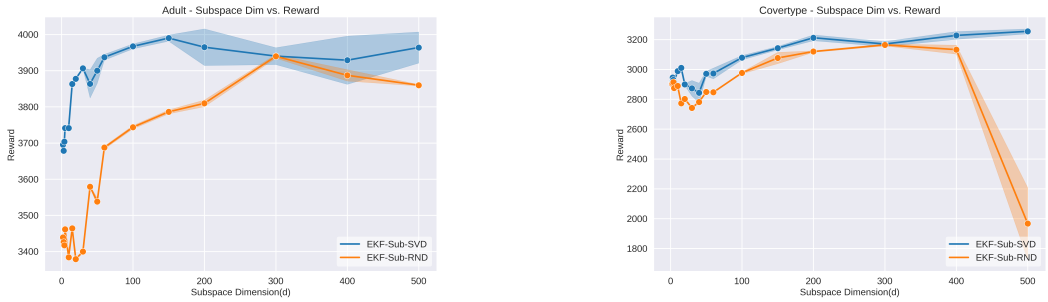
4.5 Time and space complexity

One aspect of bandit algorithms that has been overlooked in the literature is their time and space complexity, which is important in many practical applications, like recommender systems or robotic systems, that may run indefinitely (and hence need bounded memory) and need a fast response time. We give the asymptotic complexity of each method in Table 1. In Figure 6, we show the empirical wall clock time for each method when applied to the MovieLens dataset. We see the following trends: Neural-linear methods (orange) are the slowest, with the limited memory version usually being slightly faster than the unlimited memory version, as expected. The EKF subspace methods are the second slowest, with SVD slightly slower than RND, and full covariance (blue) slower than diagonal (green). Finally, the fastest method is diagonal EKF in the original parameter space; however, the performance (expected reward) of this method is poor. It is interesting to note that our subspace models are faster than the linear baseline; this is because we only have to invert a $d \times d$ matrix, instead of inverting N_a matrices, each of size $N_z \times N_z$.

In Figure 7, we show the empirical wall clock time for each method when applied to the MNIST dataset. The relative performance trends (when viewed on a log scale) are similar to the MovieLens case. However, the linear baseline is much slower than most other methods, since it works with the 784-dimensional input features, whereas the neural methods work with lower dimensional latent features. We also see that the neural linear method is quite slow, especially when applied to CNNs, and even more so in the unlimited memory setting. (We could not apply Lim2 to MNIST since the code is designed for the tabular datasets in the showdown benchmark.)

In addition to time constraints, memory is also a concern for long-running systems. Most online neural bandit methods store the entire past history of observations, to avoid catastrophic forgetting. If we limit SGD updates of the feature extractor to a window of the last $M = 100$ observations, performance drops (see e.g., Figure 2). The Lim2 method attempts to solve this, but is very slow, as we have seen. Our subspace EKF method is both fast and memory efficient.

[6] See https://bit.ly/3r9WkQd.



(a) *(b)*

Figure 5: Reward vs dimensionality of the subspace on (a) Adult, (b) Covertype. Blue estimates the subspace using SVD, orange uses a random subspace.

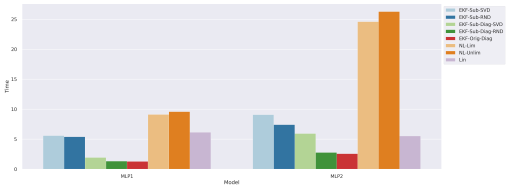


Figure 6: Running time (CPU seconds) for 5000 steps using various methods on MovieLens.

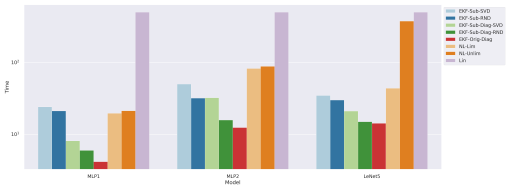


Figure 7: Running time (CPU seconds) for 5000 steps using various methods on MNIST. Note the vertical axis is logarithmic.

5 Discussion

We have shown that we can perform efficient online Bayesian inference for large neural networks by applying the extended Kalman filter to a low dimensional version of the parameter space. In future work, we would like to apply the method to other sequential decision problems, such as Bayesian optimization and active learning. We also intend to extend it to Bernoulli and other GLM bandits [Fil+10]. Fortunately, we can generalize the EKF (and hence our method) to work with the exponential family, as explained in [Oll18].

Finally, a note on societal impact. Our method makes online Bayesian inference for neural networks more tractable, which could increase their use. We view this as a positive thing, since Bayesian methods can express uncertainty, and may be less prone to making confident but wrong decisions [Bha+21]. However, we acknowledge that bandit algorithms are often used for recommender systems and online advertising, which can have some unintended harmful societal effects [MTF20].

Acknowledgements

We would like to thank Luca Rossini, Alex Shestopaloff and Efi Kokiopoulou for helpful comments on an earlier draft of the paper. This work was supported by an EPSRC studentship (for Gerardo) and by Google TPU Research Cloud (TRC).

References

[AG13] S. Agrawal and N. Goyal. "Thompson Sampling for Contextual Bandits with Linear Payoffs". In: *ICML*. Ed. by S. Dasgupta and D. McAllester. Vol. 28. Proceedings of Machine Learning Research. Atlanta, Georgia, USA: PMLR, 2013, pp. 127–135 (page 1).

[AKR21] M. Aziz, E. Kaufmann, and M.-K. Riviere. "On Multi-Armed Bandit Designs for Dose-Finding Clinical Trials". In: *JMLR* 22.14 (2021), pp. 1–38 (page 1).

[AZL19] Z. Allen-Zhu and Y. Li. "What Can ResNet Learn Efficiently, Going Beyond Kernels?" In: *NIPS*. 2019 (page 2).

[Bha+21] U. Bhatt et al. "Uncertainty as a Form of Transparency: Measuring, Communicating, and Using Uncertainty". In: *AAAI/ACM Conference on Artificial Intelligence, Ethics, and Society (AIES)*. 2021 (page 9).

[BK07] R. M. Bell and Y. Koren. "Lessons from the Netflix Prize Challenge". In: *SIGKDD Explor. Newsl.* 9.2 (Dec. 2007), pp. 75–79 (page 7).

[Blu+15] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. "Weight Uncertainty in Neural Networks". In: *ICML*. May 2015 (page 2).

[Bor16] S. M. Borodachev. "Recursive least squares method of regression coefficients estimation as a special case of Kalman filter". In: *Intl. Conf. of numerical analysis and applied mathematics*. Vol. 1738. American Institute of Physics, 2016, p. 110013 (page 13).

[CRW17] K. Choromanski, M. Rowland, and A. Weller. "The Unreasonable Effectiveness of Structured Random Orthogonal Embeddings". In: *NIPS*. Mar. 2017 (page 6).

[Dax+21a] E. Daxberger, A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig. "Laplace Redux–Effortless Bayesian Deep Learning". In: *arXiv preprint arXiv:2106.14806* (2021) (page 2).

[Dax+21b] E. Daxberger, E. Nalisnick, J. U. Allingham, J. Antorán, and J. M. Hernández-Lobato. "Bayesian Deep Learning via Subnetwork Inference". In: *ICML*. 2021 (page 2).

[Du+21] C. Du et al. "Exploration in Online Advertising Systems with Deep Uncertainty-Aware Learning". In: *KDD*. 2021 (page 1).

[Fil+10] S. Filippi, O. C. Ltci, L. T. P. T. Cnrs, and T. P. T. Cnrs. "Parametric bandits: The generalized linear case". In: *NIPS*. 2010 (page 9).

[FNG00] N. de Freitas, M. Niranjan, and A. Gee. "Hierarchical Bayesian models for regularisation in sequential learning". In: *Neural Computation* 12.4 (2000), pp. 955–993 (page 2).

[Fre99] R. M. French. "Catastrophic forgetting in connectionist networks". In: *Trends in Cognitive Science* (1999) (page 2).

[GDFY16] S. Ghosh, F. M. Delle Fave, and J. Yedidia. "Assumed Density Filtering Methods for Learning Bayesian Neural Networks". In: *AAAI*. 2016 (page 2).

[Gho+20] B. Ghorbani, S. Mei, T. Misiakiewicz, and A. Montanari. "When Do Neural Networks Outperform Kernel Methods?" In: (June 2020). arXiv: 2006.13409 [stat.ML] (page 2).

[Gre+17] K. Greenewald, A. Tewari, P. Klasnja, and S. Murphy. "Action Centered Contextual Bandits". In: *NIPS*. Nov. 2017 (page 1).

[Guo+20] D. Guo, S. I. Ktena, F. Huszar, P. K. Myana, W. Shi, and A. Tejani. "Deep Bayesian Bandits: Exploring in Online Personalized Recommendations". In: *RecSys*. Aug. 2020 (pages 1, 7).

[HLA15] J. M. Hernández-Lobato and R. P. Adams. "Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks". In: *ICML*. Feb. 2015 (page 2).

[HMT11] N. Halko, P.-G. Martinsson, and J. A. Tropp. "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". In: *SIAM Rev., Survey and Review section* 53.2 (2011), pp. 217–288 (page 6).

[Izm+18] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson. "Averaging Weights Leads to Wider Optima and Better Generalization". In: *UAI*. 2018 (page 2).

[Izm+19] P. Izmailov, W. J. Maddox, P. Kirichenko, T. Garipov, D. Vetrov, and A. G. Wilson. "Subspace Inference for Bayesian Deep Learning". In: *UAI*. 2019 (pages 1, 2, 5).

[Izm+21] P. Izmailov, S. Vikram, M. D. Hoffman, and A. G. Wilson. "What Are Bayesian Neural Network Posteriors Really Like?" In: *ICML*. Apr. 2021 (page 2).

[JGH18] A. Jacot, F. Gabriel, and C. Hongler. "Neural Tangent Kernel: Convergence and Generalization in Neural Networks". In: *NIPS*. 2018 (pages 2, 5).

[KCG12] E. Kaufmann, O. Cappe, and A. Garivier. "On Bayesian Upper Confidence Bounds for Bandit Problems". In: *AISTATS*. Ed. by N. D. Lawrence and M. Girolami. Vol. 22. Proceedings of Machine Learning Research. La Palma, Canary Islands: PMLR, 2012, pp. 592–600 (page 1).

[Kha20] M. E. Khan. *Deep learning with Bayesian principles*. NeurIPS tutorial. 2020 (page 2).

[Kir+17] J. Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". en. In: *PNAS* 114.13 (2017), pp. 3521–3526 (page 2).

[LA21] E. Levecque and R. Abecidan. *Study of the Neural Thompson Sampling algorithm*. Tech. rep. U. Lille, 2021 (page 15).

[Lar+21] B. W. Larsen, S. Fort, N. Becker, and S. Ganguli. "How many degrees of freedom do we need to train deep networks: a loss landscape perspective". In: (July 2021). arXiv: 2107.05802 [cs.LG] (pages 1, 5, 8).

[LeC+98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (page 8).

[Li+10] L. Li, W. Chu, J. Langford, and R. E. Schapire. "A contextual-bandit approach to personalized news article recommendation". In: *WWW*. 2010 (pages 1, 3, 7).

[Li+11] L. Li, W. Chu, J. Langford, and X. Wang. "Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms". In: *WSDM*. 2011 (page 7).

[Li+18] C. Li, H. Farkhoor, R. Liu, and J. Yosinski. "Measuring the Intrinsic Dimension of Objective Landscapes". In: *ICLR*. 2018 (pages 1, 5).

[LS19] T. Lattimore and C. Szepesvari. *Bandit Algorithms*. Cambridge, 2019 (pages 1, 2).

[Mac92] D. J. C. MacKay. "A Practical Bayesian Framework for Backpropagation Networks". In: *Neural Comput.* 4.3 (May 1992), pp. 448–472 (page 2).

[Mac95] D. MacKay. "Probable networks and plausible predictions — a review of practical Bayesian methods for supervised neural networks". In: *Network: Computation in Neural Systems* 6.3 (1995), pp. 469–505 (page 2).

[McM+13] H. B. McMahan et al. "Ad click prediction: a view from the trenches". In: *KDD*. 2013, pp. 1222–1230 (page 1).

[MTF20] S. Milano, M. Taddeo, and L. Floridi. "Recommender systems and their ethical challenges". In: *AI Soc.* 35.4 (Dec. 2020), pp. 957–967 (page 9).

[Nea95] R. M. Neal. "Bayesian Learning for Neural Networks". PhD thesis. University of Toronto, 1995 (page 2).

[Ngu+18] C. V. Nguyen, Y. Li, T. D. Bui, and R. E. Turner. "Variational Continual Learning". In: *ICLR*. 2018 (page 2).

[NZM21] O. Nabati, T. Zahavy, and S. Mannor. "Online Limited Memory Neural-Linear Bandits with Likelihood Matching". In: *ICML*. Vol. 139. Proceedings of Machine Learning Research. 2021, pp. 7905–7915 (pages 2, 4, 6, 14, 15).

[Oll18] Y. Ollivier. "Online natural gradient as a Kalman filter". en. In: *Electron. J. Stat.* 12.2 (2018), pp. 2930–2961 (pages 9, 19).

[Osa+19] K. Osawa et al. "Practical Deep Learning with Bayesian Principles". In: *NIPS*. 2019 (page 2).

[PAYD19] M. Phan, Y. Abbasi-Yadkori, and J. Domke. "Thompson sampling with approximate inference". In: *NIPS*. Aug. 2019 (page 1).

[PF03] G. V. Puskorius and L. A. Feldkamp. "Parameter-based Kalman filter training: Theory and implementation". In: *Kalman Filtering and Neural Networks*. New York, USA: John Wiley & Sons, Inc., 2003, pp. 23–67 (page 20).

[PF91] G. V. Puskorius and L. A. Feldkamp. "Decoupled extended Kalman filter training of feedforward layered networks". In: *International Joint Conference on Neural Networks*. Vol. i. July 1991, 771–777 vol.1 (page 20).

[PS17] N. G. Polson and V. Sokolov. "Deep Learning: A Bayesian Perspective". en. In: *Bayesian Anal.* 12.4 (Dec. 2017), pp. 1275–1304 (page 2).

[RBB18] H. Ritter, A. Botev, and D. Barber. "Online Structured Laplace Approximations for Overcoming Catastrophic Forgetting". In: *NIPS*. Curran Associates, Inc., 2018, pp. 3738–3748 (page 2).

[Rob95] A. Robins. "Catastrophic Forgetting, Rehearsal and Pseudorehearsal". In: *Conn. Sci.* 7.2 (June 1995), pp. 123–146 (page 2).

[RTS18] C. Riquelme, G. Tucker, and J. Snoek. "Deep Bayesian Bandits Showdown: An Empirical Comparison of Bayesian Deep Networks for Thompson Sampling". In: *ICLR*. 2018 (pages 1, 2, 4, 6, 14).

[Rus+18] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband, and Z. Wen. "A Tutorial on Thompson Sampling". In: *Foundations and Trends in Machine Learning* 11.1 (2018), pp. 1–96 (page 1).

[SJ03] N. Srebro and T. Jaakkola. "Weighted low-rank approximations". In: *ICML*. 2003 (page 7).

[Sli19] A. Slivkins. "Introduction to Multi-Armed Bandits". In: *Foundations and Trends in Machine Learning* (2019) (pages 1, 2).

[SW89] S. Singhal and L. Wu. "Training Multilayer Perceptrons with the Extended Kalman Algorithm". In: *NIPS*. Vol. 1. 1989 (pages 2, 5).

[WH97] M. West and J. Harrison. *Bayesian forecasting and dynamic models*. Springer, 1997 (page 14).

[WI20] A. G. Wilson and P. Izmailov. "Bayesian Deep Learning and a Probabilistic Perspective of Generalization". In: *NIPS*. Feb. 2020 (page 2).

[Wil20] A. G. Wilson. "The Case for Bayesian Deep Learning". In: (Jan. 2020). arXiv: 2001.10995 [cs.LG] (page 2).

[Zha+21] W. Zhang, D. Zhou, L. Li, and Q. Gu. "Neural Thompson Sampling". In: *ICLR*. 2021 (pages 2, 4, 6, 15).

[ZLG20] D. Zhou, L. Li, and Q. Gu. "Neural Contextual Bandits with UCB-based Exploration". In: *ICML*. Vol. 119. Proceedings of Machine Learning Research. 2020, pp. 11492–11502 (pages 2, 4).

Supplementary Material: Efficient Online Bayesian Inference for Neural Bandits

A More details on the methods

A.1 Linear bandits

In this section, we discuss how to do belief updating for a linear bandit, where the reward model has the form $f(\boldsymbol{s}, a; \boldsymbol{\theta}) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{s}$, where $\boldsymbol{\theta} = \mathbf{W}$ are the parameters. (We ignore the bias term, which can be accomodated by augmenting the input features $\boldsymbol{s}$ with a constant 1.) To simplify the notation, we give the derivation for a single arm. In practice, this procedure is repeated separately for each arm, using the contexts and rewards for the time periods where that arm was used.

A.1.1 Known variance σ^2

For now, we assume the observation noise σ^2 is known. We start with the uniformative prior $\boldsymbol{b}_0 = \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$, where $\boldsymbol{\mu}_0 = \mathbf{0}$ is the prior mean and $\boldsymbol{\Sigma}_0 = (1/\epsilon)\mathbf{I}$ is the prior covariance for some small $\epsilon > 0$. Let $\mathbf{X}$ be the $N \times N_s$ matrix of contexts for this arm during the warmup period (so $N = N_w$ if we pull each arm N_w times), and let $\boldsymbol{y}$ be the corresponding $N \times 1$ vector of rewards. We can compute the initial belief state based on the warmup data by applying Bayes rule to the uninformative prior to get

$$p(\boldsymbol{w}|\mathbf{X}_\tau, \boldsymbol{y}_\tau) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}_\tau, \boldsymbol{\Sigma}_\tau) \tag{5}$$

$$\boldsymbol{\Sigma}_\tau = (\boldsymbol{\Sigma}_0^{-1} + \frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\mathbf{X})^{-1} \tag{6}$$

$$\boldsymbol{\mu}_\tau = \boldsymbol{\Sigma}_\tau(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 + \frac{1}{\sigma^2}\mathbf{X}^\mathsf{T}\boldsymbol{y}) \tag{7}$$

After this initial batch update, we can perform incremental updates. We can use the Sherman-Morrison formula for rank one updating to efficiently compute the new covariance, without any matrix inversions:

$$\boldsymbol{\Sigma}_t = (\boldsymbol{\Sigma}_{t-1}^{-1} + \frac{1}{\sigma^2}\boldsymbol{x}_t\boldsymbol{x}_t^\mathsf{T})^{-1} = \boldsymbol{\Sigma}_{t-1} - \frac{\boldsymbol{\Sigma}_{t-1}\boldsymbol{x}_t\boldsymbol{x}_t^\mathsf{T}\boldsymbol{\Sigma}_{t-1}}{\sigma^2 + \boldsymbol{x}_t^\mathsf{T}\boldsymbol{\Sigma}_{t-1}\boldsymbol{x}_t} \tag{8}$$

To compute the mean, we will assume $\boldsymbol{\mu}_0 = \mathbf{0}$ and $\boldsymbol{\Sigma}_0 = \kappa^2\mathbf{I}$. Then we have

$$\boldsymbol{\mu}_t = \frac{1}{\sigma^2}\boldsymbol{\Sigma}_t\mathbf{X}^\mathsf{T}\boldsymbol{y} = \frac{1}{\sigma^2}\boldsymbol{\Sigma}_t\boldsymbol{\psi}_t \tag{9}$$

$$\boldsymbol{\psi}_t = \boldsymbol{\psi}_{t-1} + \boldsymbol{x}_t y_t \tag{10}$$

An alternative (but equivalent) approach is to use the recursive least squares (RLS) algorithm, which is a special case of the Kalman filter (see e.g., [Bor16] for the derivation). The updates are as follows:

$$e_t = y_t - \boldsymbol{x}_t^\mathsf{T}\boldsymbol{\mu}_{t-1} \tag{11}$$

$$s_t = \boldsymbol{x}_t^\mathsf{T}\boldsymbol{\Sigma}_{t-1}\boldsymbol{x}_t + \sigma^2 \tag{12}$$

$$\boldsymbol{k}_t = \frac{1}{s_t}\boldsymbol{\Sigma}_{t-1}\boldsymbol{x}_t \tag{13}$$

$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t-1} + \boldsymbol{k}_t e_t \tag{14}$$

$$\boldsymbol{\Sigma}_t = \boldsymbol{\Sigma}_{t-1} - \boldsymbol{k}_t\boldsymbol{k}_t^\mathsf{T} s_t \tag{15}$$

(Of course, we only update the belief state for the arm that was actually pulled at time t.)

A.1.2 Unknown variance σ^2

Now we consider the case where σ^2 is also unknown, as in [RTS18; NZM21]. This lets the algorithm explictly represent uncertainty in the reward for each action, which will increase the dynamic range of the sampled parameters, leading to more aggressive exploration. We have noticed this gives improved results over fixing σ.

We will use a conjugate normal inverse Gamma prior $\text{NIG}(\boldsymbol{w}, \sigma^2 | \boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0, a_0, b_0)$. The batch update is as follows, where $\mathbf{X}$ is all the contexts for this arm up to t, and $\boldsymbol{y}$ is all the rewards for this arm up to t:

$$p(\boldsymbol{w}, \sigma^2 | \mathbf{X}, \boldsymbol{y}) = \text{NIG}(\boldsymbol{w}, \sigma^2 | \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t, a_t, b_t) \tag{16}$$

$$\boldsymbol{\Sigma}_t = (\boldsymbol{\Sigma}_0^{-1} + \mathbf{X}^\mathsf{T}\mathbf{X})^{-1} \tag{17}$$

$$\boldsymbol{\mu}_t = \boldsymbol{\Sigma}_t(\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_t + \mathbf{X}^\mathsf{T}\boldsymbol{y}) \tag{18}$$

$$a_t = a_0 + \frac{N_t}{2} \tag{19}$$

$$b_t = b_0 + \frac{1}{2}\left(\boldsymbol{y}^\mathsf{T}\boldsymbol{y} + \boldsymbol{\mu}_0^\mathsf{T}\boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0 - \boldsymbol{\mu}_t\boldsymbol{\Sigma}_t^{-1}\boldsymbol{\mu}_t\right) \tag{20}$$

This matches Equations 1–2 of [RTS18].[7] To sample from this posterior, we first sample $\tilde{\sigma}^2 \sim \text{IG}(a_t, b_t)$, and then sample $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu}_t, \tilde{\sigma}^2\boldsymbol{\Sigma}_t)$.

We can rewrite the above equations in incremental form as follows:

$$p(\boldsymbol{w}, \sigma^2 | \mathcal{D}_{0:t}) = \text{NIG}(\boldsymbol{w}, \sigma^2 | \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t, a_t, b_t) \tag{21}$$

$$\boldsymbol{\Sigma}_t = (\boldsymbol{\Sigma}_{t-1}^{-1} + \boldsymbol{x}_t\boldsymbol{x}_t^\mathsf{T})^{-1} \tag{22}$$

$$\boldsymbol{\mu}_t = \boldsymbol{\Sigma}_t(\boldsymbol{\Sigma}_{t-1}^{-1}\boldsymbol{\mu}_{t-1} + \boldsymbol{x}_t y_t) \tag{23}$$

$$a_t = a_{t-1} + \frac{1}{2} \tag{24}$$

$$b_t = b_{t-1} + \frac{1}{2}\left(y_t^2 + \boldsymbol{\mu}_{t-1}^\mathsf{T}\boldsymbol{\Sigma}_{t-1}^{-1}\boldsymbol{\mu}_{t-1} - \boldsymbol{\mu}_t\boldsymbol{\Sigma}_t^{-1}\boldsymbol{\mu}_t\right) \tag{25}$$

It is natural to want to derive a version of these equations which avoids the matrix inversion at each step. We can incrementally update $\boldsymbol{\Sigma}_t$ without inverting $\boldsymbol{\Sigma}_{t-1}$, using Sherman-Morrison, as in Appendix A.1.1. However, computing b_t needs access to $\boldsymbol{\Sigma}_t^{-1}$. Fortunately, we can generalize the Kalman filter to the case where $V = \sigma^2$ is unknown, as described in [WH97, Sec 4.6]; this avoids any matrix inversions.

To describe this algorithm, let the likelihood at time t be defined as follows:

$$p_t(y_t | \boldsymbol{w}_t, V) = \mathcal{N}(y_t | \boldsymbol{x}_t^\mathsf{T}\boldsymbol{w}_t, V) \tag{26}$$

Let $\lambda = 1/V$ be the observation precision. To start the algorithm, we use the following prior:

$$p_0(\lambda) = \text{Ga}(\frac{\nu_0}{2}, \frac{\nu_0\tau_0}{2}) \tag{27}$$

$$p_0(\boldsymbol{w} | \lambda) = \mathcal{N}(\boldsymbol{\mu}_0, V\boldsymbol{\Sigma}_0^*) \tag{28}$$

where τ_0 is the prior mean for σ^2, and $\nu_0 > 0$ is the strength of this prior. We now discuss the belief updating step. We assume that the prior belief state at time $t-1$ is

$$\mathcal{N}(\boldsymbol{w}, \lambda | \mathcal{D}_{1:t-1}) = \mathcal{N}(\boldsymbol{z}_w | \boldsymbol{\mu}_{t-1}, V\boldsymbol{\Sigma}_{t-1}^*)\text{Ga}(\lambda | \frac{\nu_{t-1}}{2}, \frac{\nu_{t-1}\tau_{t-1}}{2}) \tag{29}$$

The posterior is given by

$$\mathcal{N}(\boldsymbol{w}, \lambda | \mathcal{D}_{1:t}) = \mathcal{N}(\boldsymbol{w} | \boldsymbol{\mu}_t, V\boldsymbol{\Sigma}_t^*)\text{Ga}(\lambda | \frac{\nu_t}{2}, \frac{\nu_t\tau_t}{2}) \tag{30}$$

[7] There is a small typo in Equation 2 of [RTS18]: the $\boldsymbol{\Sigma}_0$ should be inverted.

where

$$e_t = y_t - \boldsymbol{x}_t^\mathsf{T} \boldsymbol{\mu}_{t-1} \tag{31}$$
$$s_t^* = \boldsymbol{x}_t^\mathsf{T} \boldsymbol{\Sigma}_{t-1}^* \boldsymbol{x}_t + 1 \tag{32}$$
$$\boldsymbol{k}_t = \frac{1}{s_t^*} \boldsymbol{\Sigma}_{t-1}^* \boldsymbol{x}_t \tag{33}$$
$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t-1} + \boldsymbol{k}_t e_t \tag{34}$$
$$\boldsymbol{\Sigma}_t^* = \boldsymbol{\Sigma}_{t|t-1}^* - \boldsymbol{k}_t \boldsymbol{k}_t^\mathsf{T} s_t^* \tag{35}$$
$$\nu_t = \nu_{t-1} + 1 \tag{36}$$
$$\nu_t \tau_t = \nu_{t-1} \tau_{t-1} + e_t^2 / s_t^* \tag{37}$$

If we marginalize out V, the marginal distribution for $\boldsymbol{z}_t$ is a Student distribution. However, for Thompson sampling, it is simpler to sample $\tilde{\lambda} \sim \text{Ga}(\frac{\nu_t}{2}, \frac{\nu_t \tau_t}{2})$, and then to sample $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu}_t, \tilde{\sigma}^2 \boldsymbol{\Sigma}_t^*)$, where $\tilde{\sigma}^2 = 1/\tilde{\lambda}$.

A.2 Neural linear bandits

The neural linear model assumes that $f(\boldsymbol{s}, a; \boldsymbol{\theta}) = \boldsymbol{w}_a^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{s}; \mathbf{V})$, where $\boldsymbol{\phi}(\boldsymbol{s}; \mathbf{V})$ is the feature extractor. It approximates the posterior over all the parameters by using a point estimate for $\mathbf{V}$, a Gaussian distribution for each $\boldsymbol{w}_i$ (conditional on σ_i^2), and an inverse Gamma distributon for each σ_i^2, i.e.,

$$p(\boldsymbol{\theta} | \mathcal{D}_{1:t}) = \delta(\mathbf{V} - \hat{\mathbf{V}}_t) \prod_{i=1}^{N_a} \mathcal{N}(\boldsymbol{w}_i | \boldsymbol{\mu}_{t,i}, \sigma_i^2 \boldsymbol{\Sigma}_{t,i}) \text{IG}(\sigma_i^2 | a_i, b_i) \tag{38}$$

where $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W}, \boldsymbol{a}, \boldsymbol{b})$ are all the parameters, and $\delta(\boldsymbol{u})$ is a delta function. Furthermore, to avoid catastrophic forgetting, we also need to store all of the previous observations, so the belief state has the form $\boldsymbol{b}_t = (\mathcal{D}_{1:t}, \hat{\mathbf{V}}_t, \boldsymbol{\mu}_{t,1:N_a}, \boldsymbol{\Sigma}_{t,1:N_a}, \boldsymbol{a}_{1:N_a}, \boldsymbol{b}_{1:N_a})$. The neural network parameters are computed using SGD. After updating $\hat{\mathbf{V}}_t$, we update the parameters of the Normal-Inverse-Gamma distribution for the final layer weights $\mathbf{W}$, using the following equations

$$\boldsymbol{\Sigma}_i = (\boldsymbol{\Sigma}_{0,i}^{-1} + \mathbf{X}_i^\mathsf{T} \mathbf{X}_i)^{-1} \tag{39}$$
$$\boldsymbol{\mu}_{t,i} = \boldsymbol{\Sigma}_i (\boldsymbol{\Sigma}_{0,i}^{-1} \boldsymbol{\mu}_{0,i} + \mathbf{X}_i^\mathsf{T} \boldsymbol{y}_i) \tag{40}$$
$$a_i = a_{0,i} + \frac{N_i}{2} \tag{41}$$
$$b_i = b_{0,i} + \frac{1}{2} (\boldsymbol{y}_i^\mathsf{T} \boldsymbol{y}_i + \boldsymbol{\mu}_{0,i}^\mathsf{T} \boldsymbol{\Sigma}_{0,i} \boldsymbol{\mu}_{0,i} - \boldsymbol{\mu}_i^\mathsf{T} \boldsymbol{\Sigma}_i \boldsymbol{\mu}_i) \tag{42}$$

where we define $\mathbf{X}_i = [\boldsymbol{\phi}_j : a_j = i]$ as the matrix whose rows are the features $\boldsymbol{\phi}_j$ from time steps where action i was taken, and $\boldsymbol{y}_i = [r_j : a_j = i]$ is the vector of rewards from time steps where action i was taken. See Algorithm 3 for the pseudocode.

A.3 LiM2

In this section, we describe the LiM2 method of [NZM21]. It is similar to the neural linear method except that the prior $(\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i})$ gets updated after each SGD step, so as to not forget old information. In addition, SGD is only applied to a rolling window of the last M most recent observations, so the memory cost is bounded. See Algorithm 4 for the pseudocode.

See Algorithm 5 for the pseudocode for the step that updates the DNN and the prior on the last layer, to avoid catastrophic forgetting.

See Algorithm 6 for the projected gradient descent (PGD) step, which solves a semi definite program to optimize the new covariance.

A.4 Neural Thompson

In this section, we discuss the "Neural Thompson Sampling" method of [Zha+21]. We follow the presentation of [LA21], that shows the connection with linear TS.

Algorithm 3: Neural Linear.

for $t = (\tau + 1) : T$ **do**
  $\boldsymbol{s}_t = \text{Environment.GetState}(t)$;
  $\tilde{\sigma}_i \sim \text{InverseGamma}(a_i, b_i)$ for all i ;
  $\tilde{\boldsymbol{w}}_i \sim \mathcal{N}(\boldsymbol{\mu}_i, \tilde{\sigma}_i \boldsymbol{\Sigma}_i)$ for all i;
  $a_t = \text{argmax}_i \, \tilde{\boldsymbol{w}}_i^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{s}_t; \mathbf{V}_t)$;
  $y_t = \text{Environment.GetReward}(\boldsymbol{s}_t, a_t)$;
  $\mathcal{D}_t = (\boldsymbol{s}_t, a_t, y_t)$;
  if t is an SGD update step **then**
    $\boldsymbol{\theta} = \text{SGD}(\boldsymbol{\theta}, \mathcal{D}_{1:t})$;
    $\mathbf{V} = \text{parameters-for-body}(\boldsymbol{\theta})$;
    Compute new features: $\boldsymbol{\phi}_j = \boldsymbol{\phi}(\boldsymbol{s}_j; \mathbf{V})$ for all $j \in \mathcal{D}_{1:t}$;
    for $i = 1 : N_a$ **do**
      // Update sufficient statistics ;
      $\boldsymbol{\psi}_i = \sum_{j \le t: a_t = i} \boldsymbol{\phi}_j y_j$;
      $\boldsymbol{\Phi}_i = \sum_{j \le t: a_t = i} \boldsymbol{\phi}_j \boldsymbol{\phi}_j^\mathsf{T}$;
      $R_i^2 = \sum_{j \le t: a_t = i} y_t^2$;
      $N_i = \sum_{j \le t: a_t = i} 1$;
      // Update belief state ;
      $(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i, a_i, b_i) = \text{update-bel}(\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}, a_{0,i}, b_{0,i}, \boldsymbol{\psi}_i, \boldsymbol{\Phi}_i, R_i^2, N_i)$
  else
    $i = a_t$;
    $\boldsymbol{\psi}_i = \boldsymbol{\psi}_i + \boldsymbol{\phi}_t y_t$;
    $\boldsymbol{\Phi}_i = \boldsymbol{\Phi}_i + \boldsymbol{\phi}_t \boldsymbol{\phi}_t^\mathsf{T}$;
    $R_i^2 = R_i^2 + y_t^2$;
    $N_i = N_i + 1$;
    $(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i, a_i, b_i) = \text{update-bel}(\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}, a_{0,i}, b_{0,i}, \boldsymbol{\psi}_i, \boldsymbol{\Phi}_i, R_i^2, N_i)$
;
function update-bel$(\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}, a_{0,i}, b_{0,i}, \boldsymbol{\psi}_i, \boldsymbol{\Phi}_i, R_i^2, N_i))$;
$\boldsymbol{\Sigma}_i = (\boldsymbol{\Sigma}_{0,i}^{-1} + \boldsymbol{\Phi}_i)^{-1}$;
$\boldsymbol{\mu}_i = \boldsymbol{\Sigma}_i (\boldsymbol{\Sigma}_{0,i}^{-1} \boldsymbol{\mu}_{0,i} + \boldsymbol{\psi}_i)$;
$a_i = a_{0,i} + \frac{N_i}{2}$;
$b_i = b_{0,i} + \frac{1}{2}(R_i^2 + \boldsymbol{\mu}_{0,i}^\mathsf{T} \boldsymbol{\Sigma}_{0,i} \boldsymbol{\mu}_{0,i} - \boldsymbol{\mu}_i^\mathsf{T} \boldsymbol{\Sigma}_i \boldsymbol{\mu}_i)$;
return $(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i, a_i, b_i)$

Algorithm 4: LiM2

for $t = (\tau + 1) : T$ **do**
- $\boldsymbol{s}_t = \text{Environment.GetState}(t)$;
- $\tilde{\sigma}_i \sim \text{IG}(a_i, b_i)$ for all i ;
- $\tilde{\boldsymbol{w}}_i \sim \mathcal{N}(\boldsymbol{\mu}_i, \tilde{\sigma}_i \boldsymbol{\Sigma}_i)$ for all i;
- $a_t = \text{argmax}_i \, \tilde{\boldsymbol{w}}_i^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{s}_t; \mathbf{V}_t)$;
- $y_t = \text{Environment.GetReward}(\boldsymbol{s}_t, a_t)$;
- $\mathcal{D}_t = (\boldsymbol{s}_t, a_t, y_t)$;
- $\mathcal{M}_t = \text{push}(\mathcal{D}_t)$;
- **if** $|\mathcal{M}_t| > M$ **then**
  - $\mathcal{M}_t = \text{pop}(\mathcal{M}_t)$
- $(\boldsymbol{\theta}, \{\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}\}) = \text{update-DNN-and-prior } (\boldsymbol{\theta}, \{\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}\}, \mathcal{M}_t)$;
- $\mathbf{V} = \text{body}(\boldsymbol{\theta})$;
- Compute new features: $\boldsymbol{\phi}_j = \boldsymbol{\phi}(\boldsymbol{s}_j; \mathbf{V})$ for all $j \in \mathcal{M}_t$;
- **for** $i = 1 : N_a$ **do**
  - // Update sufficient statistics ;
  - $\boldsymbol{\psi}_i = \sum_{j \in \mathcal{M}_t : a_t = i} \boldsymbol{\phi}_j y_j$;
  - $\boldsymbol{\Phi}_i = \sum_{j \in \mathcal{M}_t : a_t = i} \boldsymbol{\phi}_j \boldsymbol{\phi}_j^\mathsf{T}$;
  - $R_i^2 = \sum_{j \in \mathcal{M}_t : a_t = i} y_t^2$;
  - $N_i = \sum_{j \in \mathcal{M}_t : a_t = i} 1$;
  - // Update belief state ;
  - $(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i, a_i, b_i) = \text{update-bel}(\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}, a_{0,i}, b_{0,i}, \boldsymbol{\psi}_i, \boldsymbol{\Phi}_i, R_i^2, N_i)$

Algorithm 5: LiM2 update step

Input: $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W})$, $\{\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}\}$, $\mathcal{D}$;
for P_1 steps **do**
- Sample mini batch $\mathcal{D}' = \{\boldsymbol{s}_j, a_j, y_j) : j = 1 : N_b\}$ from $\mathcal{D}$;
- Compute old features: $\boldsymbol{\phi}_{j,\text{old}} = \boldsymbol{\phi}(\boldsymbol{s}_j; \mathbf{V})$ for all $j \in \mathcal{D}'$;
- $\boldsymbol{\theta} = \text{SGD}(\boldsymbol{\theta}, \mathcal{D}')$;
- $\mathbf{V} = \text{params-for-body}(\boldsymbol{\theta})$, $\mathbf{W} = \text{params-for-head}(\boldsymbol{\theta})$;
- Compute new features: $\boldsymbol{\phi}_j = \boldsymbol{\phi}(\boldsymbol{s}_j; \mathbf{V})$ for all $j \in \mathcal{D}'$;
- **for** $i = 1 : N_a$ **do**
  - $\boldsymbol{\Sigma}_{0,i} = \text{PGD}(\boldsymbol{\Sigma}_{0,i}, \{\phi_{j,\text{old}} : a_j = i\}, \{\phi_j : a_j = i\})$;

$\boldsymbol{\mu}_{0,i} = \boldsymbol{w}_i$ for each i ;
Return $\boldsymbol{\theta}$, $\{\boldsymbol{\mu}_{0,i}, \boldsymbol{\Sigma}_{0,i}\}$;

Algorithm 6: Projected Gradient Descent

Input: $\mathbf{A}$, $\{\boldsymbol{\phi}_{j,old}\}$, $\{\boldsymbol{\phi}_j\}$
$s_j^2 = \boldsymbol{\phi}_{j,\text{old}}^\mathsf{T} \mathbf{A} \boldsymbol{\phi}_{j,\text{old}}$ for all j ;
$\boldsymbol{\Phi}_j = \boldsymbol{\phi}_j \boldsymbol{\phi}_j^\mathsf{T}$ for all j ;
for P_2 steps **do**
- $\boldsymbol{g} = 2 \sum_j (\text{tr}(\mathbf{A} \boldsymbol{\Phi}_j) - s_j^2) \boldsymbol{\Phi}_j$
- $\mathbf{A} = \mathbf{A} - \eta \boldsymbol{g}$
- $(\boldsymbol{\Lambda}, \mathbf{V}) = \text{eig}(\mathbf{A})$
- $\mathcal{N} = \{k : \lambda_k < 0\}$
- $\boldsymbol{\Lambda}[k, k] = 0$ for all $k \in \mathcal{N}$
- $\mathbf{V}[:, k] = 0$ for all $k \in \mathcal{N}$
- $\mathbf{A} = \mathbf{V} \boldsymbol{\Lambda} \mathbf{V}^\mathsf{T}$

Return $\mathbf{A}$;

First consider the linear model $r_{t,a} = \boldsymbol{x}_{t,a}^\mathsf{T}\boldsymbol{w}$. We assume σ^2 is fixed, $\boldsymbol{\Sigma}_0 = \kappa^2\mathbf{I}$, $\boldsymbol{\mu}_0 = \mathbf{0}$, and $\lambda = \frac{\sigma^2}{\kappa^2}$. Recall from Appendix A.1.1 that the posterior over the parameters is given by

$$\boldsymbol{\Sigma}_t = \left[\frac{1}{\sigma^2}(\sigma^2\boldsymbol{\Sigma}_0^{-1} + \sum_{j=1}^{t}\boldsymbol{x}_j\boldsymbol{x}_j^\mathsf{T}\right]^{-1} = \sigma^2\left[\underbrace{\lambda\mathbf{I} + \sum_{j=1}^{t}\boldsymbol{x}_j\boldsymbol{x}_j^\mathsf{T}}_{\mathbf{B}_t}\right]^{-1} \tag{43}$$

$$\boldsymbol{\mu}_t = \frac{1}{\sigma^2}\boldsymbol{\Sigma}_t\boldsymbol{\psi}_t = \mathbf{B}_t^{-1}\boldsymbol{\psi}_t = \mathbf{B}_t^{-1}\sum_{j=1}^{T}\boldsymbol{x}_j y_j \tag{44}$$

Thus the posterior over the parameters is given by

$$p(\boldsymbol{w}|\mathcal{D}_{1:t}) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}_t, \lambda\kappa^2\mathbf{B}_t^{-1}) \tag{45}$$

The induced posterior predictive distribution over the reward is given by

$$p(y|\boldsymbol{s}, a, \mathcal{D}_{1:t-1}) = \mathcal{N}(y|\mu_{t,a}, v_{t,a}) \tag{46}$$

$$\mu_{t,a} = \boldsymbol{x}_{t,a}^\mathsf{T}\mathbb{E}\left[\boldsymbol{w}\right] = \boldsymbol{x}_{t,a}^\mathsf{T}\boldsymbol{\mu}_{t-1} \tag{47}$$

$$v_{t,a} = \boldsymbol{x}_{t,a}^\mathsf{T}\mathbb{V}\left[\boldsymbol{w}\right]\boldsymbol{x}_{t,a} = \kappa^2\lambda\boldsymbol{x}_{t,a}^\mathsf{T}\mathbf{B}_{t-1}^{-1}\boldsymbol{x}_{t,a} \tag{48}$$

Now consider the NTK case. We replace $\boldsymbol{x}_{t,a}$ with

$$\boldsymbol{\phi}_{t,a} = \frac{1}{\sqrt{N_h}}\nabla_{\boldsymbol{\theta}} f(\boldsymbol{s}, a; \boldsymbol{\theta})|_{\boldsymbol{\theta}_{t-1}} \tag{49}$$

which is the gradient of the neural net (an MLP with N_h units per layer). If we set $\kappa^2 = 1/N_h$, then the posterior predictive distribution for the reward becomes

$$p(y|\boldsymbol{s}, a, \mathcal{D}_{1:t}) = \mathcal{N}(y|\mu_{t,a}, v_{t,a}) \tag{50}$$

$$\mu_{t,a} = f(\boldsymbol{s}_t, a; \boldsymbol{\theta}_{t-1}) \tag{51}$$

$$v_{t,a} = \lambda\boldsymbol{\phi}_{t,a}^\mathsf{T}\mathbf{B}_{t-1}^{-1}\boldsymbol{\phi}_{t,a} \tag{52}$$

where

$$\mathbf{B}_t = \mathbf{B}_{t-1} + \boldsymbol{\phi}(\boldsymbol{s}_t, a_t; \boldsymbol{\theta}_t)\boldsymbol{\phi}(\boldsymbol{s}_t, a_t; \boldsymbol{\theta}_t)^\mathsf{T} \tag{53}$$

and we initialize with $\mathbf{B}_0 = \lambda\mathbf{I}$. We sample the reward from this distribution for each action a, and then the greedy action is chosen.

A.5 EKF

In this section, we describe the extended Kalman filter (EKF) formulation in more detail. Consider the following nonlinear Gaussian state space model:

$$\boldsymbol{z}_t = \boldsymbol{f}_t(\boldsymbol{z}_{t-1}) + \mathcal{N}(\mathbf{0}, \mathbf{Q}_t) \tag{54}$$

$$\boldsymbol{y}_t = \boldsymbol{h}_t(\boldsymbol{z}_t) + \mathcal{N}(\mathbf{0}, \mathbf{R}_t) \tag{55}$$

where $\boldsymbol{z}_t \in \mathbb{R}^{N_z}$ is the hidden state, $\boldsymbol{y}_t \in \mathbb{R}^{N_y}$ is the observation, $\boldsymbol{f}_t : \mathbb{R}^{N_z} \to \mathbb{R}^{N_z}$ is the dynamics model, and $\boldsymbol{h}_t : \mathbb{R}^{N_z} \to \mathbb{R}^{N_y}$ is the observation model. The EKF linearizes the model at each step by computing the following Jacobian matrices:

$$\mathbf{F}_t = \frac{\partial\boldsymbol{f}_t(\boldsymbol{z})}{\partial\boldsymbol{z}}|_{\boldsymbol{\mu}_{t-1}} \tag{56}$$

$$\mathbf{H}_t = \frac{\partial\boldsymbol{h}_t(\boldsymbol{z})}{\partial\boldsymbol{z}}|_{\boldsymbol{\mu}_{t|t-1}} \tag{57}$$

(These terms are easy to compute using standard libraries such as JAX.) The updates then become

$$\boldsymbol{\mu}_{t|t-1} = \boldsymbol{f}(\boldsymbol{\mu}_{t-1}) \tag{58}$$
$$\boldsymbol{\Sigma}_{t|t-1} = \mathbf{F}_t \boldsymbol{\Sigma}_{t-1} \mathbf{F}_t + \mathbf{Q}_t \tag{59}$$
$$\boldsymbol{e}_t = \boldsymbol{y}_t - \boldsymbol{h}(\boldsymbol{\mu}_{t|t-1}) \tag{60}$$
$$\mathbf{S}_t = \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t^\mathsf{T} + \mathbf{R}_t \tag{61}$$
$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t \mathbf{S}_t^{-1} \tag{62}$$
$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t \boldsymbol{e}_t \tag{63}$$
$$\boldsymbol{\Sigma}_t = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{S}_t \mathbf{K}_t^\mathsf{T} \tag{64}$$

(In the case of Bernoulli bandits, we can use the exponential family formulation of the EKF discussed in [Oll18].)

The cost of the EKF is $O(N_y N_z^2)$, which can be prohibitive for large state spaces. In such cases, a natural approximation is to use a block diagonal approximation. Let us define the following Jacobian matrices for block i:

$$\mathbf{F}_t^i = \frac{\partial \boldsymbol{f}_t^i(\boldsymbol{z})}{\partial \boldsymbol{z}}|_{\boldsymbol{\mu}_{t-1}} \tag{65}$$
$$\mathbf{H}_t^i = \frac{\partial \boldsymbol{h}_t^i(\boldsymbol{z})}{\partial \boldsymbol{z}}|_{\boldsymbol{\mu}_{t|t-1}} \tag{66}$$

We then compute the following updates for each block:

$$\boldsymbol{\mu}_{t|t-1}^i = \boldsymbol{f}^i(\boldsymbol{\mu}_{t-1}) \tag{67}$$
$$\boldsymbol{\Sigma}_{t|t-1}^i = (\mathbf{F}_{t-1}^i)^\mathsf{T} \boldsymbol{\Sigma}_{t-1}^i \mathbf{F}_t^i + \mathbf{Q}_{t-1}^i \tag{68}$$
$$\mathbf{S}_t = \sum_i (\mathbf{H}_t^i)^\mathsf{T} \boldsymbol{\Sigma}_{t|t-1}^i \mathbf{H}_t^i + \mathbf{R}_t \tag{69}$$
$$\mathbf{K}_t^i = \boldsymbol{\Sigma}_{t|t-1}^i \mathbf{H}_t^i \mathbf{S}_t^{-1} \tag{70}$$
$$\boldsymbol{\mu}_t^i = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t^i \boldsymbol{e}_t \tag{71}$$
$$\boldsymbol{\Sigma}_t^i = \boldsymbol{\Sigma}_{t|t-1}^i - \mathbf{K}_t^i \mathbf{H}_t^i \boldsymbol{\Sigma}_{t|t-1}^i \tag{72}$$

Now we specialize the above equations to the setting of this paper, where the latent state is $\boldsymbol{z}_t = \boldsymbol{\theta}_t$, and the dynamics model f_t is the identify function. Thus the state space model becomes

$$p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \mathcal{N}(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}, \mathbf{Q}_t) \tag{73}$$
$$p(y_t|\boldsymbol{x}_t, \boldsymbol{\theta}_t) = \mathcal{N}(y_t|f(\boldsymbol{x}_t, \boldsymbol{\theta}_t), \mathbf{R}_t) \tag{74}$$

where $\boldsymbol{x}_t = (\boldsymbol{s}_t, a_t)$. We set $\mathbf{R}_t = \sigma^2$, and $\mathbf{Q}_t = \epsilon \mathbf{I}$, to allow for a small amount of parameter drift. The EKF updates become

$$\boldsymbol{\Sigma}_{t|t-1} = \boldsymbol{\Sigma}_{t-1} + \mathbf{Q}_t \tag{75}$$
$$\mathbf{S}_t = \mathbf{H}_t^\mathsf{T} \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t + \mathbf{R}_t \tag{76}$$
$$\mathbf{K}_t = \boldsymbol{\Sigma}_{t|t-1} \mathbf{H}_t \mathbf{S}_t^{-1} \tag{77}$$
$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t-1} + \mathbf{K}_t \boldsymbol{e}_t \tag{78}$$
$$\boldsymbol{\Sigma}_t = \boldsymbol{\Sigma}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \boldsymbol{\Sigma}_{t|t-1} \tag{79}$$

The block diagonal version becomes

$$\boldsymbol{\Sigma}_{t|t-1}^i = \boldsymbol{\Sigma}_{t-1}^i + \mathbf{Q}_{t-1}^i \tag{80}$$
$$\mathbf{S}_t = \sum_i (\mathbf{H}_t^i)^\mathsf{T} \boldsymbol{\Sigma}_{t|t-1}^i \mathbf{H}_t^i + \mathbf{R}_t \tag{81}$$
$$\mathbf{K}_t^i = \boldsymbol{\Sigma}_{t|t-1}^i \mathbf{H}_t^i \mathbf{S}_t^{-1} \tag{82}$$
$$\boldsymbol{\mu}_t^i = \boldsymbol{\mu}_{t-1}^i + \mathbf{K}_t^i \boldsymbol{e}_t \tag{83}$$
$$\boldsymbol{\Sigma}_t^i = \boldsymbol{\Sigma}_{t|t-1}^i - \mathbf{K}_t^i \mathbf{H}_t^i \boldsymbol{\Sigma}_{t|t-1}^i \tag{84}$$

This is called the "decoupled EKF" [PF91; PF03].

To match the notation in [PF03], let us define $\mathbf{P}_t = \mathbf{\Sigma}_{t|t-1}$, $\boldsymbol{w}_t = \boldsymbol{\mu}_{t|t-1}$, $\mathbf{A}_t = \mathbf{S}_t^{-1}$, $\hat{\mathbf{H}}_t^\intercal = \mathbf{H}_t$. (Note that $\mathbf{A}_t$ is a $N_o \times N_o$ matrix, so is a scalar if $y_t \in \mathbb{R}$.) Then we can rewrite the above as follows:

$$\mathbf{A}_t = \left(\mathbf{R}_t + \sum_i (\mathbf{H}_t^i)^\intercal \mathbf{P}_t^i \mathbf{H}_t^i \right)^{-1} \tag{85}$$

$$\mathbf{K}_t^i = \mathbf{P}_t^i \mathbf{H}_t^i \mathbf{A}_t^i \tag{86}$$

$$\boldsymbol{w}_{t+1}^i = \boldsymbol{w}_t^i + \mathbf{K}_t^i \boldsymbol{e}_t \tag{87}$$

$$\mathbf{P}_{t+1}^i = \mathbf{P}_t^i - \mathbf{K}_t^i (\hat{\mathbf{H}}_t^i)^\intercal \mathbf{P}_t^i + \mathbf{Q}_t^i \tag{88}$$