Functional Regularization for Reinforcement Learning via Learned Fourier Features

Alexander C. Li
Carnegie Mellon University
alexanderli@cmu.edu

Deepak Pathak
Carnegie Mellon University
dpathak@cs.cmu.edu

Abstract

We propose a simple architecture for deep reinforcement learning by embedding inputs into a learned Fourier basis and show that it improves the sample efficiency of both state-based and image-based RL. We perform infinite-width analysis of our architecture using the Neural Tangent Kernel and theoretically show that tuning the initial variance of the Fourier basis is equivalent to *functional* regularization of the learned deep network. That is, these learned Fourier features allow for adjusting the degree to which networks underfit or overfit different frequencies in the training data, and hence provide a controlled mechanism to improve the stability and performance of RL optimization. Empirically, this allows us to prioritize learning low-frequency functions and speed up learning by reducing networks' susceptibility to noise in the optimization process, such as during Bellman updates. Experiments on standard state-based and image-based RL benchmarks show clear benefits of our architecture over the baselines[1].

1 Introduction

Most popular deep reinforcement learning (RL) approaches estimate either a value or Q-value function under the agent's learned policy. These functions map points in the state or state-action space to expected returns, and provide crucial information that is used for improving the policy. However, optimizing these functions can be difficult, since there are no ground-truth labels to predict. Instead, they are trained through bootstrapping: the networks are updated towards target values calculated with the same networks being optimized. These updates introduce noise that accumulates over repeated iterations of bootstrapping, which can result in highly inaccurate value or Q-value estimates [44, 46]. As a result, these RL algorithms may suffer from lower asymptotic performance or sample efficiency.

Most prior work has focused on making the estimation of target values more accurate. Some examples include double Q-learning for unbiased target values [16, 47], or reducing the reliance on the bootstrapped Q-values by calculating multi-step returns with TD(λ) [41]. However, it is impossible to hope that the noise in the target values estimated via bootstrapping will go to zero, because we cannot estimate the true expectation over infinite rollouts in practical setups. Hence, we argue that it is equally important to also regularize the function (in this case, the deep Q-network) that is fitting these noisy target values.

Conventional regularization methods in supervised learning can be associated with drawbacks in RL. Stochastic methods like dropout [40] introduce more noise into the process, which is tolerable when ground-truth labels are present (in supervised learning) but counterproductive when bootstrapping. An alternative approach is early stopping [50], which hurts sample efficiency in reinforcement learning because a minimum number of gradient steps is required to propagate value information backwards to early states. Finally, penalty-based methods like L_1 [45] and L_2 regularization [17, 22] can help in

[1] Code available at https://github.com/alexlioralexli/learned-fourier-features

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

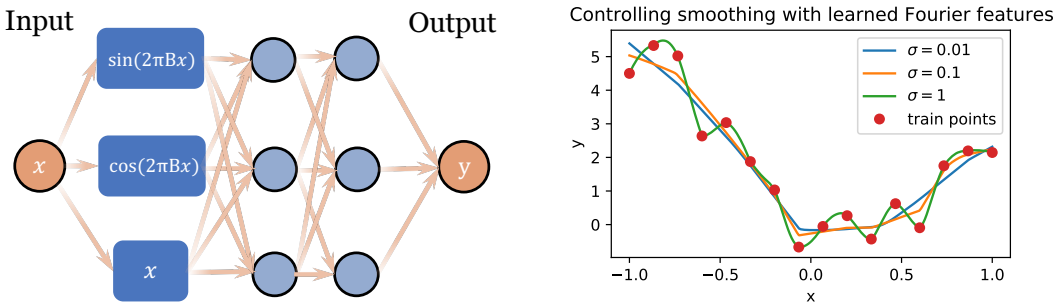


Figure 1: **Left:** the proposed learned Fourier feature (LFF) architecture. B is a matrix trained through backpropagation, and is used to create a learned set of Fourier features. The network then passes the Fourier features through alternating linear layers and ReLU nonlinearities, as is done in vanilla MLPs. **Right:** tuning the initialization variance of $B_{ij} \sim \mathcal{N}(0, \sigma^2)$ controls the rate at which target frequencies are learned. Higher σ fits higher frequencies faster, while lower σ smooths out noise. This architecture can be used as functional regularization for a Q-function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, value function $V : \mathcal{S} \to \mathbb{R}$, policy $\pi : \mathcal{S} \to \mathbb{R}^{\dim(\mathcal{A})}$, or model $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$.

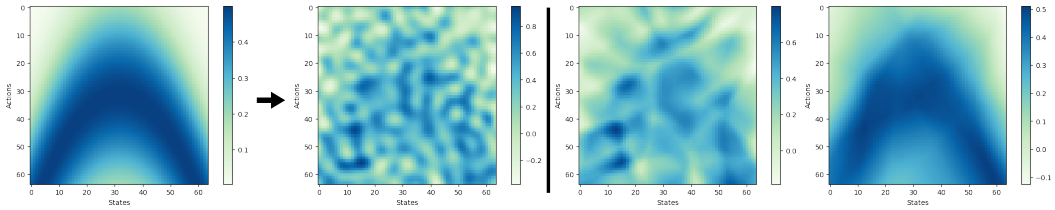
RL [26], but regularizing the network in weight space does not disentangle noise from reward signal and could make it difficult to learn the true Q-function. This leads us to ask: what is the right way to regularize the RL bootstrapping process?

We suggest that the impact of target value noise can be better reduced by *frequency-based functional regularization*: direct control over the frequencies that the network tends to learn first. If the target noise consists of higher frequencies than the true target values, discouraging high-frequency learning can help networks efficiently learn the underlying Q-function while fitting minimal amounts of noise. In this work, we propose an architecture that achieves this by encoding the inputs with learned Fourier features, which we abbreviate as LFF. In contrast to using fixed Fourier features [34, 42], we train the Fourier features, which helps them find an appropriate basis even in high dimensional domains. We analyze our architecture using the Neural Tangent Kernel [18] and theoretically show that tuning the initial variance of the Fourier basis controls the rate at which networks fit different frequencies in the training data. Thus, LFF's initial variance provides a controlled mechanism to improve the stability and performance of RL optimization (see Figure 1). Tuned to prioritize learning low frequencies, LFF filters out bootstrapping noise while learning the underlying Q-function.

We evaluate LFF, which only requires changing a few lines of code, on state-space and image-space DeepMind Control Suite environments [43]. We find that LFF produces moderate gains in sample efficiency on state-based RL and dramatic gains on image-based RL. In addition, we empirically demonstrate that LFF makes the value function bootstrapping stable even in absence of target networks, and confirm that most of LFF's benefit comes through regularizing the Q-network. Finally, we provide a thorough ablation of our architectural design choices.

2 Preliminaries

The reinforcement learning objective is to solve a Markov Decision Process (MDP), which is defined as a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. $\mathcal{S}$ and $\mathcal{A}$ denote the state and action spaces. $P(s'|s,a)$ is the transition function, $R(s,a)$ is the reward function, and $\gamma \in [0,1]$ is the discount factor. We aim to find an optimal policy $\pi^*(a|s)$ that maximizes the expected sum of discounted rewards. Q-learning finds the optimal policy by first learning a function $Q^*(s,a)$ such that $Q^*(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)}[R(s,a) + \gamma \max_{a'} Q^*(s',a')]$. The optimal policy is then $\pi^*(a|s) = \arg\max_a Q^*(s,a)$. To find Q^*, we repeatedly perform Bellman updates to Q, which uses the Q-network itself to bootstrap target values on observed transitions (s,a,r,s') [6]. The most basic way to estimate the target values is $target = r + \gamma \max_{a'} Q(s',a')$, but a popular line of work in RL aims to find more accurate target value estimation methods [12, 16, 32, 41, 47]. Once we have these target value estimates, we update our Q-network's parameters θ via gradient descent on temporal difference error: $\Delta\theta = -\eta \nabla_\theta \left(Q_\theta(s,a) - target\right)^2$. Our focus is on using the LFF architecture to prevent Q-networks from fitting irreducible, high-frequency noise in the target values during these updates.



True Q-function | Noisy targets | MLP | LFF, σ = 0.003

Figure 2: **Filtering noise with LFF input embedding.** The bootstrapped targets in RL are a mix between signal and noise. Right side: we fit different networks to these noisy targets and display their predictions. While the MLP overfits, LFF learns the Q-function and ignores almost all of the noise.

3 Reinforcement Learning with Learned Fourier Features

We present a visualization of the noisy target value problem in Figure 2. The target value estimates can be noisy due to stochastic transitions, replay buffer sampling, or unintended generalization across state-action pairs due to function approximation [48]. We simulate this in Figure 2 by adding noise to the optimal Q-function of a small gridworld. MLPs are susceptible to fitting the noise, resulting in inaccurate Q-values that could diverge after repeated bootstrapping. In contrast, our LFF architecture controls how quickly low- and high-frequency signals are learned. Tuned properly, LFF filters out the noise and learns the ground truth Q-function almost perfectly (Figure 2, right).

The problem is that MLPs provide no control over how quickly they learn signals of different frequencies. Prior work in computer vision found this a problem when MLPs blurred desired high frequencies in low-dimensional (3-5 dimensions) graphics regression problems [28, 42]. They fixed this blurring problem by transforming the input using a random Fourier feature embedding of the input [34, 42]. Specifically, the idea is to map a low-dimensional input x to an embedding $\gamma(x) = \sin(2\pi Bx)||\cos(2\pi Bx)$, where B is a $d_{\text{fourier}}/2 \times d_{\text{input}}$ matrix, $||$ denotes concatenation, and $\sin$ and $\cos$ act elementwise. The embedding $\gamma(x)$ directly provides a mix of low- and high-frequnecy functions a MLP f_θ can use to learn a desired function. Tancik et al. [42] use this to improve fidelity in coordinate-based graphics problems.

Intuitively, the row vectors b_i are responsible for capturing desired frequencies of the data. If they capture only low frequencies, then the MLP will be biased towards only learning the low-frequency signals in the data. Conversely, if the Fourier features capture sufficient high-frequency features, then a MLP can fit high frequency functions by computing simple nonlinear combinations of the features. In these low-dimensional graphics problems, initializing fixed entries $B_{ij} \sim \mathcal{N}(0, \sigma^2)$ with large σ was enough to learn desired high frequency functions; training B did not improve performance.

In the following section, we propose our learned Fourier feature architecture for deep RL, which allows practitioners to *tune the range of frequencies that the network should be biased towards learning*. We propose several key enhancements that help Fourier features learn in high-dimensional environments. Our work uses learned Fourier features to improve deep RL, in contrast to prior work focused on simple environments with fixed, hand-designed Fourier features and linear function approximation [20, 21]. Although we focus on prioritizing low-frequency signals to reduce bootstrap noise, we also present cases in Appendix F.1 where biasing networks towards high-frequency learning with learned Fourier features enables fast convergence and high asymptotic performance in RL.

3.1 Learned Fourier Feature Architecture

Standard MLPs can be written as the repeated, alternating composition of affine transformations $L_i(x) = W_i x + b_i$ and nonlinearity τ, which is usually the ReLU $\tau(x) = \max(0, x)$:

$$f_\theta(x) = L_n \circ \tau \circ L_{n-1} \circ \tau \circ \cdots \circ L_1(x) \tag{1}$$

We propose a novel architecture based on Fourier features, shown in Figure 1. We define a new layer:

$$F_B(x) = \sin(2\pi Bx)||\cos(2\pi Bx)||x \tag{2}$$

where B is a $d_{\text{fourier}}/2 \times d_{\text{input}}$ matrix and $||$ denotes concatenation. d_{fourier} is a hyperparameter that controls the number of Fourier features we can learn; increasing d_{fourier} increases the degree to which the model relies on the Fourier features. Following Tancik et al. [42], we initialize the entries $B_{ij} \sim N(0, \sigma^2)$, where σ^2 is a hyperparameter. Contrary to prior work, B is a trainable parameter.

The resulting LFF MLP can be written:

$$f_\theta = L_n \circ \tau \circ \cdots \circ L_1 \circ F_B(x) \tag{3}$$

We can optimize this the same way we optimize a standard MLP, e.g. regression would be:

$$\arg\min_{\theta, B} \sum_{i=1}^{N} (L_n \circ \cdots \circ F_B(x_i) - y_i)^2 \tag{4}$$

We propose two key improvements to random Fourier feature input embeddings [34, 42]: training B and concatenating the input x to the Fourier features. We hypothesize that these changes are necessary to preserve information in high-dimensional RL problems, where it is increasingly unlikely that randomly initialized B produces Fourier features well-suited for the task. Training B alleviates this problem by allowing the network to discover them on its own. Appendix G shows that training B does change its values, but its variance remains quite similar to the initial σ^2. This indicates that σ remains an important knob that controls the network behavior throughout training. Concatenating x to the Fourier features is another key improvement for high dimensional settings. It preserves all the input information, which has been shown to help in RL [38]. We further analyze these improvements in Section 6.4.

Algorithm 1 LFF PyTorch-like pseudocode.

```
class LFF():
    def __init__(self, input_size, output_size,
                 n_hidden=1, hidden_dim=256,
                 sigma=1.0, f_dim=256):
        # create B
        b_shape = (input_size, f_dim // 2)
        self.B = Parameter(normal(zeros(*b_shape),
                           sigma * ones(*b_shape)))
        # create rest of network
        self.mlp = MLP(in_dims=f_dim + input_size,
                       out_dims=output_size,
                       n_hidden=n_hidden,
                       hidden_dim=hidden_dim)
    def forward(self, x):
        proj = (2 * np.pi) * matmul(x, self.B)
        ff = cat([sin(proj), cos(proj), x], dim=-1)
        return self.mlp.forward(ff)
```

`normal`: sample from Gaussian with specified mean and std dev; `matmul`: matrix multiplication; `cat`: concatenation.

4 Theoretical Analysis

By providing periodic features (which are controlled at initialization by the variance σ^2), LFF biases the network towards fitting a desired range of frequencies. In this section, we hope to understand why the LFF architecture provably controls the rate at which various frequencies are learned. We draw upon linear neural network approximations in the infinite width limit, which is known as the neural tangent kernel approach [18]. This approximation allows us to understand the training dynamics of the learned neural network output function. While NTK analysis has been found to diverge from real-world behavior in certain cases, particularly for deeper convolutional neural networks [2, 10], it has also been remarkably accurate in predicting directional phenomena [4, 5, 42].

We provide background on the NTK in Section 4.1, and discuss the connection between the eigenvalues of the NTK kernel matrix and the rate at which different frequencies are learned in Section 4.2. We then analyze the NTK and frequency learning rate for Fourier features in networks with 2 layers (Section 4.3) or more (Section 4.4).

4.1 Neural Tangent Kernel

We can approximate a neural network using a first-order Taylor expansion around its initialization θ_0:

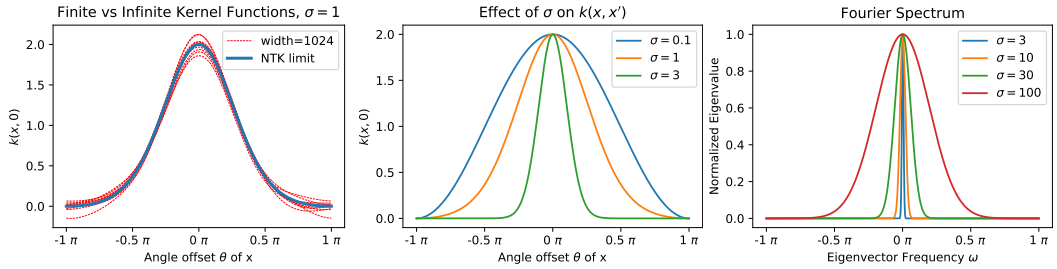
$$f_\theta(x) \approx f_{\theta_0}(x) + \nabla_\theta f_{\theta_0}(x)(\theta - \theta_0) \tag{5}$$

The Neural Tangent Kernel [18] line of analysis makes two further assumptions: f_θ is an infinitely wide neural network, and it is trained via gradient flow. Under the first condition, a trained network stays very close to its initialization, so the Taylor approximation is good (the so-called "lazy training" regime [8]). Furthermore, $f_{\theta_0}(x) = 0$ in the infinite width limit, so our neural network is simply a linear model over features $\phi(x) = \nabla_\theta f_{\theta_0}(x)$. This gives rise to the kernel function:

$$k(x_i, x_j) = \langle \nabla_\theta f_{\theta_0}(x_i), \nabla_\theta f_{\theta_0}(x_j) \rangle \tag{6}$$

The kernel function k is a similarity function: if $k(x_i, x_j)$ is large, then the predictions $f_\theta(x_i)$ and $f_\theta(x_j)$ will tend to be close. This kernel function is deterministic and does not change over the course of training, due to the infinite width assumption. If we have n training points (x_i, y_i), k defines a PSD kernel matrix $K \in \mathbb{R}_+^{n \times n}$ where each entry $K_{ij} = k(x_i, x_j)$. Fascinatingly, when we train this infinite width neural network with gradient flow on the squared error, we precisely know the model output at any point in training. At time t, we have training residual:

$$f_{\theta_t}(x) - y = e^{-\eta K t}(f_{\theta_0}(x) - y) \tag{7}$$



(a) Quality of NTK approximation (b) NTK kernel function (c) Eigenvalue spectrum

Figure 3: **NTK of the 2-layer Fourier feature model.** We plot the NTK $k(x, 0)$ for points x that lie on the unit circle. In (a) and (b), $\theta \in [-\pi, \pi]$ denotes the offset from a reference point with $\theta_0 = 0$. Note that our NTK is shift invariant, so these figures are valid for any reference point θ_0. **Left:** we compare the NTK infinite-width limit to the kernel function of 10 randomly initialized 2-layer Fourier feature networks with width 1024. The NTK limit is quite accurate for realistically wide networks. **Middle:** NTK kernel function k for varying settings of the Fourier feature variance σ^2. Larger σ enable sharp, local learning, while smaller σ induce smoother function learning. **Right:** we compute the NTK kernel matrix K, then find its eigenvalues with the discrete Fourier transform. The y-axis shows eigenvalues, and the x-axis indicates the corresponding frequencies. With low σ, only the lowest frequencies have nonvanishing eigenvalues, so they are the only ones learned through training. Increasing σ here increases the higher frequencies' eigenvalues, so they can be learned faster.

where $f_{\theta_t}(x)$ is the column vector of model predictions for all x_i, and y is the column vector of stacked training labels (see Appendix A.1 for proof sketch). Eq. 7 is critical because it describes how different components of the training loss decrease over time. Section 4.2 will build on this result and examine the training residual in the eigenbasis of the kernel matrix K. This analysis will reveal that each frequency present in the labels y will be learned at its own rate, determined by K's eigenvalues.

4.2 Eigenvalues of the LFF NTK

Consider applying the eigendecomposition of $K = Q\Lambda Q^*$ to Eq. 7, noting that $e^{-\eta K t} = Qe^{-\eta\Lambda}Q^*$ as the matrix exponential is defined $e^X := \sum_{k=0}^{\infty} \frac{1}{k!}X^k$, so Q and Q^* will repeatedly cancel in the middle of X^k since Q is unitary.

$$Q^*(f_{\theta_t}(x) - y) = e^{-\eta\Lambda t}Q^*(f_{\theta_0}(x) - y) \tag{8}$$

Note that the ith component of the residual $Q^*(f_{\theta_t}(x) - y)$ decreases with rate $e^{-\eta\lambda_i}$.

Consider the scenario where the training inputs x_i are evenly spaced on the d-dimensional sphere $\mathbb{S}^{d-1}$. When k is isotropic, which is true for most networks whose weights are sampled isotropically, the kernel matrix K is circulant (each row is a shifted version of the row above). In this special case, K's eigenvectors correspond to frequencies from the discrete Fourier transform (DFT), and the corresponding eigenvalues are the DFT values of the first row of K (see Appendix A.2). Combining this fact with Eq. 8, where we looked at the residual in K's eigenbasis, shows that each frequency in the targets is learned at its own rate, determined by the eigenvalues of K. For a ReLU MLP, these eigenvalues decay approximately quadratically with the frequency [4]. This decay rate is rather slow, so MLPs often fit undesirable medium and high frequency signals. **We hypothesize that LFF controls the frequency-dependent learning rate by tuning the kernel matrix K's eigenvalues.** We examine LFF's kernel matrix eigenvalues in Sections 4.3 and 4.4 and verify this hypothesis.

4.3 NTK Analysis of 2-layer Network with Fourier Features

To simplify our LFF NTK analysis, we consider a two layer neural network $f : \mathbb{R}^d \to \mathbb{R}$:

$$f(x) = \sqrt{\frac{2}{m}} W^\top \begin{bmatrix} \sin(Bx) \\ \cos(Bx) \end{bmatrix} \tag{9}$$

where each row of B is a vector $b_i^\top \in \mathbb{R}^{1\times d}$, and there are m rows of B. $W_i \sim \mathcal{N}(0, 1)$ and $B_{ij} \sim \mathcal{N}(0, \sigma^2)$, where σ is a hyperparameter. Note that concatenating x is omitted for this two-layer

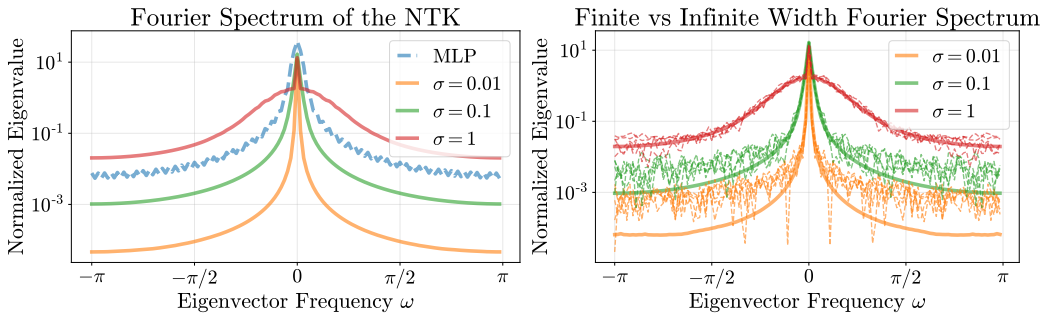


Figure 4: Left: we compare the NTK eigenvalue spectrum (which determines the frequency-specific learning rate) of deep networks with Fourier features to those of a vanilla MLP. Right: we initialize finite-width LFF networks with 2 hidden layers of 1024 units each and compare their kernels (dashed) to the corresponding NTK infinite-width limit (solid). We find that the NTK is accurate (note the log scale) and that decreasing σ indeed results in lower convergence rates for higher frequencies.

model. This is because any contribution from concatenation goes to zero as we increase the layer width m. Lemma 1 determines an analytical expression for the LFF kernel function $k(x, x')$.

Lemma 1. *For $x, x' \in \mathbb{S}^{d-1}$ with angle $\theta = \cos^{-1}(x^\top x')$, we have the NTK kernel function:*

$$k(x, x') = \left(2 - \frac{\|x - x'\|_2^2}{2}\right) \exp\left\{-\frac{\sigma^2}{2}\|x - x'\|_2^2\right\} \tag{10}$$

Proof: see Appendix A.3. This closed form expression for $k(x, x')$ elucidates several desirable properties of Fourier features. σ directly controls the rate of the exponential decay of k, which is the similarity function for points x and x'. For large σ, $k(x, x')$ rapidly goes to 0 as x and x' get farther apart, so their labels only affect the learned function output in a small local neighborhood. This intuitively corresponds to high-frequency learning. In contrast, small σ ensures $k(x, x')$ is large, even when x and x' are relatively far apart. This induces smoothing behavior, inhibiting high-frequency learning. We plot the NTK for varying levels of σ in Figure 3(b) and show that σ directly controls the frequency learning speed in Figure 3(c). Figure 3(a) also verifies that the NTK limit closely matches the empirical behavior of realistically sized networks at initialization.

4.4 NTK of Deeper Networks

Figure 3(c) shows that larger initialization variance σ^2 corresponds to larger eigenvalues for high frequencies in the 2-layer model. This matches empirical results that small σ leads to underfitting and large σ leads to overfitting [42]. However, Figure 3(c) indicates that only extremely large σ, on the order of $10^2 - 10^3$, result in coverage of the high frequencies. This contradicts Tancik et al. [42], who fit fine-grained image details with $\sigma \in [1, 10]$. We suggest that the 2-layer model, even though it accurately predicts the directional effects of increasing or decreasing σ, fails to accurately model learning in realistically sized networks. Manually computing the kernel functions of deeper MLPs with Fourier feature input embeddings is difficult. Thus, we turn to Neural Tangents [30], a library that can compute the kernel function of any architecture expressible in its API .

We initialize random Fourier features of size 1024 with different variances σ^2 and build an infinite-width ReLU MLP on top with 3 hidden layers using the Neural Tangents library. As we did in Figure 3, we take input data x that is evenly spaced on the 2D unit circle and evaluate the corresponding kernel function $k(x, 0)$ between the point $(1, 0)$ and the point $x = (\cos\theta, \sin\theta)$. Figure 4 shows the eigenvalues of Fourier features and vanilla MLPs in this scenario. We see the same general trend, where increasing σ leads to larger eigenvalues for higher frequencies, as we did in Figure 3. Furthermore, Figure 4 also shows that this trend also holds for the exact finite-width architectures that we use in our experiments (Section 6). These results now reflect the empirical behavior where $\sigma \in [1, 10]$ results in high frequency learning. This indicates that deeper networks are crucial for understanding the behavior of Fourier features.

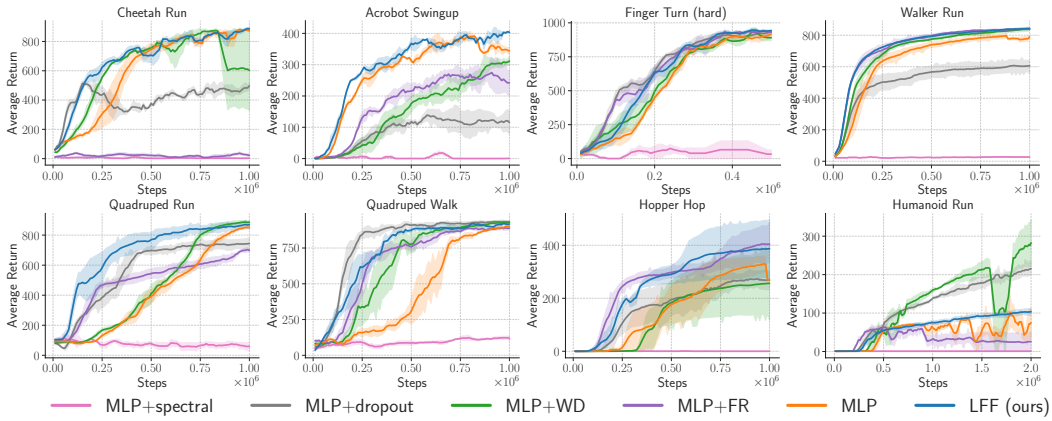


Figure 5: **Off-policy State-based Evaluation**: Soft Actor Critic (SAC) experiments on 8 DM Control environments. We emphasize that these results are produced using the same hyperparameters (e.g. learning rate, Polyak averaging parameter, and batch size) tuned for MLPs. These results show that plugging in our LFF architecture can yield more sample-efficient learning on most environments.

5 Experimental Setup

We treat the learned Fourier feature network as a drop-in replacement for MLP and CNN architectures. We show that just adding Fourier features improves the performance of *existing* state-of-the-art methods on *existing* standard benchmark environments from DeepMind Control Suite [43]. We will release the code, **which involves only changing a few lines of code in existing RL algorithms**.

State-based LFF Architecture Setup We use soft actor-critic (SAC), an entropy-regularized off-policy RL algorithm [14], to learn 8 environments from the DeepMind Control Suite [43]. We keep the default hyperparameters fixed, varying only the architecture for the policy and Q-function. Our LFF architecture uses our learnable Fourier feature input layer, followed by 2 hidden layers of 1024 units. We use Fourier dimension d_{fourier} of size 1024. We initialize the entries of our trainable Fourier basis with $B_{ij} \sim \mathcal{N}(0, \sigma^2)$, with $\sigma = 0.01$ for all environments except Cheetah, Walker, and Hopper, where we use $\sigma = 0.001$. To make the parameter count roughly equal, we compare against an MLP with three hidden layers. The first MLP hidden layer is slightly wider, about 1100 units, to compensate for the extra parameters in LFF's first layer due to input concatenation. Learning curves are averaged over 5 seeds, with the shaded region denoting 1 standard error.

Image-based LFF Architecture Setup We test image-based learning on 4 DeepMind Control Suite environments [43] with SAC + RAD [24], which uses data augmentation to improve the sample efficiency of image-based training. The vanilla RAD architecture, which uses the convolutional architecture from Srinivas et al. [39], is denoted as "CNN" in Figure 6. To apply LFF to images, we observe that computing Bx at each pixel location is equivalent to a 1x1 convolution without bias. This 1x1 convolution maps the the RGB channels at each pixel location from 3 dimensions to $d_{\text{fourier}}/2$ channels. We then compute the $\sin$ and $\cos$ of those channels and concatenate the original RGB values, so our image goes from $H \times W \times 3$ to a $H \times W \times (d_{\text{fourier}} + 3)$ embedding. The 1x1 conv weights are initialized from $\mathcal{N}(0, \sigma^2)$ with $\sigma = 0.1$ for Hopper and Cheetah and $\sigma = 0.01$ for Finger and Quadruped. As we did in the state-based setup, we make the CNN baseline fair by adding an additional 1x1 convolution layer at the beginning. This ensures that the "CNN" and "CNN+LFF" architectures have the same parameter count, and that performance gains are solely due to LFF.

6 Results

We provide empirical support for the approach by investigating the following questions:

1. Does LFF improve the sample efficiency of off-policy state-based or image-based RL?
2. Do learned Fourier features make the Bellman update more stable?
3. Does LFF help more when applied to the policy or the Q-function?
4. Ablation: How important is input concatenation or training the Fourier basis B?

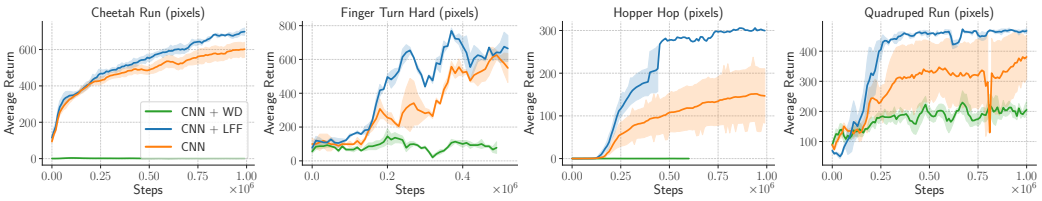


Figure 6: **Off-policy Image-based Evaluation**: SAC experiments on learning 4 DMControl environments from pixels. LFF can yield dramatic improvements in sample-efficiency over CNNs.

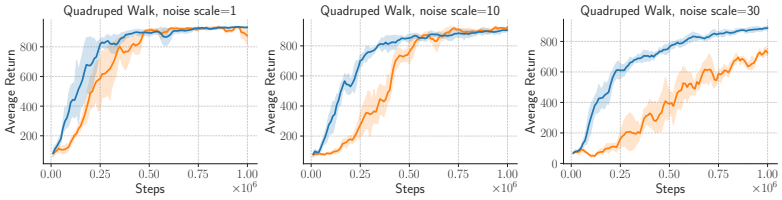


Figure 7: **State-based with Added Noise**: We add zero-mean Gaussian noise to the targets. As the standard deviation of the added noise increases, LFF maintains its performance better than MLPs.

6.1 LFF Architecture for Off-policy RL

We show the results of using the LFF architecture for state-based RL with SAC in Figure 5. LFF does clearly better than MLPs in 6 out of 8 environments, and slightly better in the remaining 2. Figure 6 shows even stronger results on image-based RL with SAC and RAD. This is especially promising because these results use the hyperparameters that were tuned for the MLP or CNN baseline. We find that the return consistently starts increasing much earlier with the LFF architecture. We hypothesize that LFF reduces noise propagation due to bootstrapping, so less data is required to overcome incorrect targets. SAC can use these more accurate Q-values to quickly begin exploring high-reward regions of the MDP.

For the state-space experiments, we also test several baselines:

- MLP with weight decay, tuned over the values $\{10^{-3}, 3 \times 10^{-4}, 10^{-4}, 3 \times 10^{-4}, 10^{-5}\}$. Weight decay helps learning in most environments, but it can hurt performance (Acrobot, Hopper) or introduce instability (Cheetah). Weight decay strong enough to reduce overfitting may simultaneously bias the Q-values towards 0 and cause underestimation bias.
- MLP with dropout [40]. We add a dropout layer after every nonlinearity in the MLP. We search over $[0.05, 0.2]$ for the drop probability, and find that lower is better. Dropout does help in most environments, although occasionally at the cost of asymptotic performance.
- MLP with functional regularization [31]. Instead of using a target network to compute target values, we use the current Q-network, but regularize its values from diverging too far from the Q-values from a lagging snapshot of the Q-network.
- MLP with spectral normalization [13]. We add spectral normalization to the second-to-last layer of the network, as is done in [13], but find that this works very poorly. It is likely necessary to tune the other hyperparameters (learning rate, target update frequency, Polyak averaging) in order to make spectral normalization work.

Overall, LFF consistently ranks around the top across all of the environments. It can be combined with weight decay, dropout, or functional regularization for more gains, and has a simple plug-and-play advantage because a single set of parameters works over all environments.

6.2 Do learned Fourier features improve the stability of the Bellman updates?

Our key problem is that standard ReLU MLP Q-functions tend to fit the noise in the target values, introducing error into the Q-function at some (s, a). Bootstrapping with this incorrect Q-value to calculate target values for other (s', a') yields even noisier targets, propagates the error to other states, and causes instability or divergence (see Appendix D for more details). To further test whether

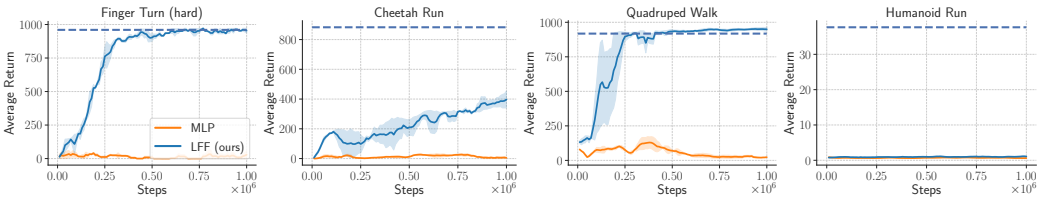


Figure 8: **Effect of LFF on Stability of Bootstrapping**: We train SAC, foregoing a target network, by bootstrapping directly from the Q-network being trained. The dashed line shows the LFF performance with a target network after 1M steps. We find that the LFF network is remarkably stable, and even learns faster on Quadruped Walk than when using target networks. However, LFF fails to learn on Humanoid, indicating that higher dimensional problems still pose problems.

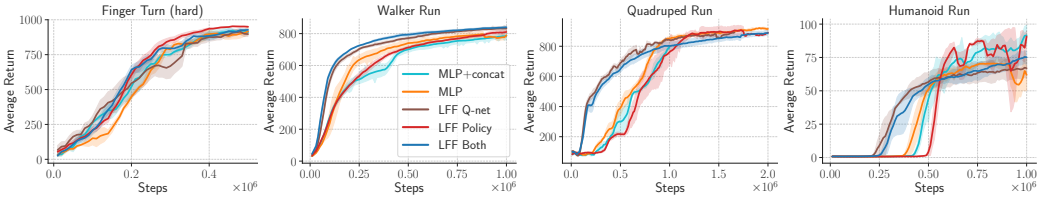


Figure 9: **LFF Policy vs Q-function.** Walker and Quadruped results indicate that only using LFF for the Q-network is just as good as using LFF for both networks. In contrast, using LFF for the policy network is about as bad as the MLP baseline. This suggests that LFF primarily improves off-policy learning by reducing noise in the Q-network optimization.

LFF solves this problem by filtering out the noise, we train a SAC agent on state-based DMControl environments with either of the following modifications: testing the Q-function's robustness by adding Gaussian noise to the targets, or removing target networks altogether.

Gaussian noise added to targets In each bootstrap step, we add zero-mean Gaussian noise with standard deviation 1, 10, or 30 to the targets. LFF maintains higher performance even at significant noise levels, indicating that it is more robust to bootstrap noise. Full results are in Figure 12.

No target network Target networks, updated infrequently, slow down the propagation of noise due to bootstrapping [29]. LFF fits less noise to begin with, so it should work even when the target network is omitted. Here, we bootstrap directly from the network being trained. Figure 8 shows that MLPs consistently fail to learn on all environments in this setting, while the LFF architecture still performs well, except when the problem is very high dimensional. LFF even manages to learn faster in Quadruped Walk than it does when using a target network, since there is no longer Polyak averaging [25] with a target to slow down information propagation. Omitting the target network allows us to use updated values for $Q_\theta(s', a')$, instead of stale values from the target network. This result is in line with recent work that achieves faster learning by removing the target network and instead penalizing large changes to the Q-values [37].

Overall, Figure 7 and 8 validate our theoretical claims that LFF controls the effect of high-frequency noise on the learned function, and indicates that LFF successfully mitigates bootstrap noise in most cases. Tuning the SAC hyperparameters should increase LFF sample efficiency even further, since we can learn more aggressively when the noise problem is reduced.

6.3 Where Do Learned Fourier Features Help?

In this section, we confirm that our LFF architecture improves RL performance by primarily preventing the Q-network from fitting noise. We train state-based SAC with an MLP policy and LFF Q-network (LFF Q-net in Figure 9), or with an LFF policy and MLP Q-network (LFF Policy). Figure 9 shows that solely regularizing the Q-network is sufficient for LFF's improved sample efficiency. This validates our hypothesis that the Bellman updates remain noisy, even with tricks like double Q-networks and Polyak averaging, and that LFF reduces the amount of noise that the Q-network accumulates. These results suggest that separately tuning σ for the Q-network and policy networks may yield further improvements, as they have separate objectives. The Q-network should be resilient

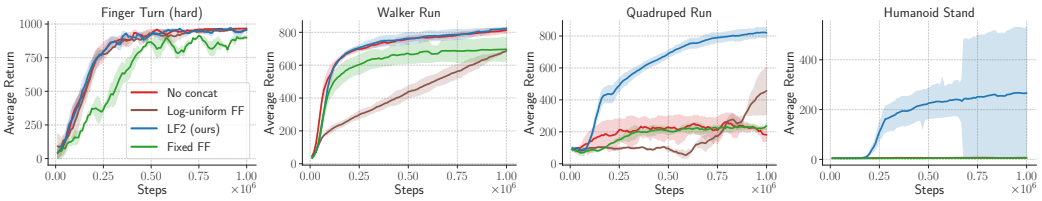


Figure 10: **Ablation analysis**: We train SAC on 4 DMControl environments with three variants of our architecture: LFF, LFF with fixed Fourier features, and LFF without input concatenation. Lower dimensional environments like Finger and Walker are more forgiving for fixed Fourier features or omitting input concatenation. However, Humanoid absolutely requires both modifications to learn.

to noise, while the policy can be fine-grained and change quickly between nearby states. However, for simplicity, we use LFF for the Q-networks and the policy networks. Finally, we also train vanilla MLPs where we concatenate the input x to the first layer output. LFF outperforms this variant, confirming that concatenation is not solely responsible for the improved sample efficiency.

6.4 Architectural Ablations

LFF features two key improvements: learning the Fourier feature basis B and concatenating the input x to the Fourier features. We perform an ablation on several DMControl environments with SAC in Figure 10 to investigate the impact of these modifications.

We first find that training the Fourier feature basis B is critical. Across all four environments, learning is impacted by using a fixed, randomly initialized B. This is because finding the right Fourier features at initialization is unlikely in our high dimensional RL problems. Training B allows the network to discover the relevant Fourier features on its own. The relationship with dimension is clear: as the input dimension increases, the performance gap between LFF and fixed Fourier features grows.

Concatenating the input x to the Fourier features is also important. It maintains all of the information that was present in x, which is critical in very high dimensional environments. If the Fourier basis B is poorly initialized and it blends together or omits important dimensions of x, the network takes a long time to disentangle them, if at all. This problem becomes more likely as the observation dimension increases. While LLF can learn without concatenation in low-dimensional environments like Walker and Finger, it has a much harder time learning in Quadruped and Humanoid.

Finally, we test an alternative approach to initializing the values of B. B, which has shape $(k \cdot d_{\text{input}}) \times d_{\text{input}}$, is now initialized as $B = (I, cI, c^2 I, \ldots, c^{k-1} I)^\top$ where I is the identity matrix, k is an integer, and $0 < c < 1$ is a tuned multiplier. This parallels the axis-aligned, log-uniform spacing used in NeRF's positional encoding [28], but we additionally concatenate x and train B. We find that this initialization method, dubbed "Log-uniform FF" in Figure 10, is consistently worse than sampling from $\mathcal{N}(0, \sigma^2)$. This is likely because the initialization fails to capture features that are not axis-aligned, so most of the training time is used to discover the right combinations of input features.

7 Conclusions and Future Work

We highlight that the standard MLP or CNN architecture in state-based and image-based deep RL methods remain susceptible to noise in the Bellman update. To overcome this, we proposed embedding the input using *learned* Fourier features. We show both theoretically and empirically that this encoding enables fine-grained control over the network's frequency-specific learning rate. Our LFF architecture serves as a plug-and-play addition to any state-of-the-art method and leads to consistent improvement in sample efficiency on standard state-space and image-space environments.

One shortcoming of frequency-based regularization is that it does not help when the noise and the signal look similar in frequency space. Future work should examine when this is the case, and test whether other regularization methods are complementary to LFF. Another line of work, partially explored in Appendix F, is using LFF with large σ to fit high frequencies and reduce underfitting in model-based or tabular reinforcement learning scenarios. We hope this work will provide new perspectives on existing RL algorithms for the community to build upon.

Acknowledgments We thank Shikhar Bahl, Murtaza Dalal, Wenlong Huang, and Aravind Sivakumar for comments on early drafts of this paper. The work was supported in part by NSF IIS-2024594 and GoodAI Research Award. AL is supported by the NSF GRFP. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE1745016 and DGE2140739. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

References

[1] J. Achiam, E. Knight, and P. Abbeel. Towards characterizing divergence in deep q-learning. *arXiv preprint arXiv:1903.08894*, 2019. 17, 18

[2] Z. Allen-Zhu and Y. Li. What can resnet learn efficiently, going beyond kernels? *arXiv preprint arXiv:1905.10337*, 2019. 4

[3] B. Bamieh. Discovering transforms: A tutorial on circulant matrices, circular convolution, and the discrete fourier transform. *arXiv preprint arXiv:1805.05533*, 2018. 15

[4] R. Basri, D. Jacobs, Y. Kasten, and S. Kritchman. The convergence rate of neural networks for learned functions of different frequencies. *arXiv preprint arXiv:1906.00425*, 2019. 4, 5, 23

[5] R. Basri, M. Galun, A. Geifman, D. Jacobs, Y. Kasten, and S. Kritchman. Frequency bias in neural networks for input of non-uniform density. In *International Conference on Machine Learning*, pages 685–694. PMLR, 2020. 4

[6] R. Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966. 2

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. 19

[8] L. Chizat, E. Oyallon, and F. Bach. On lazy training in differentiable programming. *arXiv preprint arXiv:1812.07956*, 2018. 4

[9] F. Farnia, J. Zhang, and D. Tse. A spectral approach to generalization and optimization in neural networks. 2018. 23

[10] S. Fort, G. K. Dziugaite, M. Paul, S. Kharaghani, D. M. Roy, and S. Ganguli. Deep learning versus kernel learning: an empirical study of loss landscape geometry and the time evolution of the neural tangent kernel. *arXiv preprint arXiv:2010.15110*, 2020. 4

[11] J. Fu, A. Kumar, M. Soh, and S. Levine. Diagnosing bottlenecks in deep q-learning algorithms. In *International Conference on Machine Learning*, pages 2021–2030. PMLR, 2019. 23

[12] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018. 2

[13] F. Gogianu, T. Berariu, M. Rosca, C. Clopath, L. Busoniu, and R. Pascanu. Spectral normalisation for deep reinforcement learning: an optimisation perspective. *arXiv preprint arXiv:2105.05246*, 2021. 8

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018. 7, 26, 27

[15] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018. 26

[16] H. Hasselt. Double q-learning. *Advances in neural information processing systems*, 23:2613–2621, 2010. 1, 2

[17] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970. 1

[18] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. *arXiv preprint arXiv:1806.07572*, 2018. 2, 4

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 26, 27

[20] J. Z. Kolter, A. Y. Ng, et al. Learning omnidirectional path following using dimensionality reduction. In *Robotics: Science and Systems*, pages 27–30, 2007. 3

[21] G. Konidaris, S. Osentoski, and P. Thomas. Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, 2011. 3

[22] A. Krogh and J. A. Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992. 1

[23] A. Kumar, R. Agarwal, D. Ghosh, and S. Levine. Implicit under-parameterization inhibits data-efficient deep reinforcement learning. *arXiv preprint arXiv:2010.14498*, 2020. 21, 22

[24] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020. 7, 27

[25] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 9

[26] Z. Liu, X. Li, B. Kang, and T. Darrell. Regularization matters in policy optimization–an empirical study on continuous control. *arXiv preprint arXiv:1910.09191*, 2019. 2

[27] A. McCallum. Reinforcement learning with selective perception and hidden state [ph. d. thesis]. *Computer Science Department, University of Rochester*, 1996. 23

[28] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision*, pages 405–421. Springer, 2020. 3, 10

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 9

[30] R. Novak, L. Xiao, J. Hron, J. Lee, A. A. Alemi, J. Sohl-Dickstein, and S. S. Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. *arXiv preprint arXiv:1912.02803*, 2019. 6

[31] A. Piché, J. Marino, G. M. Marconi, C. Pal, and M. E. Khan. Beyond target networks: Improving deep q-learning with functional regularization. *arXiv preprint arXiv:2106.02613*, 2021. 8

[32] D. Precup, R. S. Sutton, and S. Dasgupta. Off-policy temporal-difference learning with function approximation. In *ICML*, pages 417–424, 2001. 2

[33] N. Rahaman, D. Arpit, A. Baratin, F. Draxler, M. Lin, F. A. Hamprecht, Y. Bengio, and A. C. Courville. On the spectral bias of deep neural networks. 2018. 23

[34] A. Rahimi, B. Recht, et al. Random features for large-scale kernel machines. In *NIPS*, volume 3, page 5. Citeseer, 2007. 2, 3, 4

[35] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015. 19

[36] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 19, 27

[37] L. Shao, Y. You, M. Yan, Q. Sun, and J. Bohg. Grac: Self-guided and self-regularized actor-critic. *arXiv preprint arXiv:2009.08973*, 2020. 9

[38] S. Sinha, H. Bharadhwaj, A. Srinivas, and A. Garg. D2rl: Deep dense architectures in reinforcement learning. *arXiv preprint arXiv:2010.09163*, 2020. 4

[39] A. Srinivas, M. Laskin, and P. Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020. 7

[40] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15 (1):1929–1958, 2014. 1, 8

[41] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 1, 2

[42] M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron, and R. Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *arXiv preprint arXiv:2006.10739*, 2020. 2, 3, 4, 6

[43] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018. 2, 7

[44] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, pages 255–263. Hillsdale, NJ, 1993. 1

[45] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. 1

[46] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690, 1997. 1

[47] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning, 2015. 1, 2

[48] H. Van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*, 2018. 3

[49] Z.-Q. J. Xu, Y. Zhang, T. Luo, Y. Xiao, and Z. Ma. Frequency principle: Fourier analysis sheds light on deep neural networks. *arXiv preprint arXiv:1901.06523*, 2019. 23

[50] Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007. 1

[51] D. Yarats and I. Kostrikov. Soft actor-critic (sac) implementation in pytorch. `https://github.com/denisyarats/pytorch_sac`, 2020. 26

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [No]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] See Appendix A
   (b) Did you include complete proofs of all theoretical results? [Yes] See Appendix A
3. If you ran experiments...

(a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Code is provided in the supplemental material

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Hyperparameters are listed in Section 5 and Appendix H

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

   (a) If your work uses existing assets, did you cite the creators? [Yes]

   (b) Did you mention the license of the assets? [Yes]

   (c) Did you include any new assets either in the supplemental material or as a URL? [No]

   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Theoretical Analysis

A.1 Functional Convergence Rate

We provide Lemma 2 and a quick proof sketch as background for readers who are not familiar with the neural tangent kernel literature.

Lemma 2. *When training an infinite width neural network via gradient flow on the squared error, the training residual at time t is:*

$$f_{\theta_t}(x) - y = e^{-\eta K t}(f_{\theta_0}(x) - y) \tag{11}$$

where $f_{\theta_t}(x)$ is the column vector of model predictions for all x_i, y is the column vector of stacked training labels, η is the multiplier for gradient flow, and K is the NTK kernel matrix with $K_{ij} = \langle \nabla_\theta f_{\theta_0}(x_i), \nabla_\theta f_{\theta_0}(x_j) \rangle$.

Proof. The squared error $L(\theta, x) = \|f_\theta(x) - y\|_2^2$ has gradient:

$$\nabla_\theta L(\theta, x) = \nabla_\theta f_\theta(x)^\top (f_\theta(x) - y) \tag{12}$$

Since we train with gradient flow, the parameters change at rate:

$$\frac{d\theta_t}{dt} = -\eta \nabla_{\theta_t} L(\theta_t, x) \tag{13}$$

$$= -\eta \nabla_{\theta_t} f_{\theta_t}(x)^\top (f_{\theta_t}(x) - y) \tag{14}$$

By the chain rule,

$$\frac{df_{\theta_t}(x)}{dt} = \frac{df_{\theta_t}(x)}{d\theta_t}\frac{d\theta_t}{dt} \tag{15}$$

$$= -\eta \nabla_{\theta_t} f_{\theta_t}(x) \nabla_{\theta_t} f_{\theta_t}(x)^\top (f_{\theta_t}(x) - y) \tag{16}$$

$$= -\eta K_{\theta_t}(f_{\theta_t}(x) - y) \tag{17}$$

where K is the NTK kernel matrix at time t, with entries $K_{ij} = \langle \nabla_{\theta_t} f_{\theta_t}(x_i), \nabla_{\theta_t} f_{\theta_t}(x_j) \rangle$. Since y is a constant, and $K_{\theta_t} \approx K_{\theta_0} \triangleq K$ due to the infinite-width limit, we can write this as:

$$\frac{d(f_{\theta_t}(x) - y)}{dt} = -\eta K(f_{\theta_t}(x) - y) \tag{18}$$

This is a well-known differential equation, with closed form solution:

$$f_{\theta_t}(x) - y = e^{-\eta K t}(f_{\theta_0}(x) - y) \tag{19}$$

□

A.2 Eigenvalues of the NTK matrix and the Discrete Fourier Transform

Lemma 3. *A circulant matrix $C \in \mathbb{R}^{n \times n}$ with first row $(c_0, \ldots, c_{n-1})$ has eigenvectors $\{x^{(k)}\}_{k=1}^n$ corresponding to the column vectors of the DFT matrix:*

$$x^{(k)} = (\omega_n^{0k}, \omega_n^{1k}, \ldots, \omega_n^{(n-1)k})^\top \tag{20}$$

where $\omega_n = e^{\frac{2\pi i}{n}}$ is the nth root of unity. The corresponding eigenvalue is the kth DFT value $\lambda^{(k)} = DFT(c_0, \ldots, c_{n-1})_k$.

Proof. This is a well-known property of circulant matrices [3]. Nevertheless, we provide a simple proof here. First, let's make clear the structure of the circulant matrix C:

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & \cdots & c_{n-1} \\ c_{n-1} & c_0 & c_1 & \cdots & c_{n-2} \\ c_{n-2} & c_{n-1} & c_0 & \cdots & c_{n-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_1 & c_2 & c_3 & \cdots & c_0 \end{bmatrix} \tag{21}$$

Again, we want to show that an eigenvector of C is $x^{(k)}$, the kth column of the DFT matrix F.

$$x^{(k)} = (\omega_n^{0k}, \omega_n^{1k}, \ldots, \omega_n^{(n-1)k})^\top \tag{22}$$

where $\omega_n = e^{\frac{2\pi i}{n}}$ is the nth root of unity. Note that for all positive integers k, ω_n^{jk} is periodic in j with period n. This is because:

$$\omega_n^{nk} = e^{\frac{2\pi nki}{n}} \tag{23}$$
$$= \cos(2\pi k) + i\sin(2\pi k) \tag{24}$$
$$= 1 \tag{25}$$

Now, let us show that $x^{(k)}$ is an eigenvector of C. Let $y = Cx^{(k)}$. The ith element of y is then

$$y_i = \sum_{j=0}^{n-1} c_{j-i}\omega_n^{jk} \tag{26}$$
$$= \omega_n^{ik} \sum_{j=0}^{n-1} c_{j-i}\omega_n^{(j-i)k} \tag{27}$$

The remaining sum does not depend on i, since c_{j-i} and $\omega_n^{(j-i)k}$ are periodic with period n. This means we can rearrange the indices of the sum to get:

$$y_i = \omega_n^{ik} \sum_{j=0}^{n-1} c_j\omega_n^{jk} \tag{28}$$
$$= x_i^{(k)}\lambda_k \tag{29}$$

where $\lambda_k = \sum_{j=0}^{n-1} c_j\omega_n^{jk}$ is exactly the kth term in the DFT of the signal $(c_0, c_1, \ldots, c_{n-1})$. Thus, $Cx^{(k)} = \lambda_k x^{(k)}$, so $x^{(k)}$ is an eigenvector of C with corresponding eigenvalue $\lambda^{(k)} = DFT(c_0, \ldots, c_{n-1})_k$. □

A.3 Proof of Lemma 1: NTK for 2 layer Fourier feature model

Proof. The gradient consists of two parts: the gradient with respect to B and the gradient with respect to W. We can calculate the NTK for each part respectively and then sum them.

For W:

$$\nabla_W f(x) = \sqrt{\frac{2}{m}} \begin{bmatrix} \sin(Bx) \\ \cos(Bx) \end{bmatrix} \tag{30}$$

The width-m kernel is then:

$$k_m^W(x, x') = \frac{2}{m} \sum_{i=1}^{m/2} \cos(b_i^\top x)\cos(b_i^\top x') + \sin(b_i^\top x)\sin(b_i^\top x') \tag{31}$$

Using the angle difference formula, this reduces to:

$$k_m^W(x, x') = \frac{2}{m} \sum_{i=1}^{m/2} \cos(b_i^\top (x - x')) \tag{32}$$

As $m \to \infty$, this converges to a deterministic kernel $k^W(x, x') = \mathbb{E}_{B_{ij}\sim\mathcal{N}(0,\tau)}[\cos(b_i^\top (x - x'))]$.

Using the fact that $E_{X\sim\mathcal{N}(0,\Sigma)}[\cos(t^\top X)] = \exp\{-\frac{1}{2}t^\top \Sigma t\}$ for fixed vector t and the fact that $\Sigma = \text{diag}(\sigma^2, \sigma^2)$ in our case, this kernel function simplifies to:

$$k^W(x, x') = \exp\left\{-\frac{\sigma^2}{2}\|x - x'\|_2^2\right\} \tag{33}$$

For B:

$$\nabla_{b_i} f(x) = \sqrt{\frac{2}{m}} \left(W_i \cos(b_i^\top x) - W_{i+m/2} \sin(b_i^\top x)\right) x \tag{34}$$

The width-m kernel for B is then:

$$\begin{aligned} k_m^B(x, x') = \frac{2x^\top x'}{m} \sum_{i=1}^{m/2} & W_i^2 \left(\cos(b_i^\top x)\cos(b_i^\top x')\right) + W_{i+m/2}^2 \left(\sin(b_i^\top x)\sin(b_i^\top x)\right) \\ & - W_i W_{i+m/2} \left(\cos(b_i^\top x)\sin(b_i^\top x') + \sin(b_i^\top x)\cos(b_i^\top x')\right) \end{aligned} \tag{35}$$

As we take $m \to \infty$, recall that $W_j \sim \mathcal{N}(0, 1)$ i.i.d.. Thus, $\mathbb{E}[W_j^2] = 1$ and $\mathbb{E}[W_j W_k] = 0$ for $j \neq k$. The NTK is then

$$k^B(x, x') = x^\top x' \mathbb{E}\left[\cos(b_i^\top x)\cos(b_i^\top x') + \sin(b_i^\top x)\sin(b_i^\top x)\right] \tag{36}$$

The interior of the expectation is exactly the same as the summand in Equation 31. Following the same steps, we get the simplified kernel function:

$$k^B(x, x') = x^\top x' \exp\left\{-\frac{\sigma^2}{2}\|x - x'\|_2^2\right\} \tag{37}$$

Finally, our overall kernel function $k(x, x') = k^W(x, x') + k^B(x, x')$ is:

$$k(x, x') = \left(1 + x^\top x'\right) \exp\left\{-\frac{\sigma^2}{2}\|x - x'\|_2^2\right\} \tag{38}$$

Since $x, x' \in \mathbb{S}^{d-1}$, they have unit norm, with $\|x - x'\|_2^2 = 2(1 - x^\top x') = 2(1 - \cos\theta)$. This gives us two equivalent forms of the NTK:

$$k(x, x') = \left(2 - \frac{\|x - x'\|_2^2}{2}\right) \exp\left\{-\frac{\sigma^2}{2}\|x - x'\|_2^2\right\} \tag{39}$$

$$= (1 + \cos\theta) \exp\left\{\sigma^2(\cos\theta - 1)\right\} \tag{40}$$

□

A.4 When is the Bellman Update a Contraction?

Here, we examine LFF's stability under Bellman updates by using results from Achiam et al. [1], who used the NTK approximation to prove that the Bellman update is a contraction in finite MDPs if

$$\forall i, \quad \alpha K_{ii} \rho_i < 1 \tag{41}$$

$$\forall i, \quad (1 + \gamma) \sum_{j \neq i} |K_{ij}| \rho_j \leq (1 - \gamma) K_{ii} \rho_i \tag{42}$$

where K is the NTK of the Q-network and ρ_i is the density of transition i in the replay buffer. Intuitively, K_{ij} measures the amount that a gradient update on transition j affects the function output on transition i. In order for the Bellman update to be a contraction, the change at (s_i, a_i) due to gradient contributions from all other transitions should be relatively small. This suggests that LFF, with very large σ, could fulfill the conditions in Equation 43 and 44. We formalize this in Theorem 1:

Theorem 1. *For a finite MDP with the state-action space as a finite, uniform subset of $\mathcal{S}^d$, and when we have uniform support for each transition in the replay buffer, the Bellman update on a 2 layer LFF architecture is a contraction for suitably small learning rate α.*

Proof. We need our kernel to satisfy two conditions [1]:

$$\forall i, \quad \alpha K_{ii} \rho_i < 1 \tag{43}$$

$$\forall i, \quad (1 + \gamma) \sum_{j \neq i} |K_{ij}| \rho_j \leq (1 - \gamma) K_{ii} \rho_i \tag{44}$$

Equation 43 is easy to satisfy, as we assumed that all transitions appear uniformly in our buffer, and we know from Lemma 1 that $k(x, x) = 2$. Thus, we only need to make the learning rate α small enough such that $\alpha < \frac{1}{2\rho_i}$.

For Equation 44, we can prove a loose lower bound on the variance σ^2 that is required for the Bellman update to be a contraction. As stated in the main text, we assume that we have $N + 1$ datapoints x_i that are distributed approximately uniformly over $\mathbb{S}^{d-1}$. Here, uniformly simply implies that we have an upper bound $x_i^\top x_j = \cos\theta < 1 - \delta$ for all $i \neq j$ and fixed positive $\delta > 0$.

First, we bound $|K_{ij}|$ for all $i \neq j$. Using the expression from Lemma 1 and our upper bound on $\cos\theta$, we have $|K_{ij}| \leq (2 - \delta) \exp\left\{-\delta\sigma^2)\right\}, \forall i \neq j$. Then, plugging this into Equation 44 and cancelling the buffer frequencies ρ_i, which are equal by assumption,

$$N(1 + \gamma)(2 - \delta) \exp\left\{-\delta\sigma^2\right\} \leq 2(1 - \gamma) \tag{45}$$

Rearranging gives us:

$$\sigma^2 \geq \frac{1}{\delta} \log \frac{N(1 + \gamma)(2 - \delta)}{2(1 - \gamma)} \tag{46}$$

As long as α is small enough and $\gamma < 1$, an infinite-width LFF architecture initialized with a suitably large τ will enjoy Bellman updates that are always contractions in the sup-norm. □

Note However, this does not align with what we see in practice, where small σ yields the best performance, and increasing σ leads to worse performance. This is because Achiam et al. [1] makes the assumption that the replay buffer assigns positive probability to every possible transition, which is impossible in our continuous MDPs. We also do stochastic optimization with minibatches, which further deviates from the theory. Finally, guarantee of a contraction does not imply sample efficiency. Indeed, an algorithm that contracts slowly at every step can be much worse than an algorithm that greatly improves in expectation.

B On-policy Results

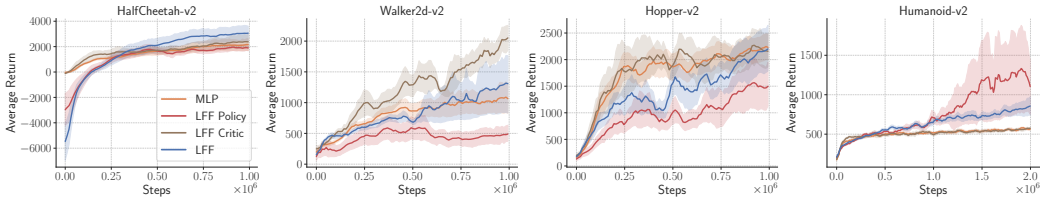


Figure 11: **On-Policy Evaluation**: We train PPO on 4 OpenAI Gym environments with our LFF architecture and a vanilla MLP, both of which have roughly the same number of parameters. LFF does not produce a consistent gain in sample efficiency here, which is consistent with our hypothesis that LFF helps only for off-policy RL.

B.1 On-policy LFF Setup

We evaluate proximal policy optimization (PPO, Schulman et al. [36]) on 4 environments from OpenAI gym [7]. These environments range from easy, e.g. HalfCheetah-v2, to difficult, e.g. Humanoid-v2. Just as we did in the off-policy setup, we modify only the architecture. keeping the hyperparameters fixed. We compare MLPs with 3 hidden layers to LFF with our Fourier feature input layer followed by 2 hidden layers. We use $d_{\text{fourier}} = 1024$ and $\sigma = 0.001$ for all environments and also test what happens if we use LFF for only the policy or only the critic.

B.2 LFF architecture for on-policy RL

In Figure 11, we show the results of using LFF for PPO. Unlike LFF on SAC, LFF did not yield consistent gains for PPO. The best setting was to use LFF for only the critic, which does as well as MLPs on 3 environments (HalfCheetah, Hopper, and Humanoid) and better on Walker2d, but this is only a modest improvement. This is not surprising and is likely because policy gradient methods have different optimization challenges than those based on Q-learning. For one, the accuracy of the value function baseline is less important for policy gradient methods. The on-policy cumulative return provides a lot of reward signal, and the value function baseline mainly reduces variance in the gradient update. In addition, generalized advantage estimation [35] further reduces variance. Thus, noise in the bootstrapping process is not as serious of a problem as in off-policy learning.

C Performance Under Added Noise

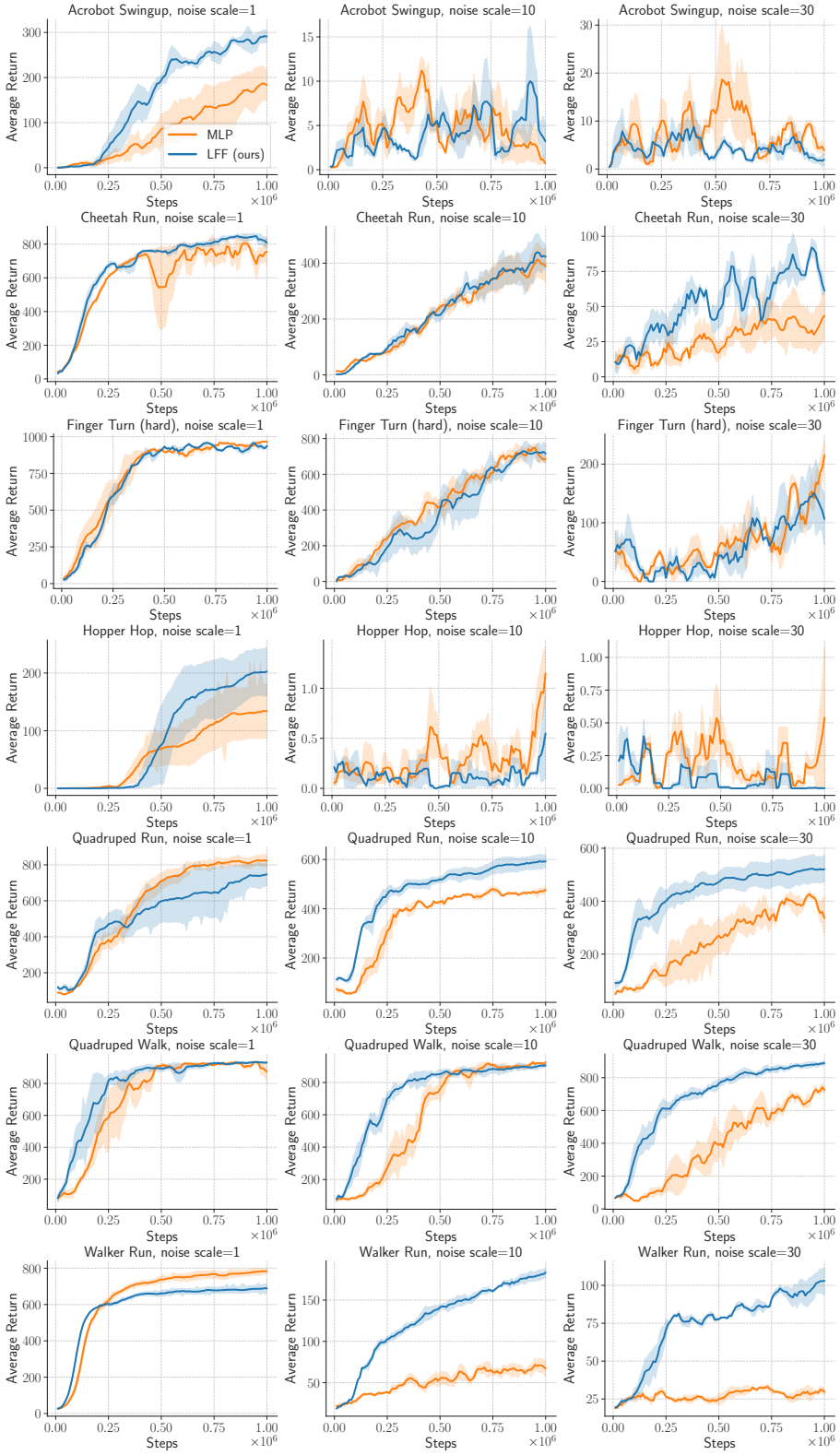
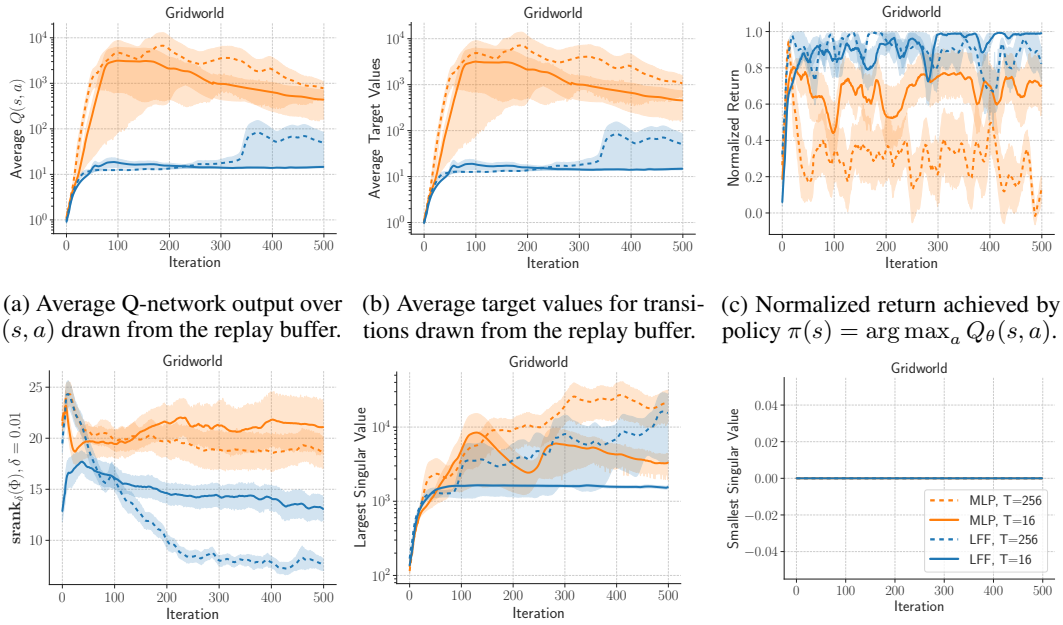


Figure 12: **State-based with Added Noise**: We add zero-mean Gaussian noise to the targets. As the standard deviation of the added noise increases, LFF maintains its performance better than MLPs.

D Noise Amplification vs Implicit Underparameterization in Gridworld MDP



(a) Average Q-network output over (s, a) drawn from the replay buffer.

(b) Average target values for transitions drawn from the replay buffer.

(c) Normalized return achieved by policy $\pi(s) = \arg\max_a Q_\theta(s, a)$.

(d) Effective rank of features over minibatch of states.

(e) Largest singular value of learned features.

(f) Smallest singular value of learned features.

Figure 13: Measuring noise amplification vs implicit underparameterization on GRID16SMOOTHOBS.

We want to examine whether noise amplification is indeed a problem that off-policy deep RL methods suffer from. Furthermore, if it is happening, Kumar et al. [23] hypothesize that it is being caused by underfitting ("implicit underparameterization"), which would contradict our claim that LFF improves learning by regularizing the training dynamics. We test this in Figure 13 by performing Fitted Q-iteration (FQI) on the GRID16SMOOTHOBS environment from Kumar et al. [23]. GRID16SMOOTHOBS is a discrete environment with 256 states and 5 actions, so we can use Q-iteration to calculate the optimal Q^* and compare that with our learned Q_θ. T denotes the number of gradient steps per FQI iteration; increasing the number of gradient steps is empirically more likely to cause divergence for MLP Q-functions. Note that we use a log scale for the y-axis in (a,b,e).

Noise amplification **(a)** shows that MLP-based Q-functions steadily blow up to orders of magnitude above the true Q^*, whose average value is around 15 in this environment. In contrast, LFF-based Q-functions either converge stably to the correct magnitude, or resist increasing as much. **(b)** shows that the MLP target values are in a positive feedback loop with the Q-values. **(c)** shows that divergence coincides with a drop in the returns. Together, these results indicate that there can be a harmful feedback loop in the bootstrapping process, and that methods like LFF, which reduce fitting to noise, can help stabilize training.

Implicit Underparameterization We run Fitted Q-iteration and calculate the effective rank of the Q-network, which we parameterize using either a MLP or LFF network. We follow the procedure from Kumar et al. [23]: sample 2048 states from the replay buffer and calculate the singular values σ_i of the aggregated feature matrix Φ. The effective rank is then defined as $\mathbf{srank}_\delta(\Phi) = \min\left\{k : \frac{\sum_{i=1}^{k}\sigma_i(\Phi)}{\sum_{j=1}^{d}\sigma_j(\Phi)} \geq 1-\delta\right\}$. **(d)** shows that the MLP's effective rank does not actually drop over training. Furthermore, LFF is able to avoid diverging Q-values, even though it has signficantly lower srank than its MLP counterpart. Thus, noise amplification for MLPs in this setting is likely not related to any underfitting measured by the effective rank. **(e)** shows that the

largest singular value blows up for MLPs, but stabilizes when training LFF for a reasonable number of gradient steps. **(f)** shows that the minimum singular value stays at zero over the course of training. In the context of Kumar et al. [23], this implies that their penalty $\sigma_{max}(\Phi)^2 - \sigma_{min}(\Phi)^2$ is exactly equivalent to penalizing only the maximum singular value $\sigma_{max}(\Phi)^2$ when using gradient descent. This is because the gradient of the second term is zero when the smallest singular value is zero. Thus, Kumar et al. [23]'s penalty works by constraining the largest singular value and regularizing the magnitude of the feature matrix. Overall, these results support our hypothesis that noise amplification is a problem that is not caused by underfitting and can be ameliorated by regularization.

E Further Ablations

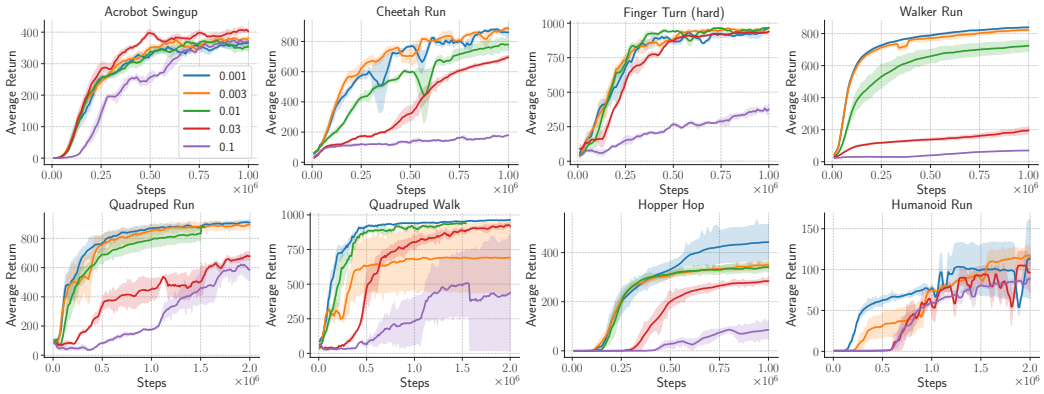


Figure 14: **Sensitivity to σ.** $\sigma = 0.001$ is a good default across all of these state-based environments. Results are averaged over 5 seeds, using a Fourier dimension of 1024, and the shaded region denotes 1 standard error.

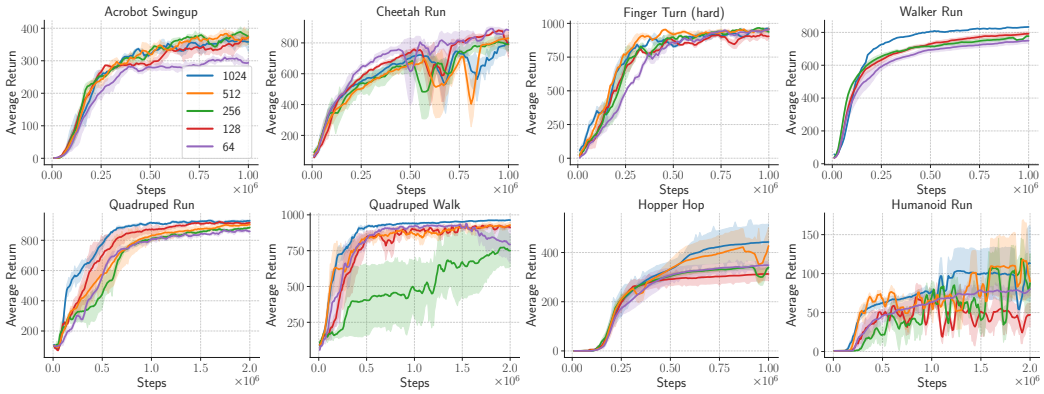


Figure 15: **Sensitivity to the Fourier dimension.** A Fourier dimension of 1024 is a good default across all of these state-based environments. Results are averaged over 5 seeds, using $\sigma = 0.001$, and the shaded region denotes 1 standard error.

F High Frequency Learning with Learned Fourier Features

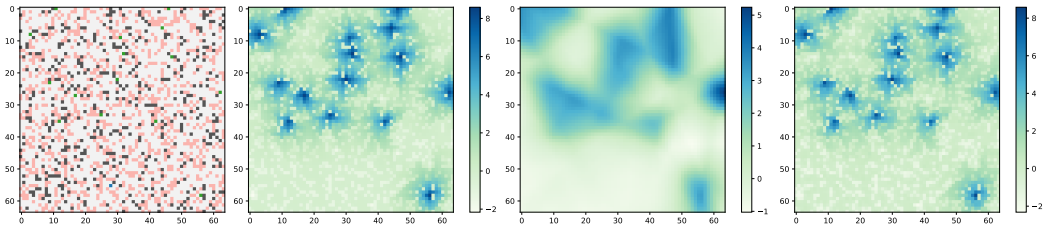


Figure 16: **Potential Underfitting in RL**. Left: gridworld structure. Red squares are lava, green are goals, gray are walls, and white squares are empty. Middle left: ground truth value function $V^*(s)$. Middle right: we fit $Q^*(s,a)$ with an MLP, then display $\max_a Q_\theta(s,a)$. The result is blurry and cannot properly distinguish between critical high and low value states. Right: we fit $Q^*(s,a)$ with our proposed learned Fourier feature architecture, then display $\max_a Q_\theta(s,a)$. It is able to exactly reproduce the ground truth, even with the same number of parameters and gradient steps.

While the main paper focused on using LFF with small σ to bias networks towards mainly learning low frequency functions, we can still use larger σ to encourage faster high-frequency learning. Prior theoretical work [4, 9, 33, 49] found that ReLU MLPs suffer from *spectral bias* – they can take an impractically long time to fit the highest frequencies. In the reinforcement learning setting, this can cause underfitting when fitting high frequencies is desirable.

F.1 Gridworld

We demonstrate that underfitting is indeed happening in RL. We create a toy 64×64 gridworld task [11] in Figure 16, where each square can be one of five types: start state, goal state, empty, wall, or lava. The agent starts at the start state and receives +1 reward for every timestep at a goal state, -1 penalty for every timestep at a lava square, and 0 reward otherwise. It cannot enter cells with a wall. The agent has five actions available: up, down, left, right, or no-op. Whenever the agent takes a step, there is a 20% chance of a random action being taken instead, so it is important for the agent to stay far from lava, lest it accidentally fall in. We use a discount of $\gamma = 0.9$. To create the environment, we randomly initialize it with 25% lava squares and 10% wall squares. We learn the optimal Q^* using Q-iteration, then attempt to fit various neural network architectures with parameters θ to the ground truth Q^* values through supervised learning:

$$\theta^* = \arg\min_\theta \sum_{s \in \mathcal{S}, a \in \mathcal{A}} (Q_\theta(s,a) - Q^*(s,a))^2 \tag{47}$$

We then try to fit Q^* using a standard MLP with 3 layers and 256 hidden units, and using our proposed architecture with an equivalent number of parameters and $\sigma = 3$. Due to the challenge of visualizing $Q(s,a)$ with 5 actions, we instead show $V(s) = \max_a Q(s,a)$ in Figure 16.

Surprisingly, MLPs have extreme difficulty fitting $Q^*(s,a)$, even when doing supervised learning on this low-dimensional toy example. Even without the challenges of nonstationarity, bootstrapping, and exploration, the deep neural network has trouble learning the optimal Q-function. The learned Q-function blurs together nearby states, making it impossible for the agent to successfully navigate narrow corridors of lava. Prior work has described this problem as state aliasing [27], where an agent conflates separate states in its representation space. We anticipate that this problem is worse in higher dimensional and continuous MDPs and is pervasive throughout reinforcement learning.

In contrast, our LFF embedding with $\sigma = 3$ is able to perfectly learn the ground-truth Q-function. This indicates that LFF with large σ can help our Q-networks and policies fit key high-frequency details in certain RL settings. We believe that there are two promising applications for high-frequency learning with LFF: model-based RL and tabular problems. Model-based RL requires modeling the dynamics, which can have sharp changes, such as at contact points. Modeling transition dynamics is also supervised, so there are no bootstrap noise problems exacerbated by accelerating the rate at which high-frequencies are learned. Tabular problems are also suited for high-frequency learning, as they often have sharp changes in dynamics or rewards (e.g. gridworld squares with walls, cliffs, or

lava). Capturing high-frequencies with LFF could improve both the sample-efficiency and asymptotic performance in this setting.

G Fourier Basis Variance After Training

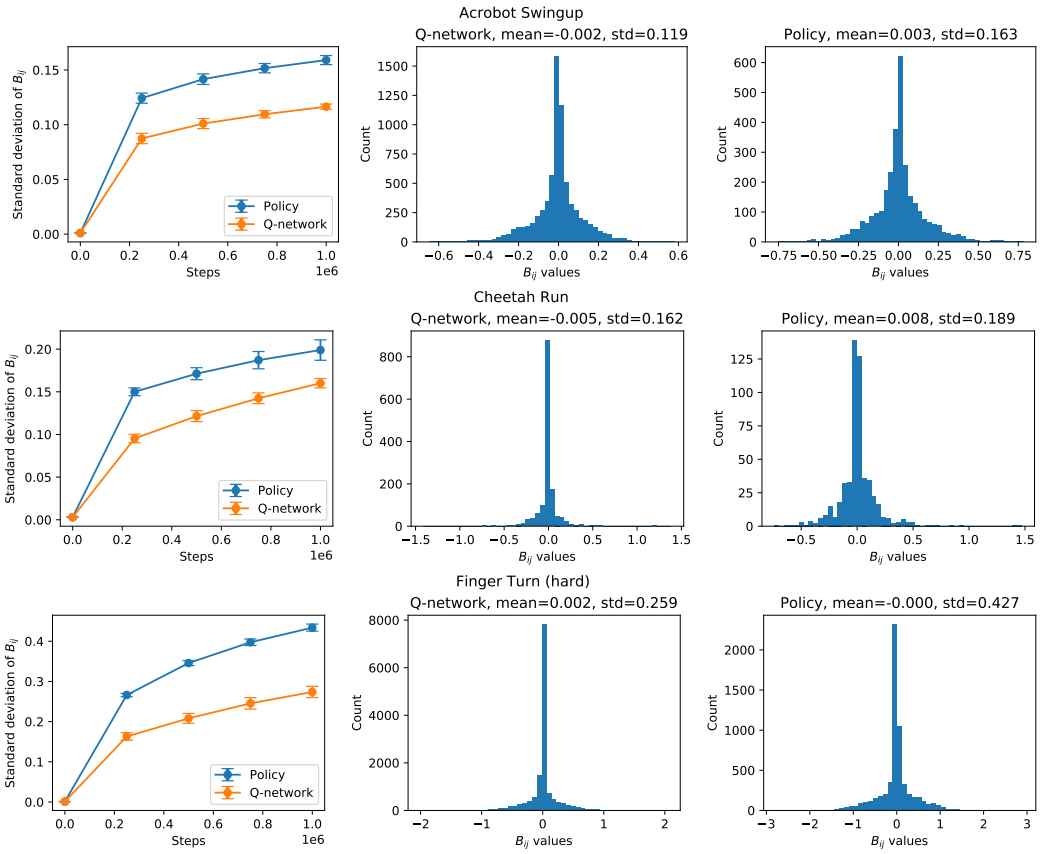


Figure 17: **Change in the standard deviation of Fourier basis entries** B_{ij}. The curves on the left show how the standard deviation changes from σ at initialization as state-based SAC training progresses. The confidence interval indicates the standard deviation of the standard deviation, measured across 5 seeds. The histograms in the middle and left show the distribution of entries within B for the policy and Q-networks at the end of training.

In Section 4, we used the Neural Tangent Kernel (NTK) perspective to show that the initialization variance of the Fourier basis B controls the per-frequency learning rate in infinite-width neural networks. However, the NTK's infinite-width assumption implies that the entries of the Fourier basis do not change over the course of training. Since we train B with finite width, we examine how its variance evolves over training, which affects its per-frequency learning rate at each point in time. Figure 17 and 18 show how the standard deviation of the Fourier basis change for the policy and Q-network. The standard deviation generally increases from $\sigma = 0.001$ or $\sigma = 0.003$ to about 0.1, which should still be more biased towards low frequencies than vanilla MLPs are (see Figure 4). Furthermore, the increase in standard deviation could actually be desirable. Having more data at the end of training could reduce the impact of bootstrap noise, so there may be less need for smoothing with small σ. Larger σ could bias the network towards fitting medium-frequency signals that are important for achieving full asymptotic efficiency. This could explain the image-based results in Figure 19, where the standard deviation rises to about 0.5.

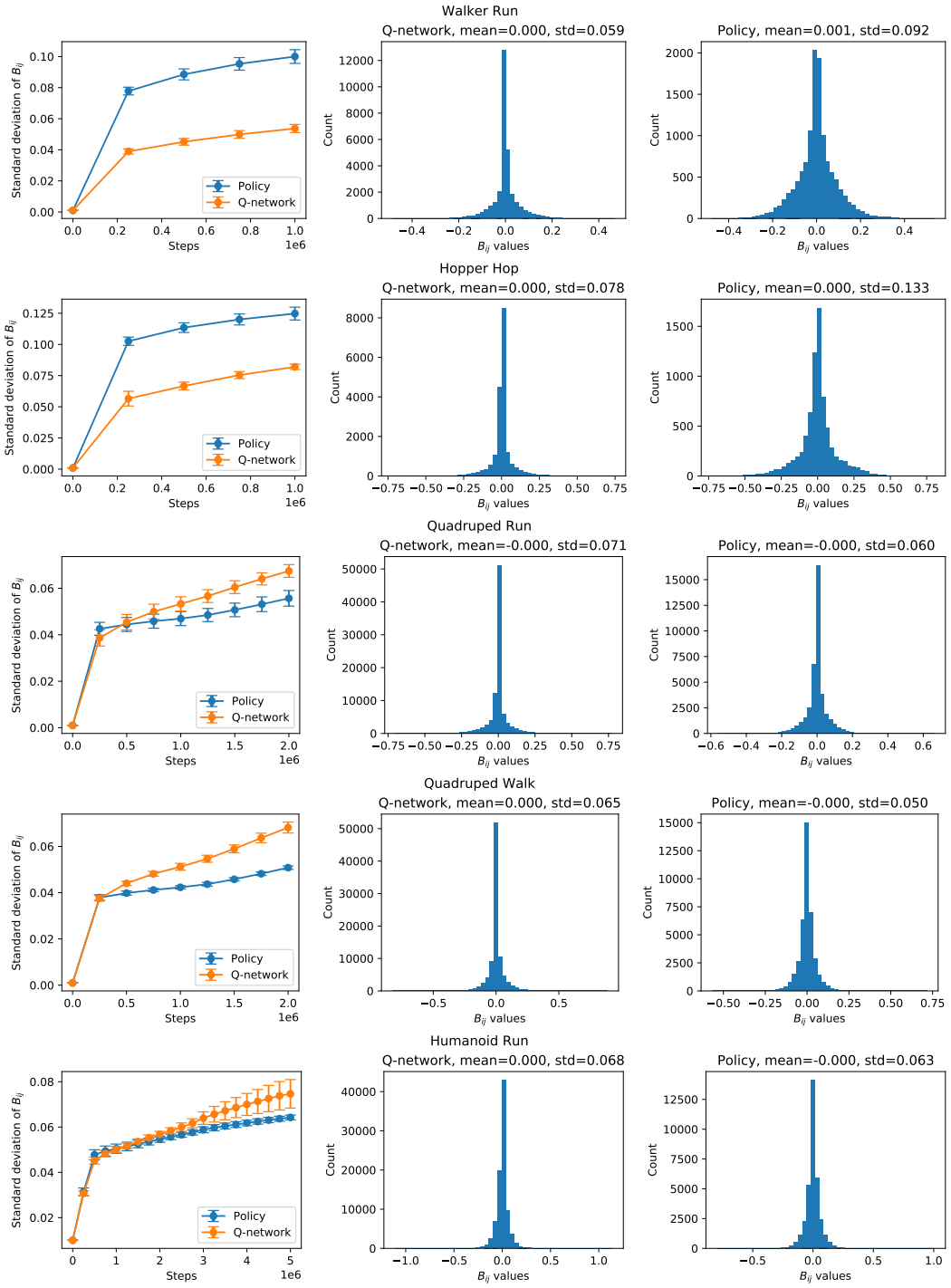


Figure 18: Continuation of Figure 17, which shows how the standard deviation of the Fourier basis changes over training.

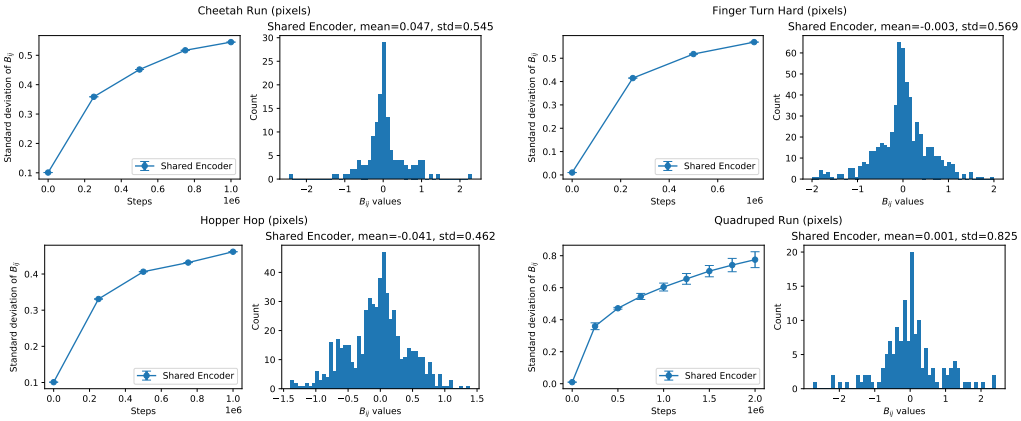


Figure 19: **Change in the standard deviation of Fourier basis entries B_{ij} for image-based experiments**. For each of the four environments, we show how the standard deviation of the shared encoder's Fourier basis changes over training. The histogram shows the distribution of the entries of the shared encoder's B at the end of training.

H Hyperparameters

We list hyperparameters for state-based and image-based SAC experiments in Table 1 and Table 2, respectively. We also show hyperparameters for PPO experiments from Appendix B in Table 3.

Parameter	Value
Algorithm	Soft Actor Critic [14]
Starting codebase	Yarats and Kostrikov [51]
Optimizer	Adam [19]
Adam (β_1, β_2)	(0.9, 0.999)
Discount	0.99
Batch size	1024
Target smoothing coefficient (τ)	0.005
Reward scale	Auto-tuned [15]
Actor learning rate	10^{-4}
Critic learning rate	10^{-4}
Reward scale learning rate	10^{-4}
Number of exploratory warmup steps	5000
Number of hidden layers	3 for MLP, 2 for LFF
Hidden size	1024
Hidden nonlinearity	ReLU
Fourier dimension	64 for Cheetah, 1024 otherwise
Standard deviation σ of Fourier basis initialization	0.003 for Cheetah, 0.001 otherwise

Table 1: Hyperparameters used for the state-basd SAC experiments.

Parameter	Value
Algorithm	Soft Actor Critic [14] + RAD [24]
Starting codebase	Laskin et al. [24]
Augmentation	Translate: Cheetah. Crop: otherwise.
Observation rendering	$(100, 100)$
Observation down/upsampling	Crop: $(84, 84)$. Translate: $(108, 108)$.
Replay buffer size	100000
Initial steps	1000
Stacked frames	3
Action repeat	4
Optimizer	Adam [19]
Adam (β_1, β_2)	$(0.5, 0.999)$ for entropy, (0.9, 0.999) otherwise
Learning rate	10^{-4}
Batch size	512
Encoder smoothing coefficient (τ)	0.05
Q-network smoothing coefficient (τ)	0.01
Critic target update freq	2
Convolutional layers (excluding LFF embedding)	4 for LFF, 5 otherwise
Discount	0.99
Fourier dimension	128 for Hopper, 64 otherwise
Initial standard deviation σ of Fourier basis	0.1 for Hopper, Cheetah; 0.01 for Finger, Quadruped

Table 2: Hyperparameters used for the image-based SAC experiments.

Parameter	Value
Algorithm	Proximal Policy Optimization [36]
Learning rate	3×10^{-4}
Learning rate decay	linear
Entropy coefficient	0
Value loss coefficient	0.5
Clip parameter	0.2
Environment steps per optimization loop	2048
PPO epochs per optimization loop	10
Batch size	64
GAE λ	0.95
Discount	0.99
Total timesteps	10^6
Number of hidden layers	2
Hidden size	256
Hidden nonlinearity	$\tanh$
Fourier dimension	64
Variance of Fourier basis initialization τ	0.01

Table 3: Hyperparameters used for the PPO experiments.