Functional Regularization for Reinforcement Learning via Learned Fourier Features

Anonymous Author(s)
Affiliation
Address
email

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 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 cumulative 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 [27, 29]. As a result, these RL algorithms may suffer from lower asymptotic performance or worse 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 [6, 30], or reducing the reliance on the bootstrapped Q-values by calculating multi-step returns with TD(λ) [24]. 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 [23] 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 [32], 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 [28] and L_2 regularization [7, 11] can help in

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

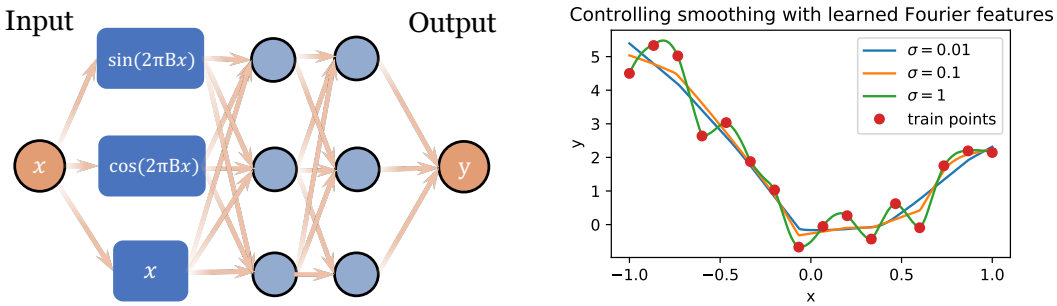


Figure 1: **Left:** the proposed learned Fourier feature (LF2) architecture. B is a matrix trained through backpropagation, and is used to create a learned set of Fourier features. The network then passes the Fourier features through alternating linear layers and ReLU nonlinearities, as is done in vanilla 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 $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$.

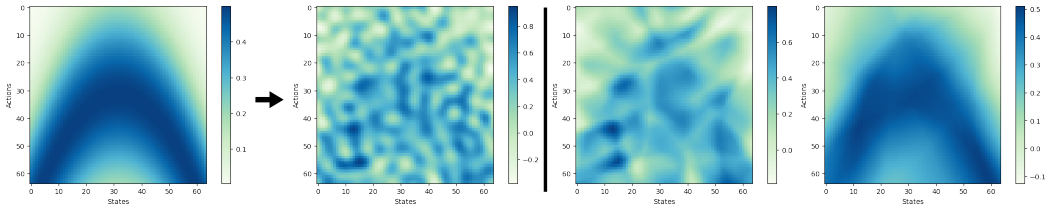
RL [14], but regularizing the network in weight space does not disentangle noise from reward signal and could make it difficult to learn the true Q-function. This leads us to ask: what is the right way to regularize the RL bootstrapping process?

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 LF2. In contrast to using fixed Fourier features [19, 25], 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 [8] 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, LF2'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, LF2 filters out bootstrapping noise while learning the underlying Q-function.

We evaluate LF2, which only requires changing a few lines of code, on state-space and image-space DeepMind Control Suite environments [26]. We find that LF2 produces moderate gains in sample efficiency on state-based RL and dramatic gains on image-based RL. In addition, we empirically demonstrate that LF2 makes the value function bootstrapping stable even in absence of target networks, and confirm that most of LF2'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') [2]. 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 [4, 6, 18, 24, 30]. 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 LF2 architecture to prevent Q-networks from fitting irreducible, high-frequency noise in the target values during these updates.



True Q-function | Noisy targets | MLP | LF2, σ = 0.003

Figure 2: **Filtering noise with LF2 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, LF2 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 [31]. 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 LF2 architecture controls how quickly low- and high-frequency signals are learned. Tuned properly, LF2 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 [15, 25]. They fixed this blurring problem by transforming the input using a random Fourier feature embedding of the input [19, 25]. 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. [25] 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 [9, 10]. 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. [25], 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 LF2 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 [19, 25]: 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 B 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 [21]. We further analyze these improvements in Section 6.4.

Algorithm 1 LF2 PyTorch-like pseudocode.

```
class LF2():
    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 (who are controlled at initialization by the variance σ^2), LF2 biases the network towards fitting a desired range of frequencies. In this section, we hope to understand why the LF2 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 [8]. This approximation allows us to understand the training dynamics of the learned neural network output function. 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:

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

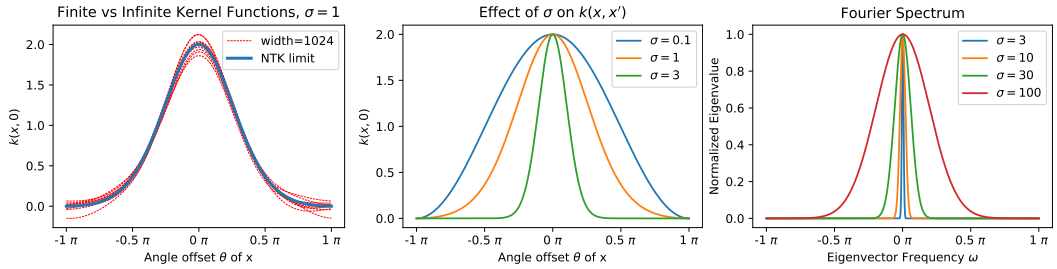
The Neural Tangent Kernel [8] line of analysis makes two further assumptions: f_θ is an infinitely wide neural network, and it is trained via gradient flow. Under the first condition, a trained network stays very close to its initialization, so the Taylor approximation is good (the so-called "lazy training" regime [3]). 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:

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

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 D.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.



(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.

4.2 Eigenvalues of the LF2 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 D.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 [1]. This decay rate is rather slow, so MLPs often fit undesirable medium and high frequency signals. **We hypothesize that LF2 controls the frequency-dependent learning rate by tuning the kernel matrix K's eigenvalues.** We examine LF2'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 LF2 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 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 LF2 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 D.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 [25]. 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. [25], 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 [17], a library that can compute the kernel function of any architecture expressible in its API .

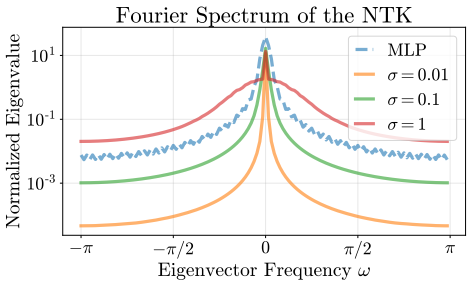


Figure 4: We compare the eigenvalue spectrum (which determines the frequency-specific learning rate) of deep networks with Fourier features to those of a vanilla MLP.

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. However, these results now reflect the empirical results where $\sigma \in [1, 10]$ results in high frequency learning. This indicates that deeper networks are crucial for understanding the behavior of Fourier features.

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 [26]. We will release the code, **which involves only changing a few lines of code in existing RL algorithms**.

State-based LF2 Architecture Setup We use soft actor-critic (SAC), an entropy-regularized off-policy RL algorithm [5], to learn 8 environments from the DeepMind Control Suite [26]. We keep the default hyperparameters fixed, varying only the architecture for the policy and Q-function. Our LF2 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 LF2's first layer due to input concatenation.

Image-based LF2 Architecture Setup We test image-based learning on 4 DeepMind Control Suite environments [26] with SAC + RAD [12], 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. [22], is denoted as "CNN" in Figure 6. To apply LF2 to images,

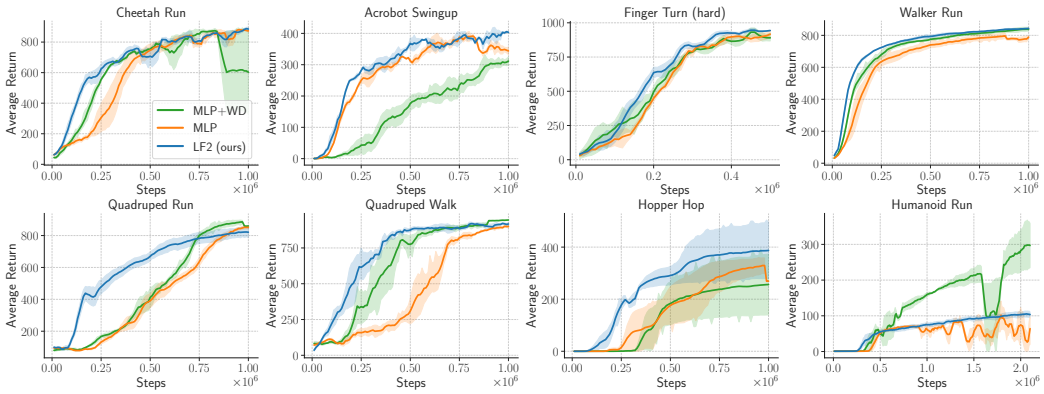


Figure 5: **Off-policy State-based Evaluation**: Soft Actor Critic (SAC) experiments on 8 DM Control environments. We emphasize that these results are produced using the same hyperparameters (e.g. learning rate, Polyak averaging parameter, and batch size) tuned for MLPs. These results show that plugging in our LF2 architecture can yield more sample-efficient learning on most environments.

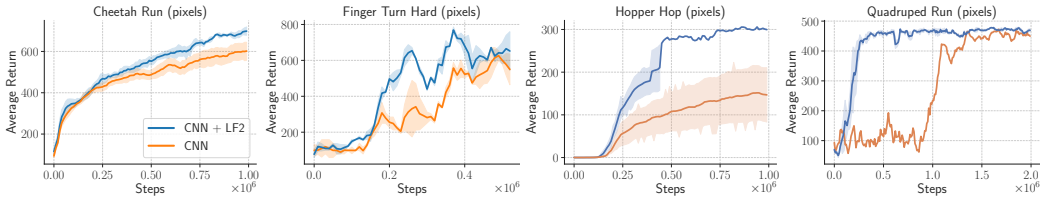


Figure 6: **Off-policy Image-based Evaluation**: SAC experiments on learning 4 DMControl environments from pixels. LF2 can yield dramatic improvements in sample-efficiency over CNNs.

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+LF2" architectures have the same parameter count, and that performance gains are solely due to LF2.

6 Results

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

1. Does LF2 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 LF2 help more when applied to the policy or the Q-function?
4. Ablation: How important is input concatenation or training the Fourier basis B?

Learning curves are averaged over 5 seeds, with the shaded region denoting a 68% confidence interval.

6.1 LF2 Architecture for Off-policy RL

We show the results of using the LF2 architecture for state-based RL with SAC in Figure 5. LF2 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 LF2 architecture. We hypothesize that LF2 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.

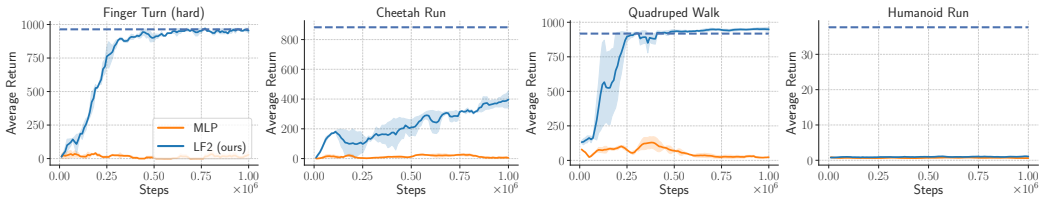


Figure 7: **Effect of LF2 on Stability of Bootstrapping**: We train SAC, foregoing a target network, by bootstrapping directly from the Q-function being trained. The dashed line shows the LF2 performance with a target network after 1M steps. We find that the LF2 network is remarkably stable, and even learns faster on Quadruped Walk than when using target networks. However, LF2 fails to learn on Humanoid, indicating that higher dimensional problems still pose problems.

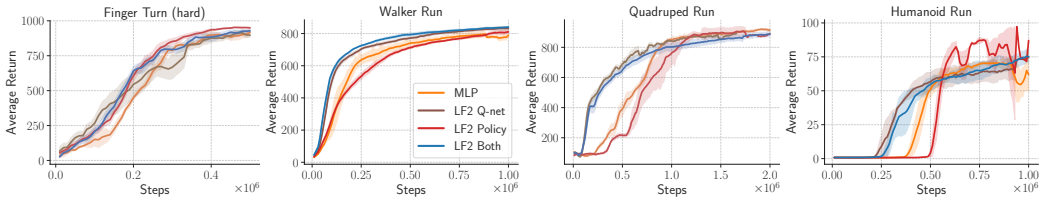


Figure 8: **LF2 Policy vs Q-function.** Walker and Quadruped results indicate that only using LF2 for the Q-network is just as good as using LF2 for both networks. In contrast, using LF2 for the policy network is about as bad as the MLP baseline. This suggests that LF2 primarily improves off-policy learning by reducing noise in the Q-network optimization.

For the state-space experiments, we also test weight decay with MLP networks, tuned over the range $\{10^{-3}, 3 \times 10^{-4}, 10^{-4}, 3 \times 10^{-4}, 10^{-5}\}$. Weight decay also helps learning in most environments, but it can hurt performance (Acrobot, Hopper) or introduce instability (Cheetah). We suggest that weight decay strong enough to reduce overfitting will simultaneously bias the Q-values towards 0 and cause underestimation bias. This is a shortcoming of weight-space regularization that LF2 avoids.

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. To further test whether LF2 solves this problem by filtering out the noise, we "crank the noise up to 11." We train a SAC agent on state-based DMControl environments, but no longer use target networks to calculate target values in the Bellman updates. Target networks, updated infrequently, slow down the propagation of noise due to bootstrapping [16]. LF2 fits less noise to begin with, so it should work even when the target network is omitted.

Figure 7 shows that SAC with MLP networks consistently fails to learn on all environments in this setting, while the LF2 architecture still performs well, except when the problem is very high dimensional. LF2 manages to even learn faster in Quadruped Walk than it does when using a target network, since there is no longer Polyak averaging [13] with a target to slow down information propagation. Omitting the target network allows us to use fresh, 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 unintended changes to the Q-network outputs [20]. Overall, Figure 7 validates our theoretical claims that LF2 controls the effect of high-frequency noise on the learned function, and indicates that LF2 successfully mitigates bootstrap noise in most cases. Tuning the SAC hyperparameters should increase LF2 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 LF2 architecture improves RL performance by primarily preventing the Q-network from fitting noise. We train state-based SAC with an MLP policy and LF2 Q-network (LF2 Q-net in Figure 8), or with an LF2 policy and MLP Q-network (LF2 Policy). Fig-

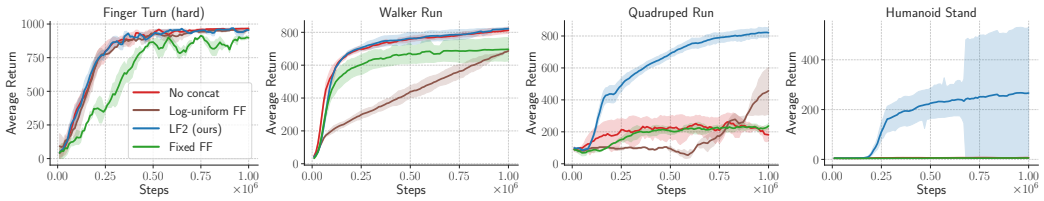


Figure 9: **Ablation analysis**: We train SAC on 4 DMControl environments with three variants of our architecture: LF2, LF2 with fixed Fourier features, and LF2 without input concatenation. Lower dimensional environments like Finger and Walker are more forgiving for fixed Fourier features or omitting input concatenation. However, Humanoid absolutely requires both modifications to learn.

ure 8 shows that solely regularizing the Q-network is sufficient for LF2'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 LF2 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 to noise, while the policy can be fine-grained and change quickly between nearby states. However, for simplicity, we use LF2 for the Q-networks and the policy networks.

6.4 Architectural Ablations

LF2 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 9 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 LF2 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 [15], but we additionally concatenate x and train B. We find that this initialization method, dubbed "Log-uniform FF" in Figure 9, 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 LF2 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 benchmark environments. One shortcoming of frequency-based functional 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 LF2. We also plan to examine whether LF2 helps RL work in new environments where current methods fail. We hope this work will provide new perspectives on existing RL algorithms for the community to build upon.

References

[1] 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. 5

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

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

[4] 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

[5] 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. 6

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

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

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

[9] 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

[10] 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

[11] 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

[12] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020. 6

[13] 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. 8

[14] 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

[15] 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, 9

[16] 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. 8

[17] 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

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

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

[20] 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. 8

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

[22] A. Srinivas, M. Laskin, and P. Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020. 6

[23] 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

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

[25] 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

[26] 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, 6

[27] 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

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

[29] 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

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

[31] 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

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

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section **??**.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

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 D
   (b) Did you include complete proofs of all theoretical results? [Yes] See Appendix D
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 will be 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 A.
   (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]