Improving PILCO with a Neural Network Gaussian Process

Word Count: 5586

March 29, 2025

I. Introduction & Related Work

Machine learning can be defined as the problem of teaching computers how to learn from data. Reinforcement learning (RL) is a subfield of machine learning that was developed to mimic the process of reinforcement learning in humans. In a very abstract RL problem, an agent/controller learns how to make decisions, called actions, under a policy which is some framework that depicts possible actions. Over time, the controller continuously acts in an environment, which is essentially some model of the problem. After the controller chooses an action, the environment provides a numerical cost that quantifies the quality of the action made. The goal of the agent is to minimize this cost. Although RL can be applied to a multitude of problems, this paper's discussion will be limited to learning how to control robots in a physical system.

Model-based RL (MBRL) refers to a proper subset of RL algorithms in which a model of the environment is learned to aid in the solving of the problem. MBRL algorithms use this learned environment model and run trials on different actions in a step called planning to guess how the environment will respond to certain actions to gain simulated experience. One of these MBRL algorithms is Probabilistic Inference for Learning Control (PILCO) [Deisenroth et al., 2015]. PILCO utilizes Gaussian Process (GP) inference to develop a probabilistic dynamical model of its given problem

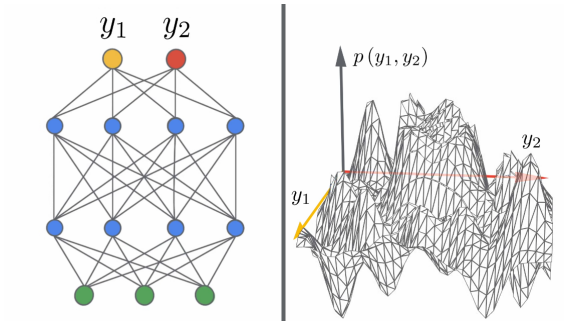

Figure 1: *A Simple Neural Network*

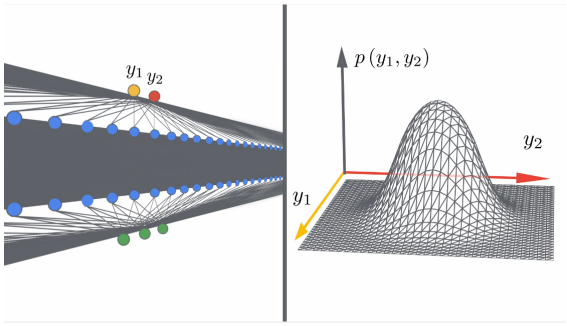

Figure 2: *A Neural Network of Infinite Width converges to a Gaussian Process*

and was able to solve non completely trivial RL problems from scratch in a comparatively short time frame. Although GPs are very data efficient, the main contributor behind PILCO's ability to learn from scratch, GPs are not the best at learning non-smooth dynamics models. Additionally, the running time of PILCO scales cubically with respect to trials [Levine, 2017] and thus, does not scale up very well to high dimensional problems.

To address these issues, [Gal et al., 2016] introduced DeepPILCO. DeepPILCO replaced PILCO's GP with a Bayesian Neural Network (BNN) to mimic PILCO's probabilistic nature. Neural networks (NNs) do not suffer from the aforementioned issues with GP's as they are able to scale up to problems of higher dimension easily. However, training a NN requires quite a bit of data which diminishes PILCO's crucial ability of data efficiency.

Although additional methods like MC-PILCO [Amadio et al., 2022] have been created to address some of the aforementioned issues, MC-PILCO still relies on a GP and may suffer from the same issues mentioned with PILCO.

It has been known since [Neal, 1996] that in the infinite limit, in some cases Neural Networks (NNs) results in GP-like behavior. [Lee et al., 2017] derived an exact equivalence for this relationship in the form of Neural Network Gaussian Processes (NNGPs) and the Neural Tangent Kernel (NTK). These equivalence relations allow exact Bayesian inference[1] on BNNs of infinite width. This equivalence suggests that there may be improvements to the PILCO algorithm if one replaces its GP with a NNGP. This paper serves to test this hypothesis on the cart-pole swing-up problem: a standard RL benchmark problem that was used to test PILCO and its many variants. To be more specific, Does replacing the GP in the PILCO algorithm with a NNGP provide gains in performance in the cart-pole swing-up problem? There is a substantial lack of research regarding the usage of NNGPs in RL. An extensive search yielded one paper in which a NNGP was used to successfully solve a RL problem [Goumiri et al., 2020]. The paper was completely unrelated to the usage of PILCO and their results showed great promise in the usage of NNGPs in RL which provides more motivation for a possible improvement in performance.

[1]This is formal name for the inference part of PILCO

II. Model

i. A Quick Overview of the PILCO Algorithm

The PILCO algorithm ([Deisenroth et al., 2015] and [Deisenroth and Rasmussen, 2011]) learns how to control dynamical systems of the form

$$\boldsymbol{x_{t+1}} = f(\boldsymbol{x_t}, \boldsymbol{u_t} + \epsilon) \tag{1}$$

Where x_t is a vector that describes the state of the system at time t, u_t is the vector that describes the control signal applied at time t, f is an unknown function that describes the transition dynamics of the physical system, and ϵ is some noise sampled from a gaussian distribution $\mathcal{N}(\mathbf{0}, \sigma_\epsilon)$ which is defined as

$$\mathcal{N}(\mu, \Sigma) = \frac{\exp(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^T(\mathbf{x}-\boldsymbol{\mu}))}{\sqrt{(2\pi)^k \det(\boldsymbol{\Sigma})}} \tag{2}$$

A gaussian (normal) distribution is completely defined by its mean μ and covariance Σ which is inherently a measure of how "spread out" the distribution is and quantifies the uncertainty of our distribution. The goal of PILCO is to find the controller, a deterministic policy which is a function of the current state and θ, $\pi(x_t, \theta) = u_t$, that minimizes the sum over the average (expected) value of the cost function $c(x_t)$ over time T under the policy π

$$J^\pi(\theta) = \sum_{t=0}^{T} \mathbb{E}[c(x_t)]. \tag{3}$$

The possible policies are parameterized by a vector θ which is initially sampled from $\mathcal{N}(0, I)$.

PILCO uses GP regression, a method of kernel regression, to learn the function f. Kernel regression is a nonlinear regression method. The essential idea of kernel regression is sending our data with a nonlinear mapping into something called a reproducing kernel Hilbert space (RKHS). In this RKHS, the data will have

a linear-esque relationship and it is then possible to perform linear regression on this transformed data. The kernel is a mathematical function that serves as a way to measure distance between two points in this transformed data space.

A GP is defined by its mean and covariance functions. PILCO assumes a mean function of zero and uses a Squared Exponential (SE) covariance function (our kernel) that is defined as

$$k(\tilde{x}, \tilde{x}') = \alpha^2 e^{-\frac{1}{2}(\tilde{x}-\tilde{x}')^T \Lambda^{-1}(\tilde{x}-\tilde{x}')} \tag{4}$$

A GP can be thought of as a probability distribution over the possible functions that can approximate given data which will be generated by applying the initial policy in PILCO. PILCO's GP uses inputs (x_{t-1}, u_{t-1}) and outputs $\Delta t = x_t - x_{t-1} + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma_\epsilon)$ to predict the latent function f which in turn is used to predict the difference between states Δt given the control signal u_t and the previous state x_t. This probability distribution is assumed to be Gaussian and gives a one-step prediction of the environment. The values $\alpha^2, \Lambda, \sigma_\epsilon$ are respectively the hyperparameters variance, lengthscale, and noise variances. These hyperparameters were learned via Expectation Maximization in the original PILCO algorithm.

To minimize the cost, PILCO needs to evaluate the policy over time T rather than from just the next step of the environment. To do this, PILCO cascades these one-step predictions, which require calculating the distribution for the next state distribution. This predictive distribution $p(\Delta t)$ can be represented as an integral however, the integral is not analytically tractable. Therefore, the predictive distribution was assumed to be Gaussian distributed, meaning the distribution can be represented as a Gaussian with mean μ and covariance Σ, and the distribution was then approximated via a technique called Gaussian moment matching. Given the predictive mean μ_Δ and predictive covariance Σ_Δ (derived in [Deisenroth and Rasmussen, 2011]), $p(x_{t+1})$ is given by

$$\mathcal{N}(x_{t+1}|\mu_{t+1}, \Sigma_{t+1}) \tag{5}$$

$$\mu_{t+1} = \mu_t + \mu_\Delta \tag{6}$$

$$\Sigma_{t+1} = \Sigma_t + \Sigma_\Delta + \text{cov}[x_t, \Delta_t] + \text{cov}[\Delta_t, x_t] \tag{7}$$

The cost is then follows by the definition of the expected value $\mathbb{E}[X] = \int_\infty x f(x) dx$ given by

$$\mathbb{E}[c(x_t)] = \int c(x_t) \mathcal{N}(x_t|\mu_t, \Sigma_t) dt \tag{8}$$

where we choose a cost function $c(x_t)$ such that this integral is tractable at $t = 1, 2, \ldots, T$ so we can solve 3. In this context, the integral $\int$ is splitting the new distribution $\mathcal{N}(x_t|\mu_t, \Sigma_t)$ into an infinite number of states x_{ti}. All of these slices are then being multiplied by $c(x_{ti})$ and the probability that the state $c(x_{ti})$ occurs and then summed up. This will give us the expected cost of being in that distribution. Repeating and then summing over T, we will be able to calculate the long term cost of following the policy π.

To update the parameters θ, one can simply calculate the gradients/derivatives of J, μ, and Σ with respect to θ. The derivative of a function is another function that gives us the rate of change of the original function with respect to a certain variable. Intuitively, this new function "points" in the direction of the minimum/maximum of the original function. With analytic gradients, it is possible to perform any standard gradient-based optimization method such as limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) as in the original implementation of PILCO. These steps of learning the GP model, calculating the cost, and applying the gradient-based policy improvement method are then repeated until convergence to a good policy.

ii. PILCO Modifications: Neural Networks as Gaussian Processes

Five years ago, a paper by [Lee et al., 2017] introduced a correspondence between Neural

Networks and Gaussian Processes. Essentially speaking, a simple Neural Network f can be defined as

$$f(x) = A\phi(Bx) \tag{9}$$

where the matrices A, B are the parameters of the Neural Network and ϕ is an element-wise activation function. The elemenet-wise activation function essentially models neurons: it decides whether or not a certain element of the matrix is either useful or not. The goal of the Neural Network is to learn a relation between the input data x and the outputs y by minimizing the loss

$$L(A, B) = \frac{1}{2}\sum_{i=0}^{n}(y^{(i)} - f(x^{(i)}) \tag{10}$$

given by the sum of the difference of the $i = 1, 2, \ldots, n$ training samples $y^{(i)}$ and outputs of the Neural Network $f(x^{(i)})$.

To train the Neural Network, one can perform gradient descent which can be defined by the equations

$$\begin{aligned} A^{t+1} &= A^{(t)} - \alpha\frac{\partial L}{\partial A} \\ \mathrm{B}^{t+1} &= B^{(t)} - \alpha\frac{\partial L}{\partial B} \end{aligned} \tag{11}$$

The intuition behind training under gradient descent is adding small changes of the gradients to the parameters to move those parameters to their minimum values.

When the training of B in 11 is frozen, it turns out that training a Neural Network under gradient descent is equivalent to kernel regression: $\phi(Bx)$ acts as the map, and we are simply performing linear regression.

Furthermore, when B is sampled from $N(0, I)$ and the dimensions of A and B tend towards to infinity, our Neural Network converges to a Gaussian Process. When this infinite limit has a closed form, it is known as a Neural Network Gaussian Process (NNGP) [Radhakrishnan, 2022]. This NNGP has a kernel that depends on the architecture of the neural network and it follows that all the equations in PILCO are the exact same for NNGPs where all computations regarding the GP kernel are replaced with the new NNGP kernel.

III. Method

i. Cart-Pole Swing-Up

To test if replacing the GP in PILCO with an NNGP will provide gains in perofrmance, we need to test it on a RL problem. We test the algorithms PILCO and NNGP-PILCO on their performance at learning a controller for the cart-pole swing-up problem. In the cart-pole swing-up problem (see **Figure 3**), a pendulum that hangs freely solely under the force of gravity is attached to a cart that is initially at rest. At every time step, a force can be applied to either the left or right of the cart. The goal for the controller is to learn the sequence of forces to upright the pendulum and center it on the track. Intuitively, the solution to this problem is is sinusoidal in nature: We would have to repeatedly apply a force to the left and right of the force in even amounts.

The state of the system is defined as

$$x = [x, \dot{x}, \theta, \dot{\theta}]. \tag{12}$$

The target is

$$x_{\text{target}} = [0, 0, \pi, 0] \tag{13}$$

and the control signal is a force $u \in [-10N, 10N]$. The cost function is defined as

$$c(x_t) = 1 - e^{-\frac{||x_{\text{target}} - x_t||^2}{\sigma_c^2}} \tag{14}$$

where σ_c^2 defines the width of the cost penalty.

The controller used was a nonlinear RBF network which is defined as

$$\begin{aligned} \pi(x, \theta) &= \sum_{i=0}^{n} w_i\phi_i(x) \\ \phi_i(x) &= e^{-\frac{1}{2}(x-\mu_i)^T\Lambda^{-1}(x-\mu_i)} \end{aligned} \tag{15}$$

where the controller parameters are given by $\theta = \{w_i, \Lambda, \mu_i\}$. A nonlinear RBF network

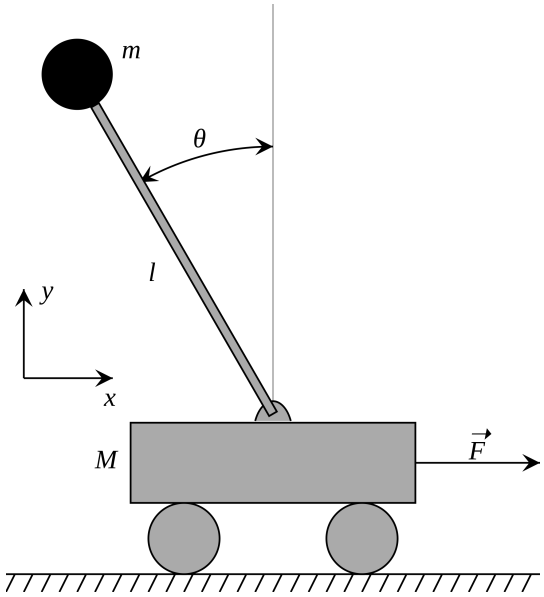

Figure 3: *Cart-Pole Swing-Up*

is functionally equivalent to a GP with zero variance.

To measure the effectiveness of the algorithms, we can measure the accumulated cost over time. By graphing the accumulated cost over time, we can quantify the performance of the algorithms [Poole and Mackworth, 2017]. Graphs that are consistently lowest over time will be of better performance as they are reducing more cost in a shorter period of time. We chose this method as it used to quantify the performance of DeepPILCO and MC-PILCO. Additionally, it is perfect given the definition of a RL problem: The goal of the agent is to minimize the cost.

To keep consistency, we use the same cart-pole environment, which was created using the differential equations of motion for the Cart-Pole Swing-Up problem mentioned in section IV of [Gal et al., 2016] and Euler's method.

ii. Implementing PILCO and NNGP-PILCO

PILCO was implemented in Python using Jax and GPJax and NNGP-PILCO would have been implemented in Neural Tangents. An issue arose though: the kernel in PILCO is defined as

Parameter	Value
T	25
n	50
Λ	**Learned via MLL**
α^2	1
ϵ	0
σ_c^2	0.25

Table 1: *Hyperparameter Settings*

$$k(\tilde{x}, \tilde{x}') = \alpha^2 e^{-\frac{1}{2}(\tilde{x}-\tilde{x'})^T \Lambda^{-1}(\tilde{x}-\tilde{x}')} \tag{16}$$

and the exact same kernel in Neural Tangents is defined as

$$\begin{aligned} k(x, x') &= e^{-\gamma||x-x'||^2} \\ \gamma &= \frac{1}{2\ell^2} \end{aligned} \tag{17}$$

Although these are essentially equivalent, the kernel in Neural Tangents does not account for the variance α^2 term. Therefore, we are forced to set α^2 to one.

IV. Experimental Results

An issue arose when implementing the PILCO algorithm. When running the PILCO algorithm with the parameters given in Table 1, there was unexpected behavior in the Gaussian Moment Matching approximations:

The true rollouts for the states for $t = 0, 1, 2$ (calculated via the differential equations) were given as

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ -0.693974 \\ 0 \\ -2.1513193 \end{bmatrix} \rightarrow \begin{bmatrix} -0.0693974 \\ -1.4428222 \\ -0.21513192 \\ -2.321429 \end{bmatrix}.$$

However, the rollouts given by PILCO were completely nonsensical. Starting from the same

state $x_0 = [0\ 0\ 0\ 0]$, when Λ was optimized for 100 iterations, the next state predicted was

$$\begin{bmatrix} -2268761.5671538827 \\ -22759325354304.797 \\ 5146873.879487108 \\ -58380499596828.28 \end{bmatrix}.$$

Under the suspicion of these results being due to the inaccuracy of the parameter Λ, the training time of Λ_x was varied where x will denote the number of iterations the hyperparameters were optimized. However, similar results were observed. All starting from the initial state x_0, for Λ_{10}, the state at $t = 1$ was predicted to be

$$\begin{bmatrix} -4472381.692014483 \\ 77024158285233.61 \\ 10135887.13921894 \\ -208090469534488.7 \end{bmatrix}.$$

For $\Lambda_{1000}, t = 1$ was predicted to be

$$\begin{bmatrix} 0.0 \\ 0.27953227024835475 \\ 0.0 \\ 0.8665500377698998 \end{bmatrix}.$$

For $\Lambda_{10000}, t = 1$ was predicted to be

$$\begin{bmatrix} -6349.973886719554 \\ -597128281.0319556 \\ 20480.363992728126 \\ 622629532.6654978 \end{bmatrix}$$

For $\Lambda_{50000}, t = 1$ was predicted to be

$$\begin{bmatrix} 81.52807585224842 \\ -501197928.971957 \\ -21691.286512553186 \\ -1115504489.6414776 \end{bmatrix}$$

For $t = 2$, the states were given as

$$\begin{bmatrix} -32133746267120.06 \\ 1.8241505722521188e + 24 \\ -1104014051687.7798 \\ -4.121815078211794e + 23 \end{bmatrix}$$

$$\begin{bmatrix} -19892448425295.797 \\ 5.010264248209529e + 23 \\ -683567062744.3505 \\ 1.1081037055224104e + 24 \end{bmatrix}$$

$$\begin{bmatrix} 0.0 \\ 0.27953227024835475 \\ 0.0 \\ 0.8665500377698998 \end{bmatrix}$$

$$\begin{bmatrix} 5142948.250507345 \\ -743316740545.9059 \\ 959484.2719658513 \\ 1957064455579.111 \end{bmatrix}$$

$$\begin{bmatrix} -646442364.843772 \\ 86651659452032.06 \\ -20333429.403938837 \\ 266062367021231.88 \end{bmatrix}$$

for Λ being trained for $10, 100, 1000, 10000$, and 50000 iterations respectively.

V. Discussion

i. Conclusion

After seeing essentially no change in the quality of the rollouts provided by the PILCO algorithm with $\alpha^2 = 1$ and varying the accuracy of the Λ parameter, we can conclude that the lengthscale parameter of a GP is not enough to encode information about both μ_Δ and Σ_Δ in the PILCO algorithm. These results, although odd, are sensible because α^2 is a scaling factor. Based off of this evidence, we can conjecture that replacing the GP in PILCO with a NNGP will not provide gains in performance as there is no way to account for the variance term in the dual RBF kernel of the NNGP.

ii. Limitations

There are several limitations to this conclusion. Foremost, there were some discrepancies in the implementation of the PILCO algorithm. The hyperparameters were trained via the Maximum Log Likelihood instead of via Evidence

Maximization. Although unlikely, this deviation may have resulted in the discrepancies in the rollouts generated by PILCO. Additionally, there may have been a missed error in the implementation of the PILCO algorithm. Lastly, we can not conclusively say that NNGP-PILCO will not provide gains in performance as the implementation was not tested. Although the conclusion from the issue with the variance term locally follows, it is not conclusive.

iii. Further Research

There exist several avenues for future research. First of all, the results of this experiment could be verified by implementing NNGP-PILCO with $\alpha^2 = 1$ and test to see if the algorithm is capable of solving the problem. Something that may work a bit better though would be re deriving the equations in PILCO for a different kernel with at least two parameters if such a thing is possible.

References

[Amadio et al., 2022] Amadio, F., Libera, A. D., Antonello, R., Nikovski, D., Carli, R., and Romeres, D. (2022). Model-based policy search using monte carlo gradient estimation with real systems application. *IEEE Transactions on Robotics*, 38(6):3879–3898.

[Deisenroth et al., 2015] Deisenroth, M. P., Fox, D., and Rasmussen, C. E. (2015). Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423.

[Deisenroth and Rasmussen, 2011] Deisenroth, M. P. and Rasmussen, C. E. (2011). Pilco: A model-based and data-efficient approach to policy search. In Getoor, L. and Scheffer, T., editors, *ICML*, pages 465–472. Omnipress.

[Gal et al., 2016] Gal, Y., McAllister, R., and Rasmussen, C. E. (2016). Improving PILCO with Bayesian neural network dynamics models. In *Data-Efficient Machine Learning workshop, International Conference on Machine Learning*.

[Goumiri et al., 2020] Goumiri, I. R., Priest, B. W., and Schneider, M. D. (2020). Reinforcement learning via gaussian processes with neural network dual kernels.

[Han et al., 2022] Han, I., Zandieh, A., Lee, J., Novak, R., Xiao, L., and Karbasi, A. (2022). Fast neural kernel embeddings for general activations. In *Advances in Neural Information Processing Systems*.

[Hron et al., 2020] Hron, J., Bahri, Y., Sohl-Dickstein, J., and Novak, R. (2020). Infinite attention: Nngp and ntk for deep attention networks. In *International Conference on Machine Learning*.

[Lee et al., 2017] Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J., and Sohl-

Dickstein, J. (2017). Deep neural networks as gaussian processes.

[Levine, 2017] Levine, S. (2017). CS294-112 Deep reinforcement learning: Lecture 12, Model-based reinforcement learning.

[Murphy, 2022a] Murphy, K. P. (2022a). *Probabilistic Machine Learning: An introduction*. MIT Press.

[Murphy, 2022b] Murphy, K. P. (2022b). *Probabilistic Machine Learning: An introduction*. MIT Press.

[Murphy, 2023] Murphy, K. P. (2023). *Probabilistic Machine Learning: Advanced Topics*. MIT Press.

[Neal, 1996] Neal, R. M. (1996). *Bayesian Learning for Neural Networks, Vol. 118 of Lecture Notes in Statistics*. Springer-Verlag.

[Novak et al., 2022] Novak, R., Sohl-Dickstein, J., and Schoenholz, S. S. (2022). Fast finite width neural tangent kernel. In *International Conference on Machine Learning*.

[Novak et al., 2020] Novak, R., Xiao, L., Hron, J., Lee, J., Alemi, A. A., Sohl-Dickstein, J., and Schoenholz, S. S. (2020). Neural tangents: Fast and easy infinite neural networks in python. In *International Conference on Learning Representations*.

[Piedrahita, 2017] Piedrahita, D. (2017). Cartpole control.

[Poole and Mackworth, 2017] Poole, D. and Mackworth, A. (2017). *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, Cambridge, UK, 2 edition.

[Radhakrishnan, 2022] Radhakrishnan, A. (2022). 6.s088 modern machine learning: Simple methods that work lecture notes.

[Raiko and Tornio, 2009] Raiko, T. and Tornio, M. (2009). Variational bayesian learning of nonlinear hidden state-space models for model predictive control. *Neurocomputing*, 72(16):3704–3712. Financial Engineering Computational and Ambient Intelligence (IWANN 2007).

[Schoenholz and Novak, 2020a] Schoenholz, S. S. and Novak, R. (2020a). Fast and easy infinitely wide networks with neural tangents.

[Schoenholz and Novak, 2020b] Schoenholz, S. S. and Novak, R. (2020b). Fast and easy infinitely wide networks with neural tangents.

[Sohl-Dickstein et al., 2020] Sohl-Dickstein, J., Novak, R., Schoenholz, S. S., and Lee, J. (2020). On the infinite width limit of neural networks with a standard parameterization.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.