Using Cumulative Distribution Based Performance Analysis to Benchmark Models

Scott M. Jordan **Daniel Cohen** **Philip S. Thomas**
University of Massachusetts Amherst
[sjordan,dcohen,pthomas]@cs.umass.edu

Abstract

When using only reported empirical results, it has become difficult to identify machine learning methods that provide meaningful advancement. One reason is that results are commonly only reported using well-tuned models, and thus represent an optimistic evaluation of performance. In this work, we propose a new framework for evaluating algorithms that presents both the performance when the system is well-tuned, as well as the difficulty of tuning the algorithm. This is achieved by considering the distribution of performances that result when applying the method with different hyper-parameter settings (e.g., different step sizes and network structures). Using common benchmark tasks in supervised and reinforcement learning, we demonstrate how this evaluation framework can both evaluate an algorithm's robustness to hyper-parameter selection and identify new areas of improvement.

1 Introduction

As new algorithms are developed, a common set of benchmark tasks are used to evaluate and compare to previous approaches. Based on the results reported on benchmark scores, it could be argued that some algorithms have attained or surpassed human level performance on specific collections, e.g., ImageNet [12] and the Arcade Learning Environment [7]. However, the style of using a single benchmark score leads researchers to tune their algorithm's hyper-parameters, e.g., steps sizes and network structures, until performance is superior to other algorithms. As a result, this optimistic approach to evaluation shows what an algorithm can achieve, not what it is likely to achieve.

The performance of an algorithm depends on four factors: the chosen hyper-parameters, seed of a random number generator, available training data, and test data used to evaluate the model's performance. The current method of evaluation does not consider all these sources of variability. Furthermore, it compresses the performance into a single metric which hides the variability of the algorithm's performance on each task. Previous work has proposed a more informative view of algorithm performance by using the the distribution of performance obtained as an algorithm's hyper-parameters are changed [2]. We extend this work by providing an experimental procedure for estimating the performance distribution as hyper-parameters, random seed, training dataset, and testing dataset are changed. Additionally, we demonstrate this evaluation method on both supervised learning (SL) and reinforcement learning (RL) benchmark tasks.

2 Current Methods of Evaluation

The current method to evaluate performance of a model or learning algorithm is to tune hyper-parameters prior to reporting performance on a held out dataset. This tune-and-test procedure splits the evaluation of the algorithm into two phases: a model search phase and an evaluation phase. The model search phase involves manual tuning or using an automated search method [1] to identify high performing hyper-parameters that score well on a portion of the training set. The evaluation phase is a single step where the tuned model is evaluated on the held out test data by computing a performance metric, e.g., classification accuracy.

The performance from this method of evaluation discards a significant amount of information as the reported score becomes a single random variable drawn from the probability distribution over the model's evaluation score. Reproducing results of this style of evaluation can be challenging. For example, previous work [5] shows that extensive hyper-parameter search fails to reach within 10 points of published SQuAD benchmark results. As a consequence, results from literature are difficult to reproduce and can lead to incorrect interpretations of performance [6]. Using a more informative style of evaluation is necessary to limit the impact of these issues.

By posing an algorithm's training procedure and scoring as a stochastic process, new and informative evaluation questions can be formulated. Consider the training and scoring of an algorithm that depends on a set of hyper-parameters, $X \in \mathcal{X}$, a training dataset, $D_{\text{train}} \in \mathcal{D}$, test dataset, $D_{\text{test}} \in \mathcal{D}$, and random seed, $S \in \mathcal{S}$, where $\mathcal{X}$ is the set of all valid hyper-parameter configurations for a particular algorithm, $\mathcal{D}$ is the set of all possible datasets, and $\mathcal{S}$ is the set of all possible random seeds. The training and scoring function, $h\colon \mathcal{X} \times \mathcal{D} \times \mathcal{D} \times \mathcal{S} \to \mathbb{R}$, that maps a sample of algorithm hyper-parameters, X, training and testing datasets D_{train} and D_{test}, and a random seed, S, to a score, $Z \in \mathbb{R}$. The standard tune and test evaluation score can be rewritten in this notation as $z = h(\hat{x}, d_{\text{train}}, d_{\text{test}}, s)$, where $d_{\text{train}} \in \mathcal{D}$ is an instance of training dataset, $d_{\text{test}} \in \mathcal{D}$ is an instance of a testing dataset, s is the random seed used, and $\hat{x} = \arg\max_{x \in \mathcal{X}} \mathbf{E}[h(x, D'_{\text{train}}, D_{\text{validation}}, S)]$, where $D'_{\text{train}}, D_{\text{validation}}$ are sampled from training dataset, d_{train}. In practice however, the hyper-parameter tuning process is often accidentally biased by the test set performance, i.e., $\hat{x} = \arg\max_{x \in \mathcal{X}} \mathbf{E}[h(x, d_{\text{train}}, d_{\text{test}}, S)]$. This results in higher performance when compared to hyper-parameters selected without any knowledge of the test set. In the next section, we will use this framework to define a new evaluation procedure.

3 Distributional Evaluation

Using the notation defined above, the performance of an algorithm can be viewed as a random variable, Z. For example, the expected performance of an algorithm where only the random seed is viewed as a random quantity is $\mathbf{E}[Z|X = x, D_{\text{train}} = d_{\text{train}}, D_{\text{test}} = d_{\text{test}}] = \sum_{s \in \mathcal{S}} h(x, d_{\text{train}}, d_{\text{test}}, s) \Pr(S = s)$. A more critical evaluation of an algorithm can be considered by taking the expectation with respect to all sources of variation e.g., $\mathbf{E}[Z]$. An algorithm which only performs well on a narrow set of hyper-parameters is likely to need additional tuning when applied to similar problems. An ideal algorithm would perform well on a new problem without requiring problem specific tuning. Thus, it is important to include variations in hyper-parameters when estimating the performance of an algorithm. This is especially important in online learning and RL where the algorithm is evaluated during training before any hyper-parameter tuning can occur.

The expected value of performance alone is insufficient to properly evaluate an algorithm as it discards information about the variance in performance. A more useful representation of performance can be obtained by considering the *cumulative distribution function* (CDF) of performance over all sources of variance. The CDF of performance for an algorithm is the function, $F_Z : \mathbb{R} \to [0, 1]$, which maps a score, z, to the probability that the algorithm achieves a score less or equal to that score, i.e., $F_Z(z) \coloneqq \Pr(Z \leq z)$ where $z \in \mathbb{R}$.

Using this distribution of performance allows for comparisons between algorithms with respect to their variance. We propose that evaluating algorithm performance should be approached in two ways: 1) visually examining the CDF of performance and 2) evaluating numerically by statistics computed from the CDF. The visual investigation by plotting the CDF allows for a quick interpretation of an algorithm's variability and range of performance. A numerical evaluation can summarize an algorithm's performance with respect to task specific performance constraints. For example, if a task has a minimum threshold of performance required, z_{thresh}, then an algorithm can be scored based on the area under the CDF curve that meets this threshold, i.e., $\int_{z_{\text{thresh}}}^{\infty} z f_Z(z) dz$, where $f_Z(z)$ is the probability density of performance at z. Another similar measure is Conditional Value-At-Risk (CVaR) [11] which computes the expected value of a random variable above a probability, α, i.e., $\text{CVaR}_\alpha(Z) = \mathbf{E}[Z|Z \geq F_Z^{-1}(\alpha)]$, where $F_Z^{-1}(\alpha) \coloneqq \arg\min_z \{z | F_Z(z) \geq \alpha\}$ is the inverse CDF of F_Z. When using CVaR, algorithms are evaluated based on the average performance of the top $(1 - \alpha)$ percent of trials, i.e., $\text{CVaR}_{0.9}(Z)$ is the average performance of the top 10% of trials. There exist another CVaR measure that computes the expected value below a probability, α, which, would provide a measure of risk in this setting. However, in this work we use the first definition of CVaR.

We present our procedure, Distributional Sampling for Evaluation (DSE), for generating the empirical CDF of an algorithm's performance in Algorithm 1. In order to conduct this type of evaluation there needs to be a method for sampling hyper-parameters for each algorithm. How this method is defined can greatly affect results. For example, if one algorithm uses a sampling method that mostly selects poor performing hyper-parameter combinations and another that only samples near optimal combinations the former algorithm will appear arbitrarily worse than the latter. For this reason an algorithm should specify how hyper-parameters should be selected. In lieu of these specifications, hyper-parameter combinations should be sampled uniformly from ranges found in literature showing the algorithm performing well on similar problems. This leaves open an avenue to improve the understanding of an existing algorithm by identifying how ranges should be set.

While DSE is specified for hyper-parameters to be randomly sampled, automated tuning of the hyper-parameters can be incorporated through modification of the training and evaluation function, f. For example, inside f the hyper-parameters could be tuned using random search and cross validation over the training set. In the experiments below, we use both DSE with automated hyper-parameter tuning and DSE with random hyper-parameter sampling.

At first glance our approach appears to require additional computation time, however, time is often already spent tuning hyper-parameters. Instead of discarding the results from sub par hyper-parameter configurations, it could instead be used to generate the distribution of performance. Additional computation time is thus minimized and the comparison to other algorithms is fair in that all algorithms are run for the same number of trials.

Algorithm 1: DSE: Distributional Sampling for Evaluation

```
Input: algorithm training and evaluation function, $f$, hyper-parameter sampling function, $\phi$,
data sampling function, $\psi$, and number of trials to run, $N$.
Return: empirical CDF, $F$, of the performance metric
$\tau \leftarrow []$ Initialize list of performance results
for for each random seed do
    $s \sim uniform(\mathcal{S})$ sample a random seed
    $x \leftarrow \phi(s)$ sample set of hyper-parameters
    $d_{train}, d_{test} \leftarrow \psi(s)$ sample datasets
    $z \leftarrow f(x, d_{train}, d_{test}, s)$ get performance metric
    $\tau \leftarrow \tau + [z]$ add performance metric to list
$F \leftarrow compute\_cdf(\tau)$ Compute empirical CDF
```

4 Experiments

In this section we provide examples showing the results of DSE on several classification tasks and an RL task. In the classification task we compare five algorithms, support vector machines (SVM), k-nearest neighbors (kNN), random forests (RF), logistic regression (LR), and neural networks (NN). Each algorithm is evaluated on four standard classification tasks found in the scikit-learn repository [10]: moons, circles, linear, and face identification. The moons task involves classifying points on one of two intersecting half circles, the circles task involves classifying points on an inner and outer circle, the linear task separates classes by hyper-planes and points are noisily generated on both sides. The face identification task uses images from the Labeling Faces in the Wild dataset [8].

We run two different experimental setups for the supervised classification tasks: DSE with automated hyper-parameter tuning, and DSE with randomly sampled hyper-parameters. The same hyper-parameter ranges were used for both versions and the details can be found in Appendix A.

In RL, a standard way to report results is to provide a learning curve plot that shows average total reward obtained from each episode. This method is subject to high variance and not able to capture the sensitivity of an algorithms to its hyper-parameters. To see how the distributional approach to evaluation benefits RL we compare two algorithms: Actor-Critic (AC) [14] and Proximal Policy Optimization (PPO) [13] on the benchmark task pendulum swing-up and balance [4]. For both of these algorithms hyper-parameters are sampled randomly from fixed ranges which are provided in Appendix B.

5 Results

In this section we present the results from each supervised learning task and the RL task.

5.1 Supervised Learning

The performance distributions for both DSE approaches are given in Figure 1 and the performance metrics are provided in Table 1. The provided plots are of the inverse CDF of the performance distribution where the x-axis is the cumulative probability and the y-axis is the accuracy such that the proportion of trials that have a value less than or equal to y is x.

Examining the performance distribution of DSE with automated-tuning, it is clear that the algorithms share a similar distribution shape for each problem. However, there exist cases when an algorithm deviates from this behavior, e.g., logistic regression on the circles and linear problems. This kind of insight cannot be captured by reporting mean and standard deviation of performance.

In each distribution there exists an uptick and downtick in performance on the tails of each plot that indicates rare coupling of hyper-parameters, random seed, and dataset variation that lead to exceptional changes in performance. The reporting of upticks in performance could be used to falsely show a new method as being superior. For this reason, only reporting performance from a few trials is insufficient to claim superiority of one algorithm to another. Reporting a metric that accounts for the variation in the performance distribution is more likely to indicate an increase of an algorithm's utility. This can be seen in the performance results in Table 1.

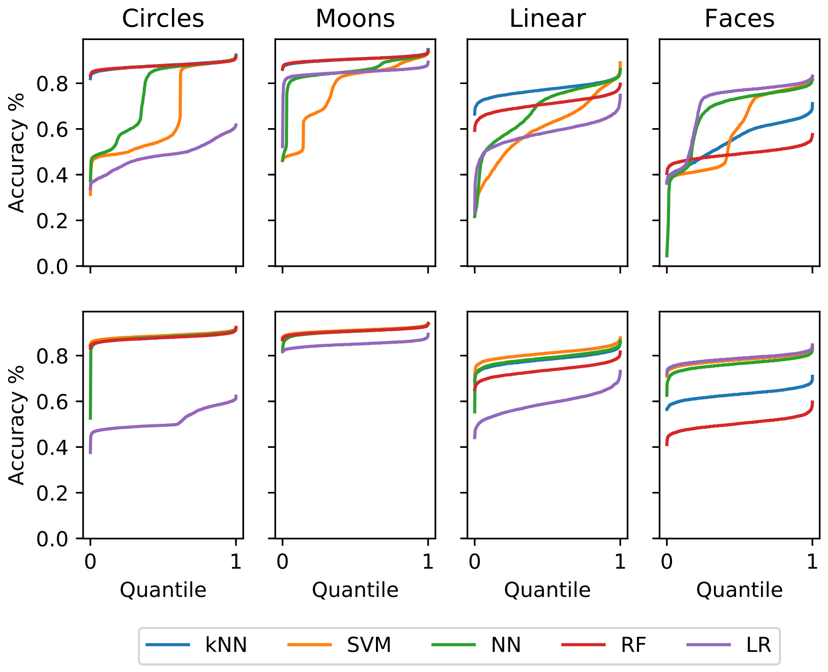

Figure 1: Inverse CDF of test accuracy when hyper-parameters are randomly sampled (top) and when hyper-parameters are tuned (bottom).

In DSE without hyper-parameter tuning approach it becomes clear which algorithms are robust to hyper-parameter choices. The non-parametric algorithms: kNN and RF, both are less sensitive to hyper-parameter choices compared to parametric algorithms. The SVM algorithm which was consistently the top performer when using automated hyper-parameter tuning (bottom row in Figure 1) dropped significantly in performance when hyper-parameters were randomly chosen (top row in

Figure 1). The gap in performance distributions between the two DSE methods indicates that there is room to improve these algorithms by developing an understanding how hyper-parameters should be set for each problem. By using the distribution of performance generated by DSE, these types of improvements can be observed.

	DSE-Tuning				**DSE**			
Algorithm	**Moons**	**Circles**	**Linear**	**Faces**	**Moons**	**Circles**	**Linear**	**Faces**
k-NN	**0.918**	0.893	0.803	0.649	**0.914**	**0.889**	**0.795**	0.624
RF	0.915	0.888	0.757	0.523	**0.914**	**0.887**	0.733	0.512
SVM	**0.919**	**0.897**	**0.828**	0.796	0.876	0.814	0.716	0.751
LR	0.859	0.552	0.627	**0.802**	0.859	0.531	0.624	**0.788**
NN	0.916	0.892	0.810	0.782	0.892	0.884	0.782	0.762

Table 1: The CVaR$_{0.5}$ performance statistic for each classification task when the hyper-parameters are automatically tuned (left), and randomly sampled (right).

5.2 Reinforcement Learning

In this section we examine the results of the RL experiment by plotting the distributions of performance in Figure 2. The inverse CDF in Figure 2 shows that on this environment, with this method of selecting hyper-parameters, the Actor-Critic algorithm is likely to outperform PPO. Both the mean and CVaR$_{0.5}$ performance metrics in Table 2 corroborate this finding. With both algorithms able to achieve near optimal performance, this result could not be found using standard evaluation methods. To compare to the standard evaluation method in RL we provide learning curves from single sets of hyper-parameters for PPO in Appendix C. This further shows that the learning curves do not provide a clear picture of the performance metric compared to plotting the distribution of performance.

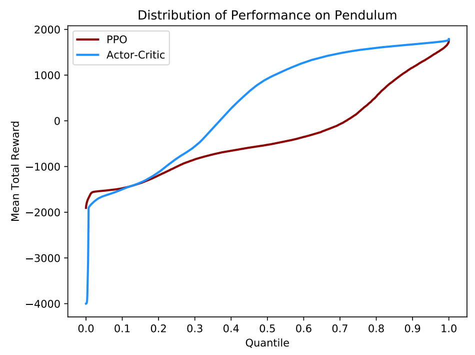

Figure 2: The inverse CDF of mean total reward for PPO and Actor-Critic critic algorithms on the pendulum environment.

	Performance Metric	
Algorithm	**Mean**	**CVaR$_{0.5}$**
Actor-Critic	388.3	1476.2
PPO	-342.1	363.8

Table 2: The mean and CVaR metrics computed from the empirical CDF.

6 Conclusion

We examine the volatility of a variety of machine learning methods and demonstrate that using traditional evaluation metrics is not an effective way to distinguish performance. Furthermore, the proposed DSE framework provides an effective way to not only determine which method is more likely to outperform the other, but how robust each method is to its experimental configuration. Future work on this approach should focus on principled methods to determine the number of trials needed to accurately evaluate an algorithm and provide statistical guarantees on performance.

7 Acknowledgements

This work was supported in part by the Center for Intelligent Information Retrieval. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

References

[1] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pages 2546–2554, 2011.

[2] Daniel Cohen, Scott M. Jordan, and W. Bruce Croft. Distributed evaluations: Ending neural point metrics. *CoRR*, abs/1806.03790, 2018.

[3] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

[4] Kenji Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):219–245, 2000.

[5] Alexander Dür, Andreas Rauber, and Peter Filzmoser. Reproducing a neural question answering architecture applied to the squad benchmark dataset: Challenges and lessons learned. In *Advances in Information Retrieval - 40th European Conference on IR Research, ECIR 2018, Grenoble, France, March 26-29, 2018, Proceedings*, pages 102–113, 2018.

[6] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3207–3214, 2018.

[7] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *CoRR*, abs/1803.00933, 2018.

[8] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[9] George Konidaris, Sarah Osentoski, and Philip S. Thomas. Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011.

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

[11] R. Tyrrell Rockafellar and Stanislav Uryasev. Optimization of conditional value-at-risk. *Journal of Risk*, 2:21–41, 2000.

[12] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

Appendix A Classification Experimental Details

The moons, circles, and linear tasks have generatable datasets. For each task $2{,}000$ points are randomly generated and the train and test data is randomly selected with a $60\%/40\%$ train/test split. The linear tasks has features in $\mathbb{R}^{10}$ and there are 4 classes with 2 clusters per class. The face identification task uses images from the Labeling Faces in the Wild dataset [8], which has 1288 data points, 7 classes, and the features used are the projection of the images on to the top 150 eigenvectors. Since the face identification tasks is not generatable, the randomness in the data only comes from a random train/test split. All algorithms and datasets used were implemented in the scikit-learn repository [10]. Each algorithm and classification problem was run for 2,000 iterations of DSE. For DSE with hyper-parameter tuning, the training and evaluation function, f, includes an inner training loop that tunes the hyper-parameters. During the tuning process, 50 random hyper-parameter configurations are sampled and the best performing set on a 3-fold cross validation of the training dataset is used to evaluate the algorithm on the test set. DSE with tuning approach was ran for $2{,}000$ iterations.

The hyper-parameters for each algorithm were selected as follows. For the k-nearest neighbors algorithm the number of neighbors was chosen uniformly from the set $\{3, 4, 5, 10, 25, 50\}$. All other parameters for the k-nearest neighbors algorithm were left as the defaults. In the SVM algorithm the C parameter was selected from a (natural) log uniform distribution in the range$[0.01, 100]$. The SVM kernel was chosen uniformly between the set $\{$linear, polynomial, RBF, sigmoid$\}$. When the kernel was selected to be polynomial the degree was chosen uniformly from the set $\{2, 3, 4, 5\}$. All other SVM parameters were left at their default values. The random forest algorithm used a number of estimators to from a (natural) log uniform distribution in the range $[10, 100]$. The decision criterion was chosen uniformly between Gini impurity and entropy. The minimum number of samples to split a node was selected uniformly from the set $\{2, 4, 8, 16\}$ and the minimum number of samples in a leaf node was selected uniformly from the set $\{1, 3, 5\}$. The logistic regression algorithm used the saga solver, an $l2$ penalty on the weights, maximum iterations of $1{,}000$, and fit the y-intercept. Additionally the C parameter for logistic regression was sampled form a log uniform random distribution over the range $[0.0001, 1{,}0000]$. The neural network used the adam optimizer with a constant learning rate, early stopping and a max number of epochs of 200. The activation function of neural network was chosen uniformly from the set $\{$logistic, tanh, relu$\}$. The hidden layer sizes were chosen uniformly random from combinations of $[50, 100, 150]$ units per layer and using one or two layers. The learning rate was randomly sampled from (natural) log uniform distribution over the range $[0.0001, 0.1]$.

Appendix B RL Experimental Details

To see how the distributional approach to evaluation benefits RL we compare two algorithms Actor-Critic (AC) [14] and Proximal Policy Optimization (PPO) [13] on the benchmark task pendulum swing-up and balance [4]. For both of these algorithms hyper-parameters are sampled randomly from the ranges which provided below. The ranges for PPO were determined from the paper introducing it with sight modifications to make it perform better on this domain. The ranges were Actor-Critic were set from a combination of parameters found in the literature. Both algorithms use a linear policy and value function using the Fourier basis [9]. Each algorithm was run with repeated trials for 12 hours spread across $1,000$ CPUs. The Actor-Critic algorithm was run for significantly more trials, $3,041,685$, than PPO, $17,415$, because it was implemented in c++ and PPO uses the python implementation in the OpenAI Baselines repository [3].

Both algorithms used the same linear policy structure that used the Fourier basis with an independent order of six and a dependent order of 5. Each algorithm also used a Fourier basis function for the critic to use with independent and dependent orders uniformly sampled from a range $[0, 10]$. Additionally, both algorithms used the reward discount parameter $\gamma = 0.99$, a λ-return mixing parameter sampled uniformly from $[0, 1]$. The actor-critic algorithm used actor and critic learning rates sampled from a log uniform distribution from the range $[0.0001, 0.1]$. The PPO algorithm used learning rate sampled from a long uniform distribution over the range $[0.00001, 0.01]$, the steps-per-batch parameter was selected from a log (base 2) uniform range of $[16, 256]$, the number of epochs was chosen uniformly over the range $[1, 15]$, the batch size was selected uniformly from the range $[8, 64]$, and the importance weights clipping parameter was selected uniformly from the range $[0.1, 0.3]$. All other PPO parameters were left as the defaults in the baselines repository.

Appendix C Comparison to learning curves

To compare the distributional evaluation to the classic evaluation of learning curves we provide learning curves for PPO with hyper-parameters selected from the 1, 50, 95, 99 percentiles of performance on the CDF. To show the variance of performance, a distributional plot for each set of hyper-parameters is given in Figure 3. In this plot it is clear the 99-percentile parameters are superior to the 95-percentile parameters in most of the trials, however there are around 10% of trials that contain oscillatory or diverging behavior that causes the mean return to drop. The fact that some sets of hyper-parameters of a high amount of variance in performance is why performance of an algorithm should be considered over variations in hyper-parameters and the distribution of performance should be shown.

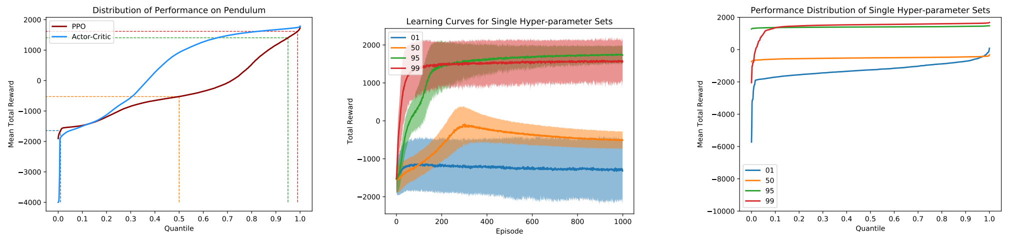

Figure 3: **Left**: Solid lines represent the inverse CDF of mean lifetime return for PPO and Actor-Critic critic algorithms on the pendulum environment. Dashed lines represent the slice of the performance curved used in the middle and right plots. Middle: Learning curves for the PPO algorithm when the hyper-parameters are selected from the quantiles show in the left plot. Solid lines indicate mean return per episode with errors bars representing the standard deviation above and below the mean computed over $1,000$ trials. **Right**: Performance distribution of the same learning curves as the middle plot.