Horizon: Facebook's Open Source Applied Reinforcement Learning Platform

Jason Gauci [1] **Edoardo Conti** [1] **Yitao Liang** [1] **Kittipat Virochsiri** [1] **Yuchen He** [1] **Zachary Kaden** [1]
Vivek Narayanan [1] **Xiaohui Ye** [1]

Abstract

In this paper we present Horizon, Facebook's open source applied reinforcement learning (RL) platform. Horizon is an end-to-end platform designed to solve industry applied RL problems where datasets are large (millions to billions of observations), the feedback loop is slow (vs. a simulator), and experiments must be done with care because they don't run in a simulator. Unlike other RL platforms, which are often designed for fast prototyping and experimentation, Horizon is designed with production use cases as top of mind. The platform contains workflows to train popular deep RL algorithms and includes data preprocessing, feature transformation, distributed training, counterfactual policy evaluation, and optimized serving. We also showcase real examples of where models trained with Horizon significantly outperformed and replaced supervised learning systems at Facebook.

1 Introduction

Deep reinforcement learning (RL) is poised to revolutionize how autonomous systems are built. In recent years, it has been shown to achieve state-of-the-art performance on a wide variety of complicated tasks (Mnih et al., 2015; Lillicrap et al., 2015; Schulman et al., 2015; Van Hasselt et al., 2016; Schulman et al., 2017), where being successful requires learning complex relationships between high dimensional state spaces, actions, and long term rewards. However, the current implementations of the latest advances in this field have mainly been tailored to academia, focusing on fast prototyping and evaluating performance on simulated benchmark environments.

While interest in applying RL to real problems in industry is high, the current set of implementations and tooling must be adapted to handle the unique challenges faced in applied settings. Specifically, the handling of large datasets with hundreds or thousands of varying feature types and distributions, high dimensional discrete and continuous action spaces, optimized training and serving, and algorithm performance estimates before deployment are of key importance.

With this in mind, we introduce Horizon - an open source end-to-end platform for applied RL developed and used at Facebook. Horizon is built in Python and uses PyTorch for modeling and training (Paszke et al., 2017) and Caffe2 for model serving (Jia et al., 2014). It aims to fill the rapidly-growing need for RL systems that are tailored to work on real, industry produced, datasets. To achieve this goal, we designed our platform with the following principles in mind.

- *Ability to Handle Large Datasets Efficiently*
- *Ability to Preprocess Data Automatically & Efficiently*
- *Competitive Algorithmic Performance*
- *Algorithm Performance Estimates before Launch*
- *Flexible Model Serving in Production*
- *Platform Reliability*

The rest of this paper goes into the details and features of Horizon, but at a high level Horizon features:

Data preprocessing: A Spark (Zaharia et al., 2010) pipeline that converts logged training data into the format required for training numerous different deep RL models.

Feature Normalization: Logic to extract metadata about every feature including type (float, int, enum, probability, etc.) and method to normalize the feature. This metadata is then used to automatically preprocess features during training and serving, mitigating issues from varying feature scales and distributions which has shown to improve model performance and convergence (Ioffe & Szegedy, 2015).

Deep RL model implementations: Horizon provides implementations of Deep Q-networks (DQN) (Mnih et al.,

[1] Facebook, Menlo Park, California, USA. Correspondence to: Jason Gauci <jjg@fb.com>, Edoardo Conti <edoardoc@fb.com>, Kittipat Virochsiri <kittipat@fb.com>.

2015), Deep Q-networks with double Q-learning (DDQN) (Van Hasselt et al., 2016), Deep Q-networks with dueling architecture (Dueling DQN & Dueling DDQN) (Wang et al., 2015) for discrete action spaces, a parametric action version of all the previously mentioned algorithms for handling very large discrete action spaces, and Deep Deterministic Policy Gradients (DDPG) (Lillicrap et al., 2015) for continuous action spaces.

Multi-GPU training: Industry datasets can be very large. At Facebook many of our datasets contain tens of millions of samples per day. Internally, Horizon has functionality to conduct training on many GPUs distributed over numerous machines. This allows for fast model iteration and high utilization of industry sized clusters. Even for problems with very high dimensional feature sets (hundreds or thousands of features) and millions of training examples, we are able to learn models in a few hours (while doing preprocessing and counterfactual policy evaluation on every batch). As part of the initial open source release, Horizon supports CPU, GPU, and multi-GPU training on a single machine.

Counterfactual policy evaluation: Unlike in pure research settings where simulators offer safe ways to test models and time to collect new samples is very short, in applied settings it is usually rare to have access to a simulator. This makes offline model evaluation important as new models affect the real world and time to collect new observations and retrain models may take days or weeks. Horizon scores trained models offline using several well known counterfactual policy evaluation (CPE) methods. The step-wise importance sampling estimator, step-wise direct sampling estimator, step-wise doubly-robust estimator (Dudık et al., 2011), sequential doubly-robust estimator (Jiang & Li, 2016)[1], and MAGIC estimator (Thomas & Brunskill, 2016) are all run as part of Horizon's end-to-end training workflow.

Optimized Serving: Post training, models are exported from PyTorch to a Caffe2 network and set of parameters via ONNX (Exchange, 2018). Caffe2 is optimized for performance and portability, allowing models to be deployed to thousands of machines.

Tested Algorithms: Testing production RL systems is a new area with no established best practices. We take inspiration from systems best practices and test our core functionality and algorithms in Horizon via unit tests and integration tests. Using custom environments (i.e. Gridworld) and some standard environments from OpenAI's Gym (Brockman et al., 2016) we train and evaluate all of our RL models on every pull request.

We end the paper discussing examples of how models trained with Horizon outperformed supervised learning and heuristic based policies to send notifications and to stream video at Facebook. We provide details into the formulation and methods used in our approach.

2 Data Preprocessing

Many RL models are trained on consecutive pairs of state/action tuples (DQN, DDPG, etc.). However, in production systems data is often logged as it comes in, requiring offline logic to join the data in a format suitable for RL. To assist in creating data in this format, Horizon includes a Spark pipeline (called the *Timeline* pipeline) that transforms logged data collected in the following row format:

- *MDP ID*: A unique ID for the Markov Decision Process (MDP) chain that this training example is a part of.
- *Sequence Number*: A number representing the location of the state in the MDP (i.e. a timestamp).
- *State Features*: The features of the current step that are independent of the action.
- *Action*: The action taken at the current step. A string (i.e. 'up') if the action is discrete or a set of features if the action is parametric or continuous.
- *Action Probability*: The probability that the current system took the action logged. Used in counter factual policy evaluation.
- *Reward*: The scalar reward at the current step.
- *Possible Actions*: An array of possible actions at the current step, including the action chosen (left blank for continuous action domains). This is optional but enables Q-Learning (vs. SARSA).

This data is transformed into data in the row format below. Note, *MDP ID*, *Sequence Number*, *State Features*, *Action*, *Action Probability*, and *Reward* are also present in the data below, but are left out for brevity.

- *Next State Features*: The features of the subsequent step that are action-independent.
- *Next Action*: The action taken at the next step.
- *Sequence Number Ordinal*: A number representing the location of the state in the MDP after the *Sequence Number* was converted to an ordinal number.
- *Time Diff*: A number representing the "time difference" between the current state and next state (computed as the difference in non-ordinal sequence numbers between states). Used as an optional way to set varying time differences between states. Particularly useful for MDPs that have been sub-sampled upstream.

[1] Two variants are implemented; one makes uses of ordinal importance sampling and the other weighted importance sampling.

- *Possible Next Actions*: A list of actions that were possible at the next step. Only present if *Possible Actions* were provided.
- *Reward Timeline*: A map containing the future rewards. Each key is the number of timesteps forward, and the value is the reward at that timestep. This column is used to measure model performance in offline evaluation.
- *Episode Value*: The sum of discounted future rewards over the MDP. This column is used to measure model performance in offline evaluation.

Internally, the *Timeline* operator is run on a Hive table containing logged data in the format described at the beginning of this section. After running the operator, post-timeline data is written to a new Hive table. In the simple examples provided in the open source Horizon repository, we read pre-timeline data from a local JSON file and write post-timeline data to a local JSON file. This is just to provide simple working examples for end users and using the Spark operator on data in other formats (e.g. Hive) is straightforward.

3 Feature Normalization

Data from recommender systems is often sparse, noisy and arbitrarily distributed (Adomavicius & Tuzhilin, 2005). Literature has shown that neural networks learn faster and better when operating on batches of features that are normally distributed (Ioffe & Szegedy, 2015). In RL, where the recurrence can become unstable when exposed to very large features, feature normalization is even more important. For this reason, Horizon includes a workflow that automatically analyzes the training dataset and determines the best transformation function and corresponding normalization parameters for each feature. Developers can override the estimation if they have prior knowledge of the feature that they prefer to use.

In the workflow, features are identified to be of type binary, probability, continuous, enum, quantile, or boxcox. A "normalization specification" is then created which describes how the feature should be normalized during training. To identify the type, we follow the process outlined in algorithm 1.

Algorithm 1 Identify feature F

```
if All values in $F$ are in $\{0, 1\}$ then
    $F$ is a binary feature
else if All values in $F$ are in the range $[0, 1]$ then
    $F$ is a probability
else if All values in $F$ are integers with $< N$ unique values then
    $F$ is a categorical feature
else if $F$ is approximately normally distributed then
    $F$ is a continuous feature
else if $F$ is approximately normally distributed after a boxcox transformation then
    $F$ is a boxcox feature
else
    $F$ is a quantile feature
end if
```

Although we pre-compute the feature transformation functions prior to training, we do not apply the feature transformation to the dataset until during training. At training time we create a PyTorch network that takes in the raw features and applies the normalization during the forward pass. This allows developers to quickly iterate on the feature transformation without regenerating the dataset. The feature transformation process begins by grouping features according to their identity (see above), and then processing each group as a single batch using vector operations.

4 Model Implementations

Horizon contains implementations of several deep RL algorithms that span to solve discrete action, very large discrete action, and continuous action domains. We also provide default configuration files as part of Horizon so that end users can easily run these algorithms on our included test domains (e.g. OpenAI Gym (Brockman et al., 2016), Gridworld). Below we describe the current algorithms supported in Horizon.

4.1 Discrete-Action Deep Q-Network (Discrete DQN)

For discrete action domains with a tractable number of actions, we provide a Deep Q-Network implementation (Mnih et al., 2015). In addition, we provide implementations for several DQN improvements, including double Q-learning (Van Hasselt et al., 2016) and dueling architecture (Wang et al., 2015). We plan on continuing to add more improvements to our DQN model (distributional DQN (Bellemare et al., 2017), multi-step learning (Sutton et al., 1998), noisy nets (Fortunato et al., 2017)) as these improvements have been shown to stack to achieve state of the art results on numerous benchmarks (Hessel et al., 2017).

4.2 Parametric-Action Deep-Q Network (Parametric DQN)

Many domains at Facebook have have extremely large discrete action spaces (more than millions of possible actions) with actions that are often ephemeral. This is a common challenge when working on large scale recommender systems where an RL agent can take the action of recommending numerous different pieces of content. In this setting, running a traditional DQN would not be practical. One alternative is to combine policy gradients with a K-NN search (Dulac-Arnold et al., 2015), but when the number of available actions for any given state is sufficiently small, this approach is heavy-handed. Instead, we have chosen to create a variant of DQN called Parametric-Action DQN, in which we input concatenated state-action pairs and output the Q-value for each pair. Actions, along with states, are represented by a set of features. The rest of the system remains as a traditional DQN. Like our Discrete-Action DQN implementation, we also have adapted the double Q-learning and dueling architecture improvements to the Parametric-Action DQN.

4.3 Deep Deterministic Policy Gradient (DDPG)

Other domains at Facebook involve tuning of sets of hyperparameters. These domains can be addressed with a continuous action RL algorithm. For continuous action domains we have implemented Deep Deterministic Policy Gradients (DDPG) (Lillicrap et al., 2015).

Support for other deep RL algorithms will be a continued focus going forward.

5 TRAINING

Once we have preprocessed data and a feature normalization function for each feature, we can begin training. Training can be done using CPUs, a GPU, or multiple GPUs. Internally, Horizon has functionality to conduct training on many GPUs distributed over numerous machines. We utilize the PyTorch multi-GPU functionality to do distributed training (Paszke et al., 2017). As part of the open source release, Horizon supports CPU, GPU, and multi-GPU training on a single machine. Multi-GPU training across numerous machines is expected to be added to Horizon open source in the near future.

Using GPU and multi-GPU training we are able to train large RL models that contain hundreds to thousands of features across tens of millions of examples in a few hours (while doing feature normalization and counterfactual policy evaluation on every batch).

6 REPORTING AND EVALUATION

There are several metrics that can inform engineers about the performance of their RL models after training.

Temporal difference loss (TD-loss) measures the function approximation error. For example, in DQN, this measures the difference between the expected value of Q given by the bellman equation, and the actual value of Q output by the model. Note that, unlike supervised learning where the labels are from a stationary distribution, in RL the labels are themselves a function of the model and as a result this distribution shifts. As a result, this metric is primarily used to ensure that the optimization loop is stable. If the TD-loss is increasing in an unbounded way, we know that the optimization step is too aggressive (e.gs. the learning rate is too high, or the minibatch size is too small).

Monte-Carlo Loss (MC-loss) compares the model's Q-value to the logged value (the discounted sum of logged rewards). When the logged policy is the optimal policy (for example, in a toy environment), MC-loss is a very effective measure of the model's policy. Because the logged policy is often not the optimal policy, the MC-loss has limited usefulness for real-world domains. Similar to TD-loss, we primarily monitor MC-loss for extreme values or unbounded increase.

Because RL is focused on policy optimization, it is more valuable to evaluate the policy (i.e. what action a model chooses) than to evaluate the model scores directly. Horizon has a comprehensive set of Counterfactual Policy Evaluation techniques.

6.1 Counterfactual Policy Evaluation

Counterfactual policy evaluation (CPE) is a set of methods used to predict the performance of a newly learned policy without having to deploy it online. CPE is important in applied RL as deployed policies affect the real world. At Facebook, we serve billions of people every day; deploying a new policy directly impacts the experience they have using Facebook. Without CPE, industry users would need to launch numerous A/B tests to search for the optimal model and hyperparameters. These experiments can be time-consuming and costly. With reliable CPE, this search work can be fully automated using hyperparameter sweeping techniques that optimize for a model's CPE score. CPE also makes an efficient and principled parameter sweep possible by combining counter-factual offline estimates with real-world testing.

Horizon includes implementations of the following CPE estimators that are automatically run as part of training:

- Step-wise importance sampling estimator
- Step-wise direct sampling estimator
- Step-wise doubly-robust estimator (Dudık et al., 2011)
- Sequential doubly-robust estimator (Jiang & Li, 2016)
- MAGIC estimator (Thomas & Brunskill, 2016)

Incorporating the aforementioned estimators into our platform's training pipeline provides us with two advantages: (1) all feature normalization improvements tailored to training are also available to CPE (2) users of our platform get CPE estimates at the end of each epoch which helps them understand how more training affects model performance.

The CPE estimators in Horizon are also optimized for running speed. The implemented estimators incur minimal time overhead to the whole training pipeline.

The biggest technical challenge implementing CPE stems from the nature of how batch RL is trained. To decrease temporal correlation of the training data, which is needed for stable supervised learning, a pseudo i.i.d environment is created by uniformly shuffling the collected training data (Mnih et al., 2015). However, the sequential doubly robust and MAGIC estimators both are built based on cumulative step-wise importance weights (Jiang & Li, 2016; Thomas & Brunskill, 2016), which require the training data to appear in its original sequence. In order to satisfy this requirement while still using the shuffled pseudo i.i.d data in training, we sample and collect training samples during the training workflow. At the end of every epoch we then sort the collected samples to place them back in their original sequence and conduct CPE on the collected data. Such deferral provides the opportunity to calculate all needed Q-values together in one run, heavily utilizing matrix operations. As a side benefit, querying for Q-values at the end of one epoch of training decreases the variance of CPE estimates as the Q-function can be very unstable during training. Through this process we are able to calculate reliable CPE estimations efficiently.

6.2 TensorboardX

To visualize the output of our training process, we export our metrics to tensorboard using the TensorboardX plugin (Huang, 2018). TensorboardX outputs tensors from pytorch/numpy to the tensorboard format so that they can be viewed with the Tensorboard web visualization tool.

7 Model Serving

At Facebook, we serve deep reinforcement learning models in a variety of production applications. The serving platform is designed to support stochastic policies without requiring online learning. We do this by producing both raw scores and the outcomes from a deterministic policy and softmax sampled policy as part of one forward pass.

The deterministic policy always chooses the highest-scoring action. While this policy has no exploration, it is still useful for collecting metrics, especially when doing an A/B test with another deterministic model.

The softmax policy converts scores to propensities using a softmax function with temperature (Sutton et al., 1998) and then samples an action from these propensities.

PyTorch 1.0 supports ONNX (Exchange, 2018), an open source format for model inference. ONNX works by tracing the forward pass of an RL model, including the feature transformation and the policy outputs. The result is a Caffe2 network and a set of parameters that are serializable, portable, and efficient. This package is then deployed to thousands of machines.

At serving time, product teams can either execute one of our policies, or fetch the scores from one of our models and develop their own policy. Either way, product teams log the possible actions, the propensity of choosing each of these actions, the action chosen, and the reward received. Depending on the problem domain, it may be hours or even days before we know the reward for a particular sample. Product teams typically log a unique key with each sample so they can later join the logged training data to other data sources that contain the reward. This joined data is then fed back into Horizon to incrementally update the model. Although all of our algorithms are off-policy, they are still limited based on the policy that they are observing, so it is important to train in a closed loop to get the best results. In addition, the data distribution is changing and the model needs to adapt to these changes over time.

8 Platform Testing Practices

Like general software systems, adequate testing in machine learning systems is important for catching algorithmic performance regressions and other issues. To test algorithm performance, Horizon is integrated with both custom environments (i.e. a self made Gridworld environment) and the popular benchmarking library OpenAI Gym (Brockman et al., 2016). Internally, when new pull requests are made, a suite of unit tests and integration tests are started that test platform core functionality (data pre-processing, feature normalization, etc.) and also algorithmic performance. Specifically, for algorithmic performance, both our Discrete-Action DQN and Parametric-Action DQN models are evaluated on OpenAI Gym's Cartpole environment while our DDPG model is evaluated on OpenAI Gym's Pendulum environment. We evaluate these models with different configurations (using Q-learning vs. SARSA, with and without double Q-learning, etc.) to ensure robustness and correctness. For open source, we have set up a continuous integration test that runs all unit tests upon push to the master branch on Github and on pull request submission. The integration test runs on pre-built Docker images of the target platforms. We have included the Dockerfile used to build the images to ensure that the test environment is reproducible.

9 Case Study: Notifications at Facebook

9.1 Push Notifications

Facebook sends notifications to people to connect them with the most important updates when they matter, which may include interactions on your posts or stories, updates about your friends, joined groups, followed pages, interested events etc. Push notifications are sent to mobile devices, and a broader set of notifications is accessible from within the app/website. It is primarily used as a channel for sending personalized and time sensitive updates. To make sure we only send the most personally relevant notifications to people, we filter notification candidates using machine learning models. Historically, we have used supervised learning models for predicting click through rate (CTR) and likelihood that the notification leads to meaningful interactions. These predictions are combined into a score that is used to filter the notifications.

This however, didn't capture the long term or incremental value of sending notifications. There can be some signals that appear long after the decision to send or drop is made or can't be attributed directly to the notification. Additionally, because notification preference varies from person to person, filtering based on a static threshold misses out on the improved experience of tailoring notifications for people with different sensitivities to being notified.

We introduced a new policy that uses Horizon to train a Discrete-Action DQN model for sending push notifications to address the problems above. The Markov Decision Process (MDP) is based on a sequence of notification candidates for a particular person. The actions here are sending and dropping the notification, and the state describes a set of features about the person and the notification candidate. There are rewards for interactions and activity on Facebook, with a penalty for sending the notification to control the volume of notifications sent. The policy optimizes for the long term value and is able to capture incremental effects of sending the notification by comparing the Q-values of the send and don't send action.

The model was incrementally retrained daily on data from people exposed to the model with some action exploration introduced during serving. The model is updated with batches of tens of millions of state transitions. We observed this to help online usage metrics as we are doing off policy batch learning.

We observed a significant improvement in activity and meaningful interactions by deploying an RL based policy for certain types of notifications, replacing the previous system based on supervised learning.

9.2 Page Administrator Notifications

In addition to Facebook users, page administrators also rely on Facebook to provide them with timely updates about the pages they manage. In the past, supervised learning models were used to predict how likely page admins were to be interested in such notifications and how likely they were to respond to them. Although the models were able to help boost page admins' activity in the system, the improvement always came at some trade-off with the notification quality, e.g. the notification click through rate (CTR). With Horizon, a Discrete-Action DQN model is trained to learn a policy to determine whether to send or not send a notification based on the state represented by hundreds of features. The training data spans multiple weeks to enable the RL model to capture page admins' responses and interactions to the notifications with their managed pages over a long term horizon. The accumulated discounted rewards collected in the training allow the model to identify page admins with long term intent to stay active with the help of being notified. After deploying the DQN model, we were able to improve daily, weekly, and monthly metrics without sacrificing notification quality.

9.3 More Applications of Horizon

In addition to making notifications more relevant on our platform, Horizon is applied by a variety of other teams at Facebook. The 360-degree video team has applied Horizon in the adaptive bitrate (ABR) domain to reduce bitrate consumption without harming people's watching experience. This was due to more intelligent video buffering and pre-fetching.

While we focused our case studies on notifications, it is important to note that Horizon is a horizontal effort in use or being explored to be used by many organizations within the company.

10 Future Work

The most immediate future additions to Horizon fall into 2 categories - 1) New models & model improvements 2) CPE integrated with real metrics.

New models & model improvements: Specifically, on the model improvement and new models front, we will be adding more incremental improvements to our current models and plan on continually adding the best performing algorithms from the research community.

CPE integrated with real metrics: Many developers struggle with deriving a single reward scalar that defines the success of a policy. Rather, they look at a suite of metrics and watch how these metrics change in concert as the policy changes. In the future, Horizon will allow developers to input a set of metrics that they are interested in tracking and we will use CPE to estimate the change to these metrics, independent of the reward CPE. With these additional tools, the reward shaping process will become more intuitive and we can eventually support more complicated representations of rewards, such as an objective function subject to a set of constraints.

We plan on continuing to improve and add to Horizon going forward and welcome community pull requests, suggestions, and feedback.

References

Adomavicius, G. and Tuzhilin, A. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge & Data Engineering*, (6):734–749, 2005.

Bellemare, M. G., Dabney, W., and Munos, R. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*, 2017.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Dudık, M., Langford, J., and Li, L. Doubly robust policy evaluation and learning. 2011.

Dulac-Arnold, G., Evans, R., van Hasselt, H., Sunehag, P., Lillicrap, T., Hunt, J., Mann, T., Weber, T., Degris, T., and Coppin, B. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

Exchange, O. N. N. Onnx github repository, 2018.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

Huang, T.-W. Tensorboardx. https://github.com/lanpa/tensorboardX, 2018.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678. ACM, 2014.

Jiang, N. and Li, L. Doubly robust off-policy value evaluation for reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML)*, volume Volume 48, pp. 652–661. JMLR. org, 2016.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

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

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529, 2015.

Paszke, A., Gross, S., Chintala, S., and Chanan, G. Pytorch, 2017.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. In *International Conference on Machine Learning*, pp. 1889–1897, 2015.

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

Sutton, R. S., Barto, A. G., et al. *Reinforcement learning: An introduction*. MIT press, 1998.

Thomas, P. and Brunskill, E. Data-efficient off-policy policy evaluation for reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML)*, pp. 2139–2148. JMLR. org, 2016.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, pp. 5. Phoenix, AZ, 2016.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.