Reproduction of Continual Backprop

Practical Work in AI 365.266

Pfleger Christoph
Supervisor: Thomas Schmied

May 5, 2023

1 Introduction

The paper "Continual Backprop: Stochastic Gradient Descent with Persistent Randomness"[1] demonstrates a gradual reduction in policy adaptability of Backprop[2]-trained reinforcement learning agents as their environment continues to change over time. It posits the gradual loss of randomness in the policy-weights, which enables efficient stochastic gradient descent learning [3, 4, 5], as the root cause. Accordingly, a novel generate-and-test process is proposed to continuously inject random features into the model, enabling long-term continuous learning in ever-changing environments.

2 Background

Many machine learning algorithms use Backprop to learn neural network weights in order to solve specific tasks, such as classification or agent control. However, while these methods achieve state-of-the-art performance in many problem domains, they struggle on non-stationary problems [6, 7, 8].

Non-stationarities can arise due to a number of factors, such as partial observability [9], other agents [10] or changing environments [11]. As these factors are common in the real world, the reinforcement learning and robotics fields would profit greatly from robust methods which can deal with non-stationarities.

While a perfect method for continuous learning would strife to remember past experience to improve performance in the future [12], the method described in the Continual Backprop paper only focuses on fast recovery after non-stationarities.

3 Method

The new process evaluates every hidden-unit of the model on it's contribution and adaptability. The contribution-utility describes how important the unit is to the model and is calculated by considering both the magnitude of outgoing weights and the magnitude of the unit's activation. The adaptation-utility is calculated as the inverse of ingoing weights, as adaptability decreases with weight magnitude due to an upper bound on one-step change in Adam-like optimizers [13]. The total utility of a hidden unit is then calculated as a running average with decay of the product between the contribution-utility and the adaptation-utility. Figure 1 shows this for a single feature of a network.

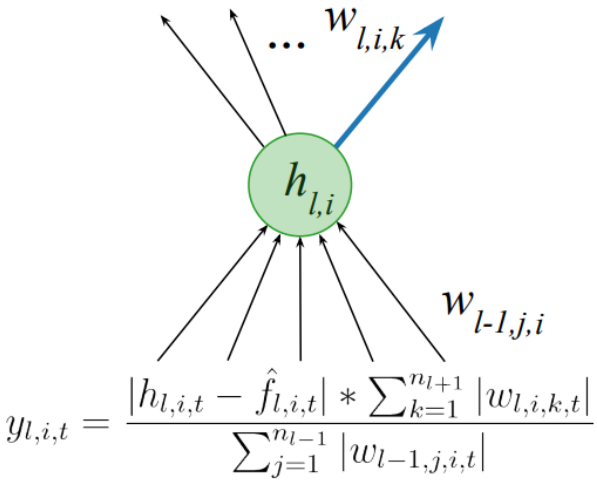

Figure 1: A feature/hidden-unit in a network. The utility of a feature at time t is the product of its contribution utility and its adaptation utility. Adaptation utility is the inverse of the sum of the magnitude of the incoming weights. And, contribution utility is the product of the magnitude of the outgoing weights and feature activation ($h_{l,i}$) minus its average ($\hat{f}_{l,i}$). $\hat{f}_{l,i}$ is a running average of $h_{l,i}$.[1]

This utility measure together with an age threshold is then used to reset low-performing units, therefore continuously injecting new randomness into the model.

4 Goal

One of the experiments in the paper uses Continual Backprop (CBP) with the Adam optimizer to train a PPO [14] agent (Continual PPO / CPPO) in a custom version of the Pybullet Ant environment [15], called Slippery Ant. This custom version changes the friction between the ant and the floor periodically, therefore introducing non-stationarities which the agent has to adapt to.

The performance of conventional PPO is compared to PPO with L2 regularization and

the new CPPO in the Slippery Ant environment on the average episodic performance over 30 random seeds. As can be seen in figure 2, PPO alone displays a clear degradation in performance as time goes on. Meanwhile, both PPO+L2 and CPPO remain adaptable, with CPPO narrowly being the best.

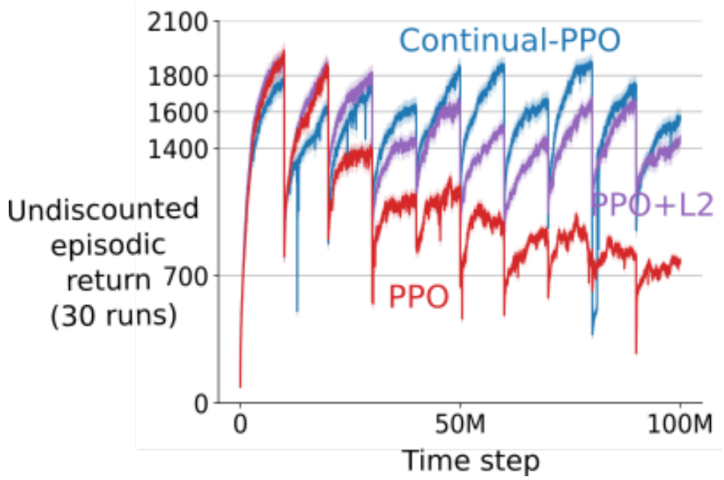

Figure 2: Figure from the CBP paper which is to be reproduced. Performance of PPO, PPO+L2 and CPPO on Slippery Ant.[1]

The goal of this project was to implement CPPO and to reproduce this plot without PPO+L2, showing clear degradation in PPO performance and the viability of CBP to counteract such degradation.

5 Setup

The following section describes all implementation details related to reproducing figure 2. The entire code base is available at on GitHub at "github.com/chripp/Continual-Backprop".

5.1 Environment

As stated in section 4, the environment is an adaptation of Pybullet's Ant environment. This new version changes the friction between the ant and the floor periodically, therefore introducing non-stationarities into the environment and challenging the agent to adapt continuously.

The Slippery Ant environment is a sub-class of the original Ant environment which overrides the step and reset functions. It includes a new counter for the number of steps taken and when the given limit is reached changes the friction to a random log-uniform number using a given seed.

Additionally, to prevent infinite episodes, another counter was implemented to reset the environment after some number of steps. This appears necessary, but there might be simpler and more elegant solutions available. As the CBP paper did not include a number for this limit the resulting plot of this project differs in the achieved rewards, although similar trends are still visible.

Finally, to make better use of available computation resources, multiple instances of the environment are used in parallel to gather data.

5.2 Optimizer

The following describes the implementation of CBP combined with the Adam optimizer as described in the paper.

5.2.1 Adam Adaptation

The Adam optimizer as implemented in PyTorch [16] internally tracks the age (number of updates done) of each weight it works on and uses that age to calculate the size of updates done to that weight. As it always operates on entire weight tensors it saves a scalar age per tensor, meaning all weights in that tensor are the same age.

Unfortunately, the reset mechanism of CBP+Adam as described in the paper requires resetting the age of individual weights in a tensor, such that they are treated as new weights in future updates.

To accommodate such resets a slightly altered Adam version was implemented, where age is saved in a tensor of the same shape as the tensor it belongs to. Further, as this change led to some performance concerns, the functions responsible for computations were declared as requiring PyTorch Just-In-Time (JIT) optimization. One of the benefits of this is that multiple sequential point operations like the ones introduced by changing to an age tensor can be combined into one-step operations, thereby massively speeding up computations.

5.2.2 Continual Backprop

As Continual Backprop is fundamentally an add-on to other optimizers, it has been implemented as a subclass of the custom Adam version described in section 5.2.1. At initialization it adds hooks to the model being trained which track relevant values for each layer in the feed forward network, such as the contribution-utility. When a weight update is initiated, it first calls Adam to update the weights before looping through the layers to calculate total utility scores and reset features.

Listing 1: The new CBP step function first performs an Adam step, then resets weights as necessary.

```
@torch.no_grad()
def step(self):
    super(CBP, self).step()
    for linears, output_linear in zip(self.linear_layers,
                                      self.output_linears):
                                      # cycle through models
        for current_linear, next_linear
                in zip(linears, linears[1:] + [output_linear]):
                # cycle through layers
            cbp_vals = self.cbp_vals[current_linear]
            pre_state = self.state[current_linear.weight]
            post_state = self.state[next_linear.weight]

            _step_calcs(cbp_vals, pre_state, post_state,
                        current_linear.weight, next_linear.weight,
                        self.eta, self.m, self.rho, self.eps)
```

Usually PyTorch optimizers only receive the weights of the model they are supposed to optimize, but as CBP requires information about activations in each layer and a sense of which weights go into a layer and which go out, the newly implemented optimizer requires additional access to the model. Although it might be possible to gain this information through the underlying computation graph which PyTorch constructs and updates in the background, this did not seem like a wise time investment. Instead, an ordered list of the models Linear layers has to be passed along, as well as a list of Activation layers belonging to those Linears, to which the above mentioned hooks are added. As with the Adam adaptation, PyTorch JIT optimization is used to speed up calculations.

Listing 2: Code for adding hooks to activation functions, which then automatically calculate important values as the model trains. Used in the Continual Backprop class.

```
def _add_hooks(self, linears, activations):
    assert len(linears) == len(activations)
    for lin, act in zip(linears, activations):
        act.register_forward_hook(self._hook_gen(lin))

def _hook_gen(self, linear_layer):
    num_units = linear_layer.weight.shape[0]
    self.cbp_vals[linear_layer] = {
        'age':  torch.zeros(num_units, dtype=int, device=self.dev),
        'h':    torch.zeros(num_units, device=self.dev),
        'f':    torch.zeros(num_units, device=self.dev),
        'fhat': torch.zeros(num_units, device=self.dev),
        'u':    torch.zeros(num_units, device=self.dev)
    }
```

```
def hook(mod, inp, out):
    if mod.training:
        cbp_vals = self.cbp_vals[linear_layer]
        with torch.no_grad():
            _hook_calcs(cbp_vals, out, self.eta)
return hook
```

Two major question about the CBP algorithm arose during the implementation of this optimizer.

For one, the paper only describes CBP for sequential input with gradient updates at each step. This is quite unlike the situation with PPO where training data is generated and then processed in batches. As there were no indications on how the authors implemented their version, it was decided to use the mean over a batch to calculate relevant CBP values (layer activations).

Another issue arose concerning the mechanism of choosing units to reset. The paper states that a fraction ρ of each layer's units is reset each update, after which those units are protected from resets for m updates. However, with the values of ρ and m stated in the paper, if the currently implemented algorithm used this approach it would cycle through all units in each layer to reset them before waiting some updates to do the same again. When no other units are of sufficient age then even extremely useful units are reset. This behavior does not seem to appear in the paper, so it might again be a difference in how batched data is handled. As a stand-in reset mechanism for the implemented CBP version, ρ is used as a probability of replacing the lowest performing unit.

5.3 RL Algorithm

Stable Baselines 3 (SB3) [17], a popular framework for training and testing reinforcement learning algorithms, provides an implementation for PPO. However, although it supports switching to various PyTorch optimizers, the additional requirements of the CBP+Adam optimizer discussed in section 5.2.2 prevent such a smooth replacement.

The solution was to create a subclass of SB3's ActorCriticPolicy implementation which overrides the default optimizer and provides the necessary additional model access. This new policy is then provided to the standard PPO implementation for training.

5.4 Logging

To ensure that evaluation metrics are calculated using an environment with the same friction as the training environment, a custom evaluation callback was implemented. This callback

updates the evaluation environment's friction to the current value whenever it is called and then calculates and logs relevant metrics. Further, it uses a separate process to run evaluation episodes, such that the evaluation does not hold up the training.

Another callback was implemented which periodically logs the age distribution of CPPO's weights. This information can then be used to verify that an appropriate number of resets occur over time.

The last callback is responsible for logging summary statistics about the model's weight magnitudes. As CBP aims to keep the randomness of weights similar to the initial state, one would expect it to slow or even prevent the increase in weight magnitudes compared to pure Adam. This can be checked using these summary statistics.

It was decided early on to use Weight & Biases (W&B) [18] for managing logs, enabling quick analysis of data using their web interface. However, while W&B offers a simple integration for SB3, this option did not work due to the multiprocessing discussed in section 5.5, as well as the folder structure implemented to separate local logs. Fortunately SB3 allows for custom Logger objects, so the solution was to create a custom W&B output format which is then added to the existing Logger. The W&B logs are available at "wandb.ai/hae_/CPPO".

5.5 Configuration Management

The Hydra framework [19] was used to manage configuration files, e.g. random seeds and CPPO hyper-parameters. It was also used for executing parallel runs using the Joblib backend, but due to conflicts with parallel environments this was later removed.

The CBP paper listed a total of 6 different configurations for the CPPO experiment and stated that only the results of the best performing configuration were plotted in the graph. Already early on it was deemed too time intensive to evaluate all 6 version for 30 random seeds each. Therefore, only one arbitrarily chosen configuration ($\rho = 10^{-3}$, $m = 500$) was run with 18 random seeds.

The runs for both PPO and CPPO were done with 6 instances in parallel. Due to the conflicts with parallel environments, these 6 instances had to be started manually.

6 Results

6.1 Main Result

After 12 days of runtime on a JKU server and some additional data processing to conform to the process described in the paper, e.g. binning, figure 3 was generated.

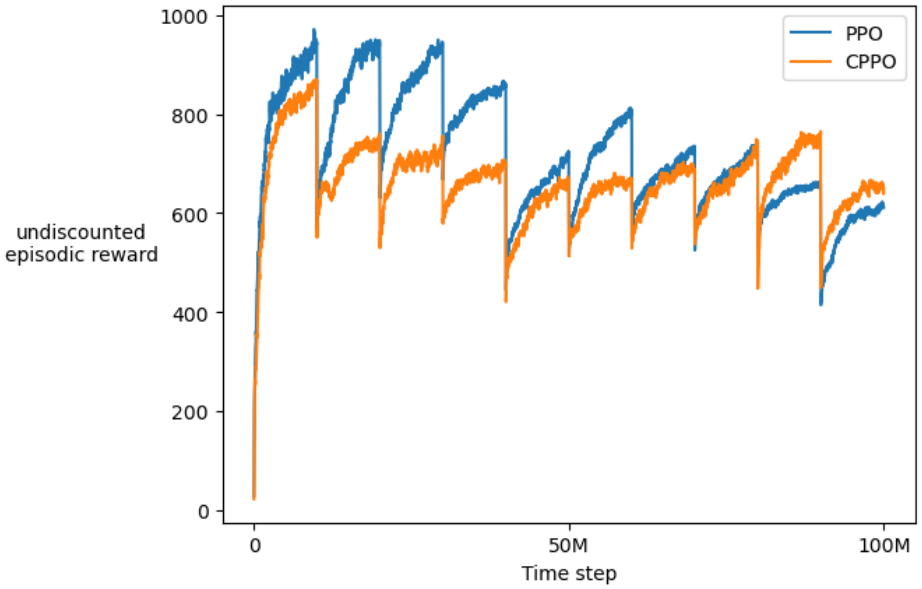

Figure 3: Performance of PPO and CPPO on Slippery Ant

Task 1, validating a decrease in PPO's performance over multiple non-stationarities and a long time, was successfully completed. The plot clearly shows a persistent downward trend.

Task 2, recreating the CPPO performance seen in the paper, was not successful. Although CPPO's performance does not seem to decline, it is outperformed by PPO at nearly every point. This discrepancy between paper and recreation could be due to a multiple reasons, such as using different CPPO configurations or the difference in reset mechanisms.

As mentioned in section 5.1, a mismatch between the original plot's and the recreation's y-axes can be observed. This is due to differing limits being used for preventing infinite episodes. Given that a limit of 500 steps was used for the new plot, it seems reasonable to assume that the paper used a limit of 1000.

6.2 Weight Magnitudes

In figure 4 the weight magnitudes over the course of training clearly show the effect of CBP's weight resets. While PPO's weights persistently increase in magnitude, CPPO's

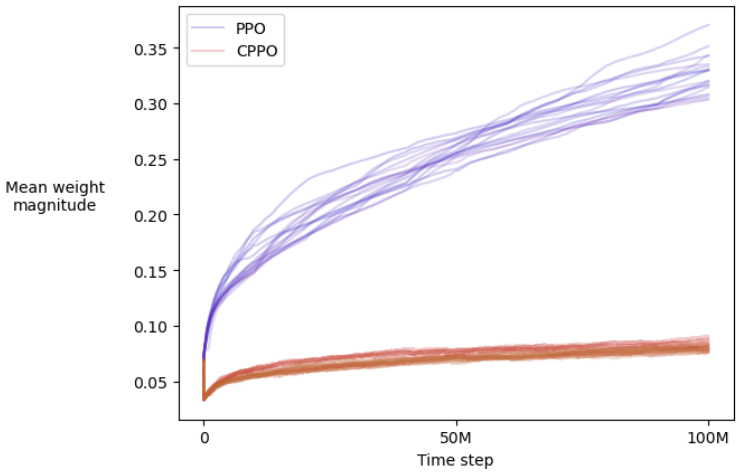

Figure 4: Overview of weight magnitudes over time. While PPO's weight magnitudes steadily increase, CPPO remains close to the starting conditions.

weight magnitudes stay close to the initialization magnitudes.

Considering the lacking performance observed in figure 3 and the stringent adherence to small weight magnitudes observed here, it may be valid to say that the chosen CPPO configuration was too aggressive in it's resets.

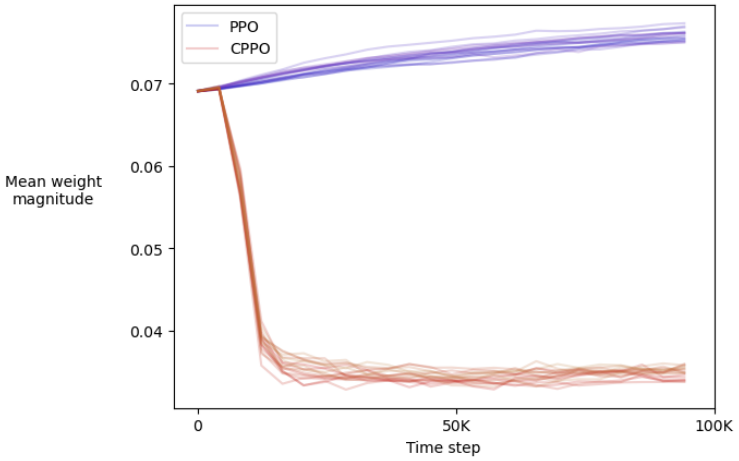

Figure 5: Closeup of weight magnitudes at the start of training. The drastic drop in CPPO's weight magnitudes may indicate a too aggressive reset strategy.

There is also a notable drop in weight magnitudes right at the beginning of training. Looking at a closeup in figure 5, CPPO closely follows PPO's progress until it's weight magnitudes suddenly drop. This is likely due to the maturity threshold protecting from resets at first, after which aggressive resets start. From the following age plot and manual inspection of weight distributions using W&B, it seems resets were much too likely, causing this decline, after which a subset of units started contributing enough to avoid resets almost indefinitely, thereby slowly increasing the weight magnitudes as with normal PPO.

6.3 Age Progression

To check if the reset mechanism works as intended, a plot was generated to present the age distribution of units over time. For better visibility this data is binned into 100 sections and colored using the logarithm of bin counts to bring out finer details. For better representation the bin counts were also averaged over all CPPO runs.

In figure 6 it is apparent that most units are either on the diagonal, meaning they are never reset, or right along the x-axis, meaning they are reset constantly. However, there are also rare cases of units becoming useful later, represented by diagonal lines starting at a later time step. This indicates that the idea of adding random features which the model can use to adapt has merit.

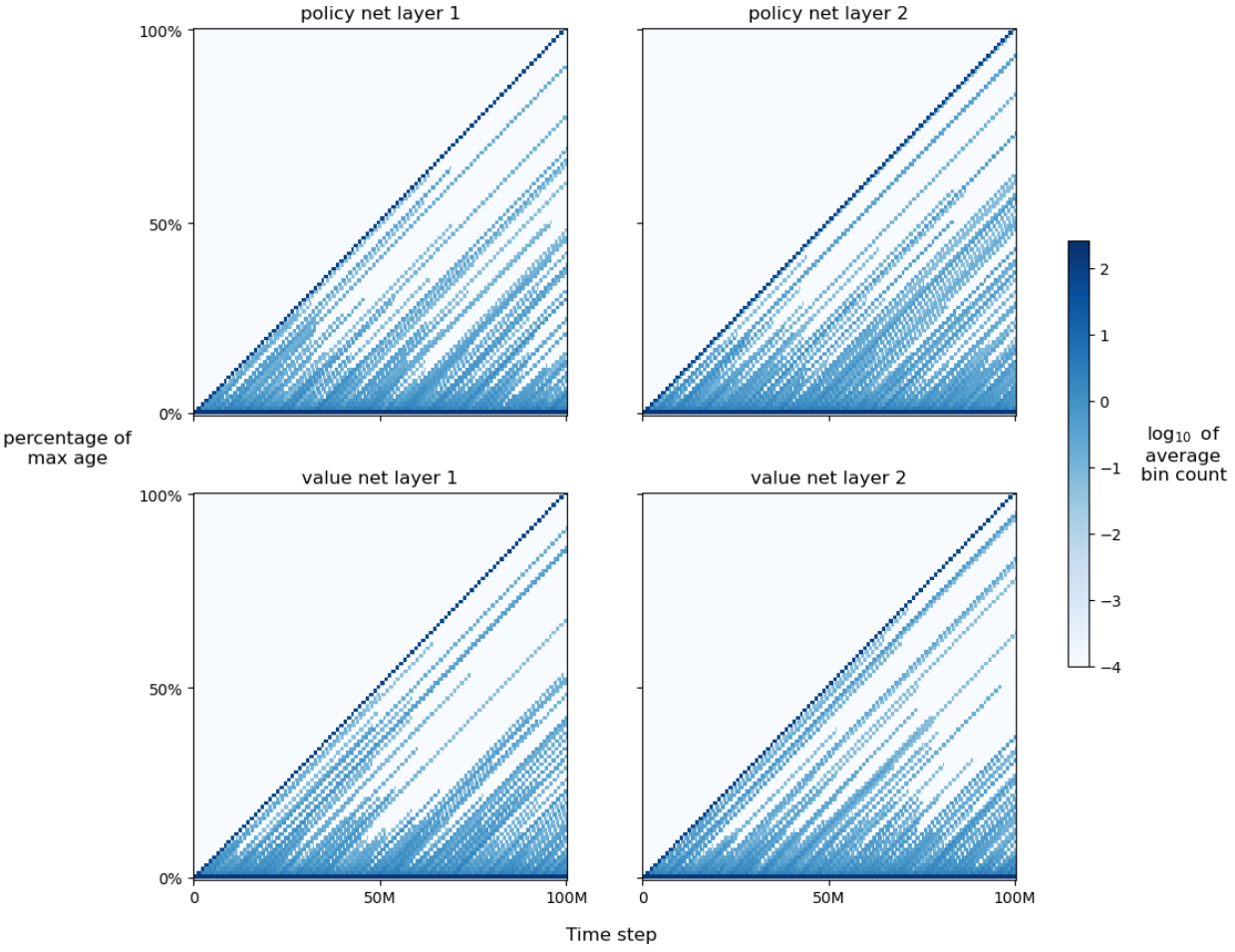

Figure 6: Overview of age progression in the different layers of the CPPO models. Most weights are either never reset or constantly reset, but some in-between is possible.

6.4 Influence of Friction

Already during the implementation of the Slippery Ant environment the question arose whether changing friction beyond a certain range actually creates a non-stationarity. For instance, going from very strong friction to even stronger friction may not force a change in behavior, as the agent is not slippery in either case. Similar reasoning could be applied to low friction. Furthermore, it seems unclear if agents could learn in environments where the friction is so low that their movements have almost no effect.

Figure 7 is an attempt to gather information concerning these questions. It plots the mean performance of an algorithm and the friction in the environment at the time. Furthermore, the datapoints are colored according to which non-stationarity in the training process they belong to. This is worth considering, as a declining performance is expected for instances close to the end of training.

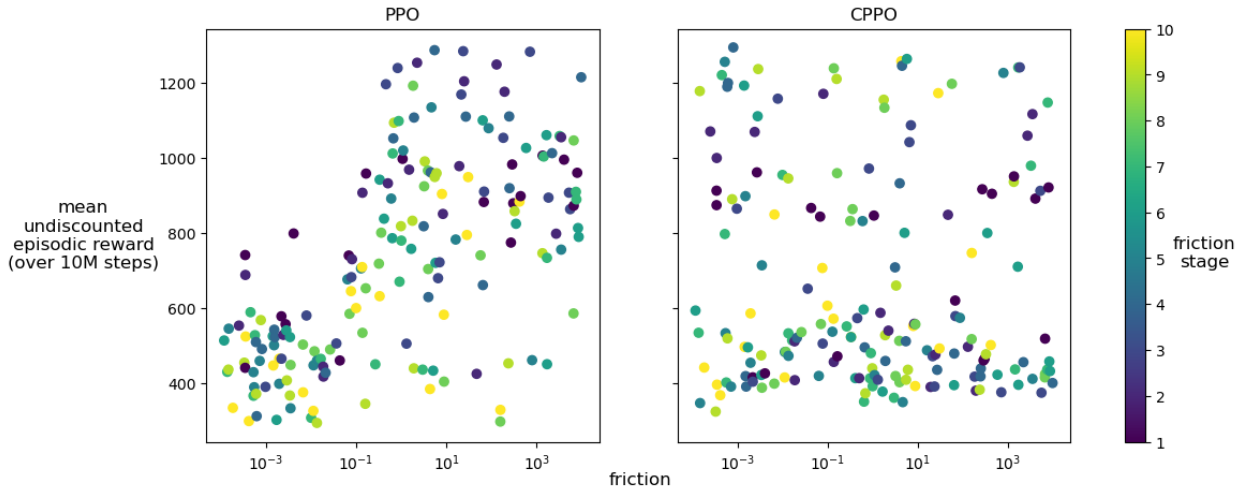

Figure 7: Influence of friction on performance. CPPO shows a surprising superiority for low friction.

For PPO there is a clear division between frictions below 10^{-1} and those above. It struggles in environments with too low friction but does well with a lot of friction.

CPPO on the other hand displays no preference in frictions. While it does worse than PPO for high friction, it does better with low friction.

It is not immediately apparent why there would be a difference on the low friction side. Especially for points early during training it would be expected that PPO has enough randomness from it's initialization to learn at least as well as CPPO. That this is not the case may indicate that CBP has advantages even in training tasks without non-stationarities.

7 Conclusion

Although the target figure could not be reproduced exactly, some valuable insights can be gained. For one, the performance degradation using standard Backprop is as described in the CBP paper, which again confirms the need for new optimization procedures which counteract this. Furthermore, while the CPPO performance could not be replicated, this shows that robust heuristics or adaptive mechanisms would substantially improve the practical applicability of the procedure. Lastly, CBP may also be advantageous for stationary learning tasks, as shown for low-friction Slippery Ant environments. In conclusion, CBP shows promise for both stationary and non-stationary environments, but still needs work to reduce or remove the need for hyperparameter search.

References

[1] Shibhansh Dohare, Richard S. Sutton, and A. Rupam Mahmood. *Continual Backprop: Stochastic Gradient Descent with Persistent Randomness*. arXiv:2108.06325 [cs]. May 2022. DOI: 10.48550/arXiv.2108.06325. URL: http://arxiv.org/abs/2108.06325 (visited on 11/15/2022).

[2] Henry J Kelley. "Gradient theory of optimal flight paths". In: *Ars Journal* 30.10 (1960), pp. 947–954.

[3] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". en. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. ISSN: 1938-7228. JMLR Workshop and Conference Proceedings, Mar. 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html (visited on 04/08/2023).

[4] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research. Issue: 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[5] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. arXiv:1502.01852 [cs]. Feb. 2015. DOI: 10.48550/arXiv.1502.01852. URL: http://arxiv.org/abs/1502.01852 (visited on 04/08/2023).

[6] Michael McCloskey and Neal J. Cohen. "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: ed. by Gordon H. Bower. Vol. 24. Psychology of Learning and Motivation. ISSN: 0079-7421. Academic Press, 1989, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: https://www.sciencedirect.com/science/article/pii/S0079742108605368.

[7] Robert M. French. "Catastrophic forgetting in connectionist networks". In: *Trends in Cognitive Sciences* 3.4 (1999), pp. 128–135. ISSN: 1364-6613. DOI: https://doi.org/10.1016/S1364-6613(99)01294-2. URL: https://www.sciencedirect.com/science/article/pii/S1364661399012942.

[8] Doyen Sahoo et al. *Online Deep Learning: Learning Deep Neural Networks on the Fly*. arXiv:1711.03705 [cs]. Nov. 2017. URL: http://arxiv.org/abs/1711.03705 (visited on 04/08/2023).

[9] Khimya Khetarpal et al. *Towards Continual Reinforcement Learning: A Review and Perspectives*. arXiv:2012.13490 [cs]. Nov. 2022. URL: http://arxiv.org/abs/2012.13490 (visited on 04/08/2023).

[10] Jakob Foerster et al. *Counterfactual Multi-Agent Policy Gradients*. arXiv:1705.08926 [cs]. Dec. 2017. URL: http://arxiv.org/abs/1705.08926 (visited on 04/08/2023).

[11] Sebastian Thrun. "Lifelong Learning Algorithms". In: *Learning to Learn*. Ed. by Sebastian Thrun and Lorien Pratt. Boston, MA: Springer US, 1998, pp. 181–209. ISBN: 978-1-4615-5529-2. DOI: 10.1007/978-1-4615-5529-2_8. URL: https://doi.org/10.1007/978-1-4615-5529-2_8.

[12] Raia Hadsell et al. "Embracing Change: Continual Learning in Deep Neural Networks". English. In: *Trends in Cognitive Sciences* 24.12 (Dec. 2020). Publisher: Elsevier, pp. 1028–1040. ISSN: 1364-6613, 1879-307X. DOI: 10.1016/j.tics.2020.09.004. URL: https://www.cell.com/trends/cognitive-sciences/abstract/S1364-6613(20)30219-9 (visited on 10/27/2022).

[13] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs]. Jan. 2017. URL: http://arxiv.org/abs/1412.6980 (visited on 04/08/2023).

[14] John Schulman et al. *Proximal Policy Optimization Algorithms*. arXiv:1707.06347 [cs]. Aug. 2017. URL: http://arxiv.org/abs/1707.06347 (visited on 04/08/2023).

[15] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. 2016. URL: http://pybullet.org.

[16] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[17] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[18] Lukas Biewald. *Experiment Tracking with Weights and Biases*. 2020. URL: https://www.wandb.com/.

[19] Omry Yadan. *Hydra - A framework for elegantly configuring complex applications*. 2019. URL: https://github.com/facebookresearch/hydra.