An Agent-Training Framework for Coping with Environments with Slow Simulators that have Fast Approximations

Matthew Anderson[*]
mga@berkeley.edu

Emaan Hariri[*]
ehariri@berkeley.edu

Sayan Paul[*]
sayanpaul@berkeley.edu

February 26, 2020

Abstract

Reinforcement Learning (RL) has been proposed as a method of rapidly solving complex design optimization problems, over a range of design specifications, without human intervention. Many of these complex design problems do not have closed-form equations based on the theory alone and instead require tedious and manual iteration over the design space to arrive at a solution. Furthermore, once derived, these solutions are specific to the given design specifications and require redoing the entire design effort when the requirements change—as they often do. This constant design effort makes solving complex design optimization problems, such as integrated circuit design, incredibly expensive and time consuming. In contrast, once trained, an RL agent could quickly produce solutions to families of design problems that would traditionally require hours to solve. RL is particularly attractive for this class of problems because labelled data is scarce and it is normally easy to generate a reward signal using physics-based simulators that evaluate a designs performance.

Despite this opportunity, RL techniques have generally not seen wide-scale adoption in this space. One of the primary challenges is that it requires a tremendous amount of time to generate samples for both training and evaluation of RL agents in this context. For the class of problems considered here, evaluating the performance (or reward) for each sample using accurate simulations can take hours or even days. However, it is often true that simpler, less-accurate simulations exist which can also be used to evaluate performance much more quickly. These faster simulators are often too inaccurate to be used exclusively but they have shown some value as a baseline for transfer learning. This has been successful at reducing the required training time, as well as the time it takes an agent to perform actions, at the expense of both performance and generality.

Motivated by this, we propose an algorithm that utilizes a faster, simplified simulator for both training and evaluation to realize potentially significant time savings in both respects. Our proposed algorithm is able to leverage the fast simulator for both training and evaluation through the use of a deep neural network (DNN), i.e. an *offset predictor*, that approximates the difference between the fast and slow simulators given the state of the agent. This approach immediately reduces the time it takes an agent to act by eliminating the slow simulator when not training this offset predictor; however, this comes at the expense of having to evaluate both the fast and slow simulators to generate labelled data during the training phase.

We combat this by developing a general strategy to evaluate the slow environment in parallel. We explore parallelization in two main contexts: one that closely matches the general setting of design optimization problems and one that makes few assumptions about the environment itself. We examine the potential speedups one might realize in both settings and through experimentation we demonstrate that significant speedup is attainable.

The proposed algorithm is evaluated in a modified two-dimensional `PointMass` environment that resembles a generic design optimization problem. This environment allowed us to visualize the agent's trajectories, the reward space, and evaluate general performance. Our results show a significant improvement in the evaluated average returns for our proposed agent when compared to a one-shot transfer learning agent as well as significant benefits in evaluation time. We also find that our parallelization techniques yield a $\sim$2x speedup in the training time of our algorithm.

[*]The authors contributed equally.

1 Introduction

A common paradigm in reinforcement learning (RL) is to have agents physically interact with environments, e.g. a robot arm moving physically in space. However, it can be both time-consuming and costly to set up and train in these environments, which has prompted the development of simulations designed to circumvent the need for physical interaction. The trade-off between physical- and simulation-based interaction with an environment in RL is further explored in [6].

The models developed to simulate environments can vary greatly in complexity, and in many cases highly-accurate models can be designed at a high computational cost. Alternatively, we are able to use less-accurate models that provide rougher approximations of the actual model but can in many cases be orders of magnitude faster. This paradigm is frequently visible in physics simulations, where accurately modeling particular environments can be extremely computationally expensive with runtime potentially exponential in the degrees of freedom . Here, we will explore the trade-off between *simulations* of varying accuracy. We take advantage of the ability for deep neural networks (DNN) to greatly aid in the computation of traditionally-complex simulation problems, such as was recently done with the n-body problem which is known to be in PSPACE-Hard [3].

Our motivating case study is in the field of analog circuit design, which has previously been explored in the context of deep RL as in [9]. Specifically, we are concerned with the effects of parasitic extraction (PEX): determining parasitic effects (e.g. resistance and capacitance) from circuit features, such as wiring, in cases where these effects are non-negligible. Calculating parasitics from circuitry as well as running circuit calculations post-PEX is potentially very computationally expensive, since the post-PEX circuit could have hundreds to thousands of more components than schematic layouts. Successful efforts have been made to apply DNNs with combinatorial optimization techniques to greatly speed up circuit design by selectively choosing more performant simulations, and thus increasing sample efficiency [4].

Our efforts now turn from increasing sample efficiency to gradually removing the need for a *slow* simulator (i.e. post-PEX simulation) altogether in the evaluation phase of our algorithm. Here, our approach is to modify the standard Actor-Critic algorithm such that our new algorithm applies broadly to applications where there is a some access to an accurate but expensive simulator as well as an inaccurate but cheap simulator. We refer broadly to this expensive simulator as the *slow simulator* and this cheap simulator as the *fast simulator*.

If we model the slow and fast simulator as two separate environments, one where the slow simulator models our original physical model, we see that our problem is one of supervised domain adaptation between some *slow distribution* and *fast distribution* [2]. We can also view our slow and fast distributions as derived from some latent distribution provided by our physical model. However, we assume that the slow simulator faithfully models our physical model, we take the slow distribution to be our latent distribution. So, our target distribution is based off of our slow simulator, while our source distribution is provided by our fast simulator, which by its nature is more readily sampled. In the ideal circumstances where the fast simulator approximates the slow simulator, the source distribution should be a perturbation of the target distribution. In this sense, we attempt to solve a transfer learning problem between two environment where one is an approximation of the other.

Our approach attempts to solve this problem via an *o set predictor* modeled as a DNN which determines the offset between what can be thought of as the slow an fast environments, derived from the slow and fast simulators respectively. We propose an algorithm which applies this approach with parallel calls to our simulator to considerably decrease training time. In evaluation (and in principle late training), our algorithm uses the fast simulator in conjunction with the trained offset predictor to provide a model which can accurately and quickly emulate slow environment.

2 Test Environment

One of our main focuses when constructing the testing environment for our algorithm was to avoid limiting ourselves to the space of analog circuit design and instead to the broader space of design optimization. In other words, we wanted to solve problems characterized by incremental hand-tuning of a set of parameters to achieve some performance requirement and design specification. With the exception of a relatively small set of “good” designs, the majority of the design space achieves relatively poor performance, leading to performance landscapes that are somewhat underspecified or pseudo-sparse: characterized by low values everywhere except for a few sharp peaks.

2.1 Point Mass

Our primary testing environments were based on the point mass (`PointMass`) environment, in which some 2D point in space attempts to reach a predefined target. Given some trajectory in this space, the associated reward is calculated as the negative sum, over time, of the distances of the point to the target. `PointMass` provides a flexible environment for manipulation since the environment allows for fine-tuned manipulation of the reward space. In the context of our experiments, `PointMass` is primarily used to show that direct transfer learning, or domain adaptation, by directly using the fast simulator with an offset predictor, performs poorly. We refer to the reward map within the `PointMass` environment as $R : \mathbb{R}^2 \to \mathbb{R}$. For the majority of our findings, we use various modifications of `PointMass`:

- **Reward-Offset Point Mass:** The reward-offset point mass (`RewardOffsetPointMass`) environment modifies `PointMass` by shifting the reward space by a certain prespecified $o\ \ set$ variable, v, which is passed in as a 2D vector. Our resulting reward map is $R_O(x) = R(x - v)$ (clipped to the appropriate domain).

- **Sparse Point Mass:** The sparse point mass environment (`SparsePointMass`) predetermines a map on R, the reward map, given a threshold $p \quad [0, 1]$ and a hash $h : \mathbb{R}^2 \to [0, 1]$. We then apply h to the reward map and set all values below the threshold to 0. The hash is used to maintain the space when the environment is refreshed. The new reward map is $R_S(x) = R(x)\mathbb{1}\{h(R(x)\}$. As expected, we find that the `SparsePointMass` environment produces more unstable learning curves as well as a longer training time.

- **Pseudo-Sparsified Point Mass:** The pseudo-sparsified point mass environment (`PseudoSparse-PointMass`) is similar to `SparsePointMass`, but also takes a convex combination parameter λ. We return a convex combination of the original reward map with the fully sparsified map, hence making the map only pseudo-sparse. The reward map is $R_P(x) = \lambda \cdot R_S(x) + (1 - \lambda) \cdot R(x)$. The `PseudoSparsePointMass` environment is somewhat underspecified and resembles our desired performance landscape more closely than `SparsePointMass` does.

- **Noisy Point Mass:** The noisy point mass (NoisePointMass) introduces some fixed Gaussian noise $\varepsilon(x) \quad N(0, \sigma^2)$ determined at each x to the reward space of `PointMass`. The new reward space becomes $R_N(x) = R(x) + \varepsilon(x)$.

The modified environments of `PointMass`, as described above, serve to simulate our fast environment, while our slow environment is vanilla `PointMass` itself. We find that this adequately reproduces our motivating example of post-PEX simulation, as well as various other design optimization problems we are concerned with.

3 Proposed Architecture

We primarily focused on making our architecture flexible in the specific RL algorithm type used, but we ran experiments for this report using the Actor-Critic algorithm. The main contributions of our architecture are two-fold to address the salient issues in using a fast simulator. First, we must find a way to account for the loss in precision when using a fast simulator. Second, we must find a way to reduce the slow training time associated with calling the slow environment.

3.1 Offset Predictor

Although there may be domain-specific ways to describe and correct for this difference, we decided to use a DNN as a general function approximator for the difference between the two which we call an $o\ \ set$ *predictor*. See figure 1 for a rough visualization of this training loop.

With this, we see that we have three sets of observations that are generated by our training loop: o_s from the slow environment, o_f from the fast environment, and $\hat{o}_s := o_f + \mathrm{Offset}_\varphi(o_f)$ which approximates o_s. We use $\hat{o}_s$ to train the policy π_θ and we simultaneously train the offset predictor using o_s and o_f. Note that we use the o to signify an observation here, but the input to the offset predictor could be the concatenation of an arbitrary set of values from the fast environment. For example, in the `RewardOffsetPointMass` environment, we define the observation to be the concatenation between the state and the exact reward value (in the slow environment) or an offset reward value (in the fast environment). We can translate the above training loop into more-concrete pseudocode (cf. algorithm 1).

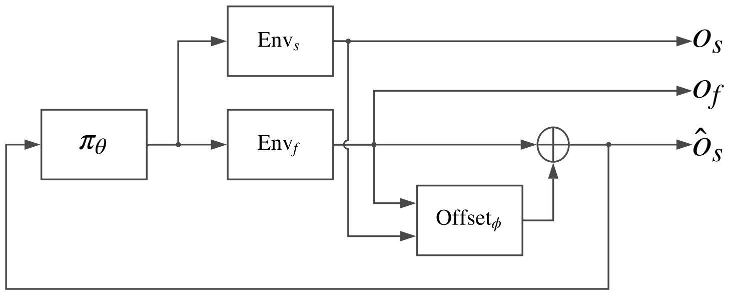

Figure 1: Training loop with an offset predictor.

Algorithm 1 Training Loop with Offset Predictor

```
Require: env$_s$: Slow environment; env$_f$: Fast environment; $\pi_\theta$: Policy; off$_\varphi$: Offset predictor; $|\tau|_{\max}$: Maximum trajectory length.
  function CollectTrajectory(env$_s$, env$_f$, $\pi_\theta$, off$_\varphi$, $|\tau|_{\max}$)
      Collect a trajectory $\tau_f := \{(o_{f,t}, a_{f,t}, r_{f,t}, o_{f,t})\}_{t=1}^{|\tau|_{\max}}$ using $\pi_\theta$ and env$_f$.
      Use $\{a_{f,t}\}_{t=0}^{|\tau|_{\max}}$ to generate a trajectory $\tau_s := \{(o_{s,t}, a_{f,t}, r_{s,t}, o_{s,t})\}_{t=0}^{|\tau|_{\max}}$ with env$_s$.
      Calculate our approximation of $\tau_s$, $\hat{\tau}_s := \{(o_{f,t} + \text{off}_\varphi(o_{f,t}), a_{f,t}, r_{f,t}, o_{f,t})\}_{t=1}^{|\tau|_{\max}}$.
      return $\tau_f$, $\tau_s$, $\hat{\tau}_s$
  end function

Require: $T$: The number of timesteps to train for; $b_{\text{des}}$: The desired batch size in timesteps
  Initialize weights $\theta$, $\varphi$.
  Initialize $t \leftarrow 0$.
  while $t < T$ do
      Build batch $\{\tau_{f,i}, \tau_{s,i}, \hat{\tau}_{s,i}\}_{i=1}^{k}$ by calling CollectTrajectory $k$ times, where $k$ is
          the number of calls it takes to reach $b_{\text{des}}$ timesteps in the batch.
      Update policy parameters $\theta$ using $\{\hat{\tau}_{s,i}\}_{i=1}^{k}$.
      Update offset predictor parameters $\varphi$ using $\{\tau_{f,i}, \tau_{s,i}\}_{i=1}^{k}$.
      $t \leftarrow t +$ # of timesteps in batch.
  end while
```

3.2 Limitations of the Offset Predictor

One thing we notice about this training loop is that we need to evaluate the slow environment on every single timestep, eliminating the benefit of having a fast simulator. In fact, this new algorithm makes training *slower*; not only do we have to evaluate both the fast and slow environments at every timestep in the training loop (instead of just the slow), we also have to train an extra neural net, which may require more training iterations overall.

We can roughly formalize this as follows. Let T be the number of timesteps we need to collect to train the policy to meet some accuracy requirement, t_s to be the time it takes to poll the slow environment once, and t_f be the time for the fast environment. In the original paradigm, where we just used the slow environment in our training loop, it took $T \cdot t_s$ time to train the policy, but now it takes $T \cdot (t_s + t_f)$ time.

We still achieve a speedup during the evaluation phase, however, since we now only need to evaluate the fast simulator and the offset predictor at every timestep. The question now is whether we can improve the training time as well.

3.3 Parallelizing Slow Environment Calls

To fix the slowdown in training time with the offset predictor, we introduce the second contribution of our proposed architecture: parallelizing the calls to the slow environment. We can approach this in multiple ways. Let's assume that we have a situation where the next state is simply a function of a state and an action (i.e. when the setting is Markovian) and the fast environment accurately keeps track of the state of the agent at each timestep. In other words, the fast environment is inaccurate in some other way (e.g. reward) and the offset predictor approximates this difference. This seems like a very specific setup, but in the space of design optimization, where the *state* of a design is defined by a set of parameters that one directly tunes, these assumptions hold. The core difficulty in this setting is measuring the performance of a design. In this case, we can spawn parallel calls to the slow environment to get the reward value for each timestep and we pass in $(s_{f,i}, a_{f,i})$ as arguments. Then, whenever we wish to update the offset predictor, we can join these parallel calls to get the correct reward values. We call this *sample-level parallelism*.

If the previous assumptions don't hold, we can still realize some benefits from parallelism. For each trajectory that we collect from the fast environment, we take the sequence of actions and fork a new parallel call to roll out a trajectory in the slow environment with these actions. We can join these parallel calls to get the batch of tuples to train the offset predictor with, and we can realize a parallelism factor up to the number of trajectories in a batch. We call this *trajectory-level parallelism*.

See figure 2 for an outline of the training loop in both cases.

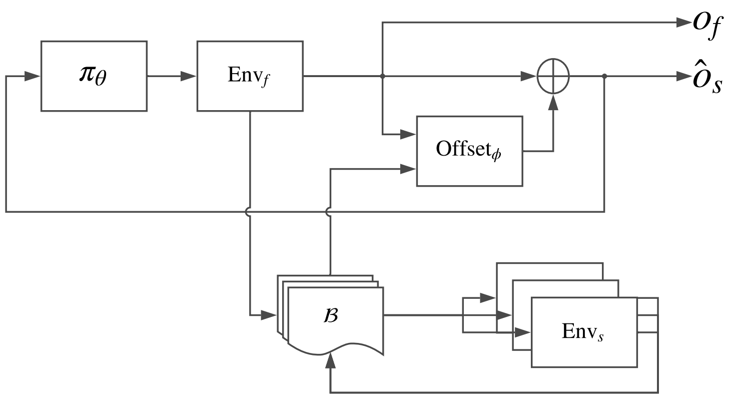

Figure 2: Training loop with parallel calls to the slow environment.

3.4 Estimating the Training Time

We have proposed two different ways to parallelize, and we now further examine the theoretical training time in each case.

3.4.1 Sample-Level Parallelism

We call the slow environment in parallel in order to construct a batch so it is sufficient to discuss the time it takes to process a batch during the training phase. Let's assume that there are n threads available on our system in addition to the main thread. We recall that b_{des} refers to the desired number of timesteps in a batch, t_s refers to the time it takes to call the slow environment for one timestep, and t_f is the time for the fast environment (so $t_f < t_s$). To further simplify our analysis, we assume that there is no overhead to spawning and joining parallel processes and that we are able to spawn all the threads at once when we just begin to collect trajectories for a batch. This represents the very best case scenario for our parallel implementation.

We have two possible execution times for the parallel calls in this scenario. We recall that if $b_{\text{des}} \leq n$, then it takes t_s time to finish the parallel calls since there is only one timestep evaluated on each parallel thread. However, if $b_{\text{des}} > n$, then we split up the timesteps evenly across the threads so each thread evaluates b_{des}/n timesteps, so it takes $b_{\text{des}} \quad t_s/n$. We are also running the fast environment for each environment in the main thread, so the overall runtime for one batch is

$$t_{\text{batch}} = \begin{cases} \max(t_s, t_f \cdot b_{\text{des}}) & b_{\text{des}} \leq n \\ \max\left(\frac{b_{\text{des}}}{n} \cdot t_s, t_f \cdot b_{\text{des}}\right) & b_{\text{des}} > n \end{cases}$$

3.4.2 Trajectory-Level Parallelism

We recall that $|\tau|_{\max}$ refers to the maximum length of a trajectory returned by the CollectTrajectory function. For simplicity, we assume that the trajectories are all this length, so there are $b_{\text{des}}/|\tau|_{\max}$ trajectories that we collect per batch. We evaluate the action sequences from these trajectories in parallel and it takes $|\tau|_{\max} \quad t_s$ time to call the slow environment for each trajectory.

Again, we have the same two possible execution times for the parallel calls in this scenario. The number of parallel calls we spawn in this case is $b_{\text{des}}/|\tau|_{\max}$ instead of b_{des}, so the time for one batch is

$$\begin{aligned} t_{\text{batch}} &= \begin{cases} \max(|\tau|_{\max} \quad t_s, t_f \cdot b_{\text{des}}) & \frac{b_{\text{des}}}{|\tau|_{\max}} \leq n \\ \max\left(\frac{b_{\text{des}}}{n \cdot |\tau|_{\max}} \cdot |\tau|_{\max} \quad t_s, t_f \cdot b_{\text{des}}\right) & \frac{b_{\text{des}}}{|\tau|_{\max}} > n \end{cases} \\ &= \begin{cases} \max(|\tau|_{\max} \quad t_s, t_f \cdot b_{\text{des}}) & \frac{b_{\text{des}}}{|\tau|_{\max}} \leq n \\ \max\left(\frac{b_{\text{des}}}{n} \quad t_s, t_f \cdot b_{\text{des}}\right) & \frac{b_{\text{des}}}{|\tau|_{\max}} > n \end{cases} \end{aligned}$$

3.4.3 Comparing Training Times

There are three ranges of n, the number of threads we have in our system, for which we need to compare the runtime of sample-level and trajectory-level parallelism.

- $n < b_{\text{des}}/|\tau|_{\max}$: The two training times are identical.
- $b_{\text{des}}/|\tau|_{\max} \leq n < b_{\text{des}}$: The training times differ based on the relationship between $|\tau|_{\max} \cdot t_s$ and $(b_{\text{des}}/n) \cdot t_s$; trajectory-level parallelism is faster if $|\tau|_{\max} < b_{\text{des}}/n$ and it is slower otherwise.
- $n \geq b_{\text{des}}$: Sample-level parallelism is faster here since $|\tau|_{\max} \geq 1$.

In our setting, we have ignored the real-world overhead of making several parallel calls, which would have impacted the sample-level parallelism more than the trajectory-level parallelism. Nevertheless, this provides a rough approximation of the speedup benefits of parallelizing calls to the slow environment.

3.5 Parallelization Implementation

In our experiments, we ran with $b_{\text{des}} = 1000$, $|\tau|_{\max} = 100$, and on machines with $n = 8$ threads, so we decided to focus on a trajectory-level parallelization to better-utilize the limited hardware threads available on our machines. This also is a natural modification to the training loop with the offset predictor (cf. algorithm 2, with major changes from algorithm 1 highlighted in blue). We use the Ray library [7] to execute multiple calls to the slow environment and to eventually join the results.

4 Results

We devise two reference experiments to serve as points of comparison for the performance of our proposed algorithm which uses an offset predictor (cf. algorithm 1). For the first, we used a standard A2C agent in conjunction with the slow simulator for both training and evaluation. This approach would take the

Algorithm 2 Training Loop with Offset Predictor and Parallel Calls to Slow Environment

```
Require: $\text{env}_s$: Slow environment; $\text{env}_f$: Fast environment; $\pi_\theta$: Policy; $\text{off}_\varphi$: Offset predictor; $|\tau|_{\max}$:
  Maximum trajectory length.
  function CollectTrajectory($\text{env}_s$, $\text{env}_f$, $\pi_\theta$, $\text{off}_\varphi$, $|\tau|_{\max}$)
      Collect a trajectory $\tau_f := \{(o_{f,t}, a_{f,t}, r_{f,t}, o_{f,t})\}_{t=1}^{|\tau|_{\max}}$ using $\pi_\theta$ and $\text{env}_f$.
      Spawn a process that uses $\{a_{f,t}\}_{t=0}^{|\tau|_{\max}}$ to generate a trajectory
          $\tau_s := \{(o_{s,t}, a_{f,t}, r_{s,t}, o_{s,t})\}_{t=0}^{|\tau|_{\max}}$ with $\text{env}_s$.
      Calculate our approximation of $\tau_s$, $\hat{\tau}_s := \{(o_{f,t} + \text{off}_\varphi(o_{f,t}), a_{f,t}, r_{f,t}, o_{f,t})\}_{t=1}^{|\tau|_{\max}}$.
      return $\tau_f$, process ids for $\tau_s$, $\hat{\tau}_s$
  end function

Require: $T$: The number of timesteps to train for; $b_{\text{des}}$: The desired batch size in timesteps
  Initialize weights $\theta$, $\varphi$.
  Initialize $t \leftarrow 0$.
  while $t < T$ do
      Build batch $\{\tau_{f,i}$, process ids for $\tau_{s,i}$, $\hat{\tau}_{s,i}\}_{i=1}^{k}$ by calling CollectTrajectory $k$ times,
          where $k$ is the number of calls it takes to reach $b_{\text{des}}$ timesteps in the batch.
      Collect $\{\tau_{s,i}\}_{i=0}^{k}$ from spawned processes.
      Update policy parameters $\theta$ using $\{\hat{\tau}_{s,i}\}_{i=1}^{k}$.
      Update offset predictor parameters $\varphi$ using $\{\tau_{f,i}, \tau_{s,i}\}_{i=1}^{k}$.
      $t \leftarrow t + \#$ of timesteps in batch.
  end while
```

longest cumulative time in training and evaluation since it requires the slow simulator in both phases, but it should achieve the best possible performance (i.e. average return). We will refer to this as the *baseline* experiment.

For the second, we utilize transfer learning to reduce the training time, similar to [9]. This reduces training time by using the fast simulator during training and the slow simulator during evaluation. We recall that transfer learning expects that there is some similarity between the reward spaces of the fast and slow simulator. This will allow an agent trained on one to perform well on the other. We call this the *transfer learning* experiment.

The performance of our proposed architecture is compared to the baseline and transfer learning experiments to determine its impact on agent performance (i.e. average evaluated returns), training time, and evaluation time.

4.1 Comparing Agent Performance

We visualize the performance of our agents in the 2D `PointMass` environment by examining their trained trajectories to gain an understanding of the potential strengths and weaknesses of each approach. In figure 3, we show the near-optimal agent trajectory of the baseline agent. The agent navigates from some random starting spot to the target location (signified by a green dot) following an almost-direct path. Note that the concentric contours around the target location represent regions of equivalent reward, with warmer colors representing higher reward and cooler colors representing lower.

In figure 4, we show the sub-optimal trajectory of the transfer learning agent. The agent navigates from the random starting spots to a region that is offset from the target location. This offset location is consistent with the high-reward locations from the `RewardOffsetPointMass` environment and highlights the limitations of using transfer learning. In situations where the differences in the reward spaces are large, one-shot transfer learning can yield poor results when given accurate data in evaluation.

In figure 5, we again see near-optimal agent trajectories, this time from our proposed agent using the fast simulator (`RewardOffsetPointMass` environment) and an offset predictor. Note the difference in the reward space in this figure compared to figure 3 and 4. Despite the difference in the reward space, the proposed agent navigates almost directly to the true target location. This is because the offset predictor is able to compensate for the differences in the reward, allowing our proposed agent to learn more optimal behaviours. The performance of the offset predictor is shown in figure 6.

Comparing the learning curves for all three cases (the baseline, the transfer learning, and our proposed agent), as shown in figure 7, we see that the baseline exhibits the best performance, followed by our

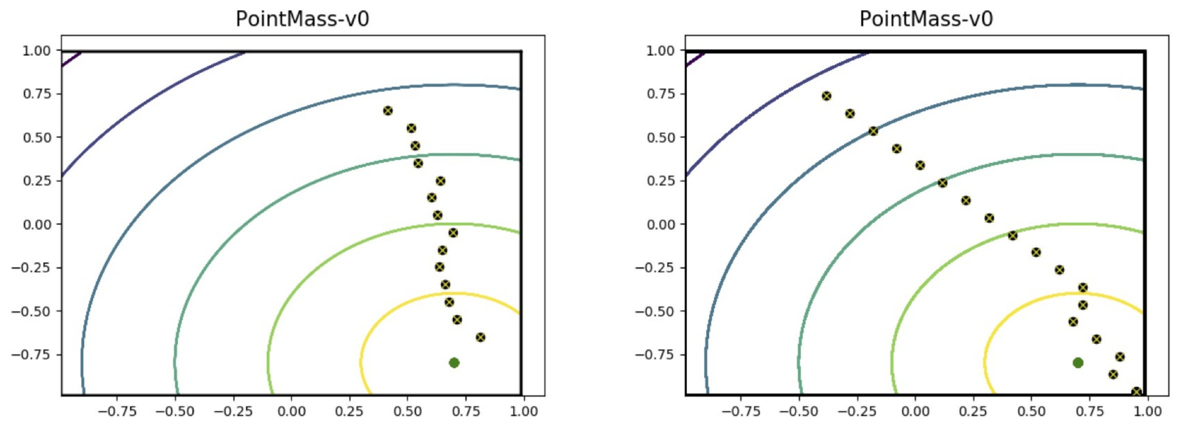

Figure 3: Visualization of two randomly-selected trajectories through the PointMass environment (slow simulator) by the baseline agent. These were trajectories were captured after 300 iterations of training.

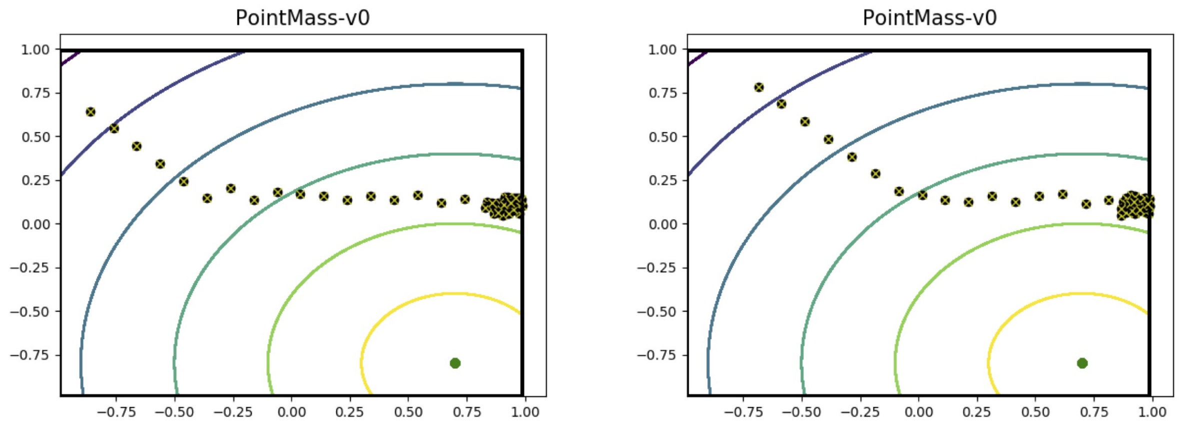

Figure 4: Visualization of two randomly-selected trajectories through the PointMass environment (slow simulator) by the transfer learning agent. These were trajectories were captured after 300 iterations of training. Note: This agent was trained on the `RewardOffsetPointMass` environment (or fast simulator).

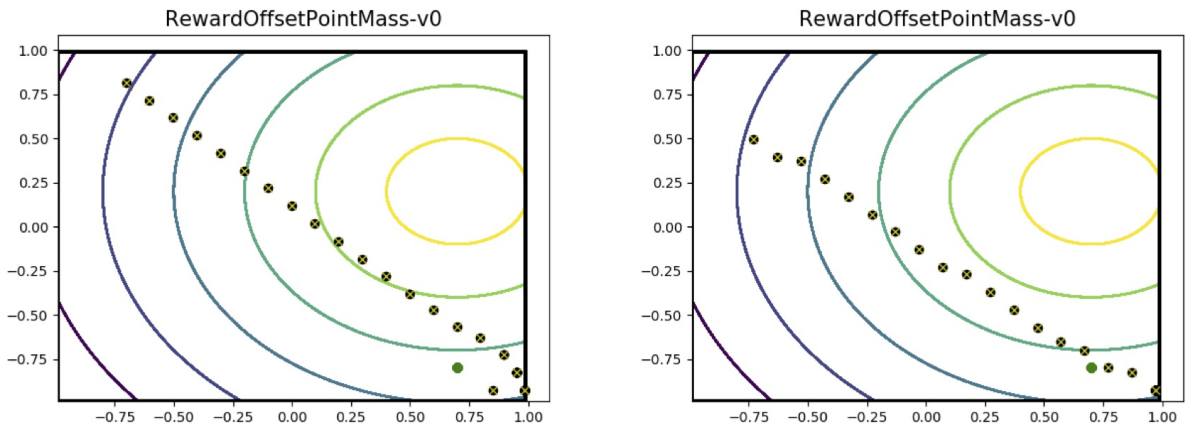

Figure 5: Visualization of two randomly-selected trajectories through the `RewardOffsetPointMass` environment (fast simulator) by our proposed agent, using the offset predictor. These were trajectories were captured after 300 iterations of training. Note: This agent was also trained on the `RewardOffsetPointMass` environment (or fast simulator).

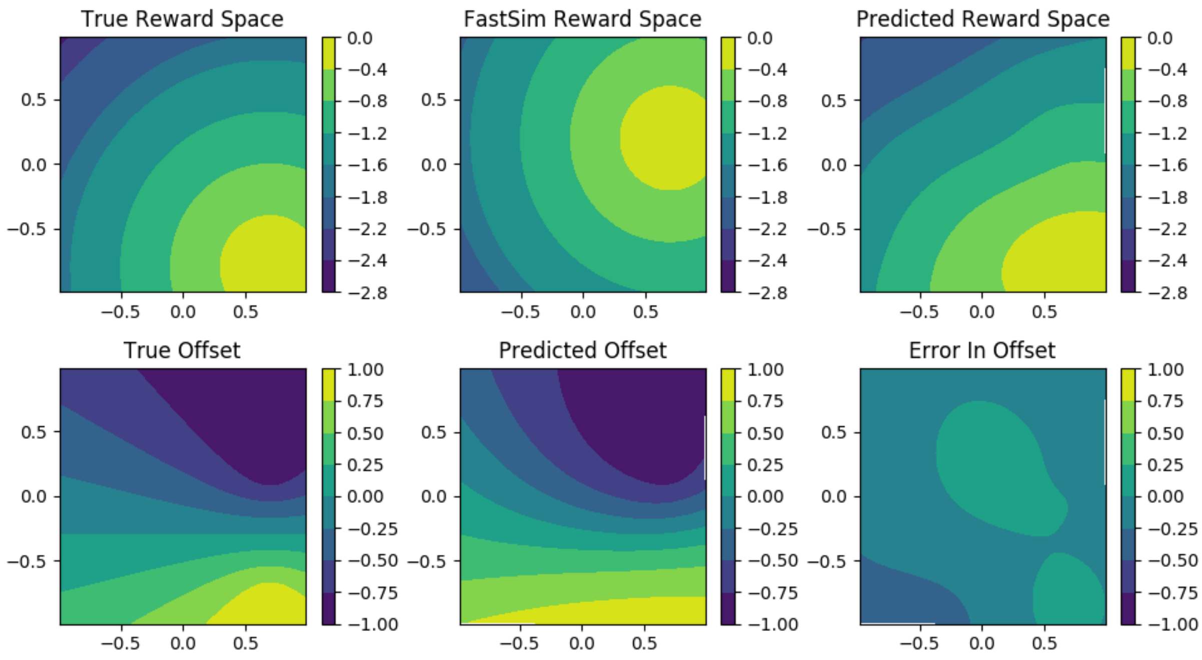

Figure 6: Visualization of offest predictor output after 300 training iterations.

proposed agent, and then the transfer learning agent. Using our proposed approach we have sacrificed some performance for the ability to save significantly on evaluation and (possibly) training time. The amount that the performance degrades when using the proposed agent depends on the error of the offset predictor in the regions of the design space that our agent traverses.

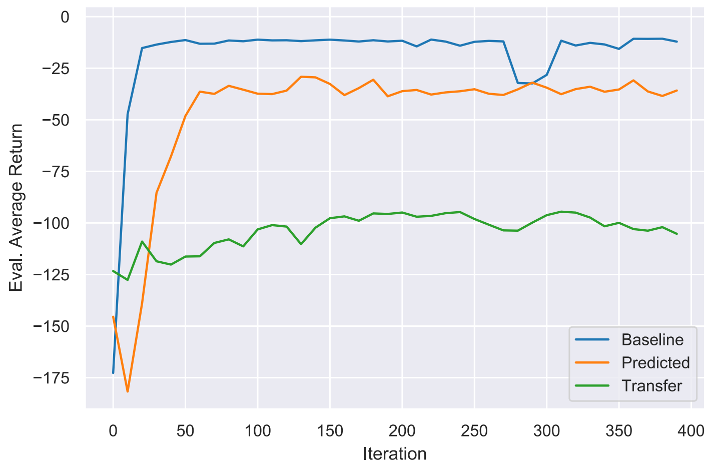

Figure 7: Comparison of learning curves for the baseline, transfer learning, and proposed agents.

4.2 Comparing Training And Evaluation Time

We have another metric of success of our algorithm, which is the combined training and evaluation time of both the agent and offset predictor. We are particularly interested in the increased savings in agent training time of algorithm 2 over algorithm 1. We see in figure 8 that the parallel algorithm runs over 2.2x the speed (40-45% of the runtime) of the serial algorithm. From figure 9 we observe that, as expected, running the slow simulator in parallel does not impact the agent's performance. Note that in this environment, the runtime of the slow simulator (t_s) is not appreciably slower than the runtime of the fast simulator (t_f). This and overhead in implementation prevents us from realizing the theoretical maximum speedup. In more realistic design optimization settings where t_f t_s, such as the motivating PEX example, we anticipate a substantially larger speedup.

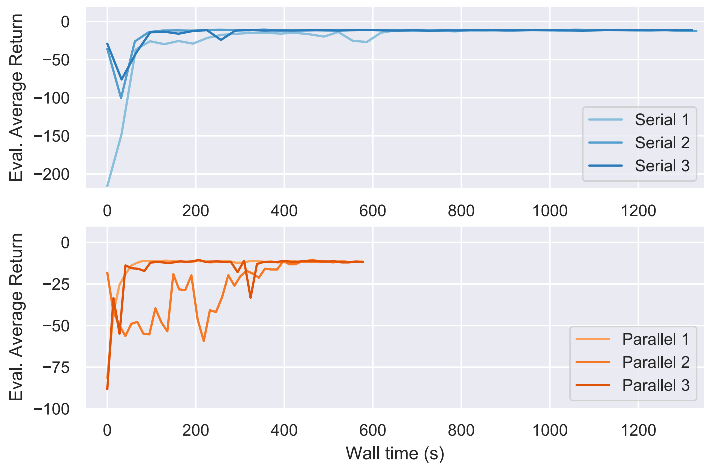

Figure 8: Comparison of total training time for the parallel and serial algorithms across three seeds.

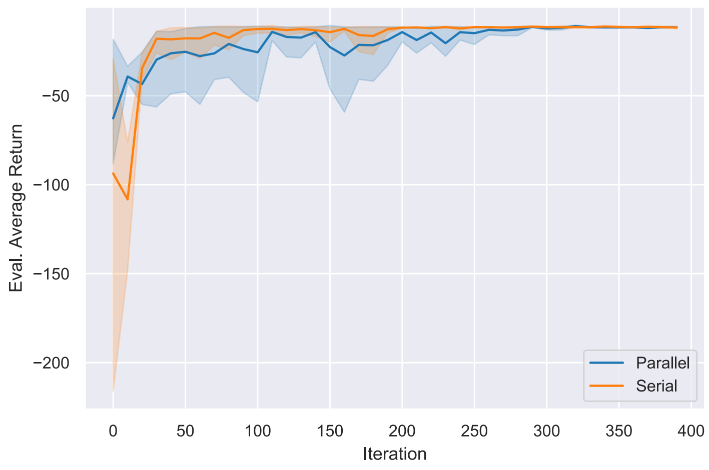

Figure 9: Evaluation average return of the parallel algorithm versus the serial algorithm averaged over 3 seeds.

We also notice that it is implicit from the construction of our algorithm that the evaluation time of the agent is improved. Specifically, the time it takes the agent to perform an action is dominated by the type of simulator it uses for the environment. Since $t_f < t_s$, and the offset predictor takes negligible time to run, a step when using the fast simulator with an offset predictor is much faster than one that uses the slow simulator. Therefore, it must be the case that the evaluation phase in our algorithm is faster than vanilla A2C or A2C with transfer learning.

5 Conclusion

We have demonstrated that there are instances in which one-shot transfer learning results in suboptimal performance, necessitating some interaction with a slow simulator to improve an agent's performance in a real environment. We approached this by introducing an offset predictor that compensates for the inaccuracies of a fast simulator and reduces the dependency on the slow simulator during the evaluation phase. We successfully improve the training time by introducing different forms of parallelism when training the offset predictor on data from the slow simulator. Our results have shown that we were able to maintain good agent evaluation performance while cutting CPU training time in half in a relatively generous setting where the slow simulator is only marginally slower than the fast.

6 Further Work

We consider two modifications to our algorithm to further improve results. The first concerns the interpretation of our problem as a data adaptation problem as described above, where the source and target distributions are derived from fast and slow simulators respectively. With this interpretation, one can apply various known techniques to calculate the *dataset shift* induced by our fast simulator's approximation. If the dataset shift is adequately complex, there are various techniques to cope, as discussed in [5]. In any case, methods exist for determining the predictive uncertainty of this dataset shift as outlined in [8]. In the space of image recognition, [10] provides a way to cope with dataset bias using an adaptation layer in a CNN that does not require fine-tuning. In our space, one could develop a DNN, a *state predictor*, which acts as a simulator that supplants the fast or slow simulator.

The second modification assumes we have adequately sampled from our fast and slow "distributions" enough to produce some offset predictor or state predictor as denoted above (this is no longer necessarily a DNN). Note, [2] has shown that we can actually choose some parameter $\alpha \quad [0, 1]$ to determine the amount that we would like to fit our model to the source and target distributions. They find that empirically the optimal α is neither 0 nor 1 but instead a value in between. However, in our setting we only wish to minimize the target error (i.e. match the slow predictor entirely). We now consider "switching off" the training of the offset predictor before we have finished training the agent, after which we label our fast simulator and offset predictor jointly as an optimized fast simulator. We now only need to consider our slow simulator alongside either a state predictor or our optimized fast simulator.

We also consider a modification to our algorithm where we avoid running both the slow and optimized fast simulators (as is the case when we produce the offset predictor) in the *late-training* phase of our agent. In further training of our agent, we instead *choose* one of the simulators in some optimized fashion to maximize a function of evaluation performance and effort which in our case is time. By using some Bayesian optimization techniques to determine the appropriate mixture of the simulations, we emulate the methods discussed in [6]. More precisely, [6] uses Entropy Search in an attempt to maximize information gain per experiment. Other techniques used broadly in simulator-to-robot transfer, such as in [1], could potentially be used here to increase increase adaptability of our agent in our fast simulator when acting in our slow simulator.

7 Contributions

Matthew Anderson worked on the serial implementation of the proposed algorithm, the `PointMass` environments, and the related visualization graphs. Matthew was also involved in the project documentation.

Emaan Hariri helped develop the serial algorithm and worked on the implementation of the custom `PointMass` environments and explored alternatives. Emaan also developed the graphs and diagrams above and researched potential further work.

Sayan Paul worked on the implementations of both the serial and parallel versions of the environment as well as the testing framework for the various environments. Sayan also worked on formalizing the potential realized training speedup achieved with parallelization and wrote on the proposed architecture above.

Special thanks to Keertana Settaluri and Kourosh Hakhamaneshi for their guidance and input. Aspects of this project build on their work and ideas.

References

[1] R. Antonova, A. Rai, and C. G. Atkeson. Deep kernels for optimizing locomotion controllers. In S. Levine, V. Vanhoucke, and K. Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 47–56. PMLR, 13–15 Nov 2017.

[2] S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. C. Pereira, and J. W. Vaughan. A theory of learning from different domains. *Machine Learning*, 79:151–175, 2009.

[3] P. G. Breen, C. N. Foley, T. Boekholt, and S. Portegies Zwart. Newton vs the machine: solving the chaotic three-body problem using deep neural networks. *arXiv e-prints*, art. arXiv:1910.07291, Oct 2019.

[4] K. Hakhamaneshi, N. Werblun, P. Abbeel, and V. Stojanović. Late breaking results: Analog circuit generator based on deep neural network enhanced combinatorial optimization. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–2. IEEE, 2019.

[5] W. M. Kouw and M. Loog. An introduction to domain adaptation and transfer learning, 2018.

[6] A. Marco, F. Berkenkamp, P. Hennig, A. P. Schoellig, A. Krause, S. Schaal, and S. Trimpe. Virtual vs. real: Trading off simulations and physical experiments in reinforcement learning with bayesian optimization. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1557–1563, May 2017. doi: 10.1109/ICRA.2017.7989186.

[7] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. *arXiv e-prints*, art. arXiv:1712.05889, Dec 2017.

[8] Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. V. Dillon, B. Lakshminarayanan, and J. Snoek. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift, 2019.

[9] K. Settaluri, A. Ali, Q. Huang, K. Hakhamaneshi, and B. Nikolic. Autockt deep reinforcement learning of analog circuit designs (submitted for publication). In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2020.

[10] E. Tzeng, J. Hoffman, N. Zhang, K. Saenko, and T. Darrell. Deep domain confusion: Maximizing for domain invariance, 2014.