A APPENDIX

B CLASSIC CONTROL EXPERIMENT DETAILS

For the classic control experiments we use the OpenAI gym (Brockman et al., 2016). To train all policies we use the DDPG algorithm, where the policies are parameterized by three layer MLPs with 256 hidden units per layer. We use the Adam optimizer, and search for a learning rate in $[1 \times 10^{-5}, 1 \times 10^{-3}]$.

For mountain car we train for a total of 15000 timesteps and begin training after 5000 timesteps. For pendulum, we train for a total of 50000 timesteps and begin learning after 25000 timesteps.

C BASELINES

Beyond the ground-truth reward, we compare the HERON algorithm with two ensemble baselines inspired by Brys et al. (2017). These ensemble baselines train a separate policy on each reward factor, and then combine the policies' outputs in a given state to select an action. In every environment we train each policy in the ensemble with the similar parameters as used for the reward engineering baseline and we again tune the learning rate in $[1 \times 10^{-5}, 1 \times 10^{-3}]$.

As described in the main text, we consider two variants of this ensemble based algorithm: one where the action is selected according to an average over each policy ($a \leftarrow \text{argmax}_{a \in \mathcal{A}} \sum_{k=1}^{n} \frac{1}{n} \pi_k(s, a)$) and one where the preference ranking used as input to HERON is used to combine the actions ($a \leftarrow \text{argmax}_{a \in \mathcal{A}} \sum_{k=1}^{n} \gamma^k \pi_k(s, a)$). With the second variant, γ is selected from $\{0.25, 0.35, 0.45, \cdots, 0.95, 0.99, 1\}$.

We also examine the performance of a reward engineering baseline where the reward is formulated as $\sum_{i=1}^{n} \beta^i z_i$, where β is a hyperparameter selected from $\{0.3, 0.4, ..., 0.9, 1.0\}$ and z_i are the normalized reward factors. The reward factors are ordered according to the HERON reward hierarchy, making this a very realistic and competitive reward engineering baseline. However, we came across a few challenges when trying to make this algorithm work. First, the reward factors all need to be normalized, which either requires complex algorithms or multiple agent rollouts before training. In addition, we find that this baseline is very sensitive to β and therefore has a higher tuning cost. In addition, it can often not beat the performance of HERON. We plot the performance of the reward engineering baseline in Figure 7. Note that this plot shows performance over all of training, and HERON typically displays larger reward (comparatively) in the last stages of training.

As we can see from Figure 7, the reward engineering baseline requires extensive tuning to achieve good performance. In addition, the choice of normalization strategy is very important (Figure 7f). These results further show the benefits of HERON.

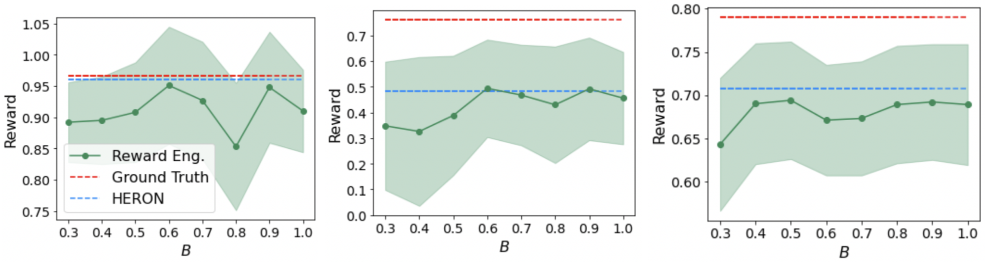
D ROBOTICS

All of our experiments are conducted with the PyBullet simulator (Coumans & Bai, 2016). The reward factors in each environment are as follows: for Ant, it is whether the robot is alive, the progress towards the goal state, whether the joints are at their limits, and whether the feet are colliding. For HalfCheetah, the factors are the potential and the power cost. For Hopper, the factors are the potential, an alive bonus, and the power cost.

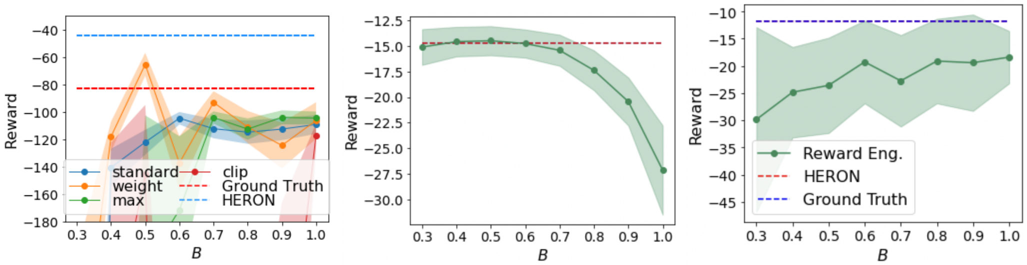
E TRAFFIC LIGHT CONTROL

In our experiments we train four agents in a two by two grid. The length of each road segment is 400 meters and cars enter through each in-flowing lane at a rate of 700 car/hour. The traffic grid can be seen in Figure 8. The control frequency is 1 Hz, i.e. we need to input an action every second. The reward is based on the following attributes for each agent n:

- q^n: The sum of queue length in all incoming lanes.
- wt^n: Sum of vehicle waiting time in all incoming lanes.



(a) Ant (b) Half-Cheetah (c) Hopper



(d) Traffic Lights (e) Pendulum (f) Mountain Car

Figure 7: Ablation study of the reward engineering baseline.

- dl^n: The sum of the delay of all vehicles in the incoming lanes.
- em^n: The number of emergency stops by vehicles in all incoming lanes.
- fl^n: A Boolean variable indicating whether or not the light phase changed.
- vl^n: The number of vehicles that passed through the intersection.

We can then define the reward-engineering reward as

$$R^n = -0.5q^n - 0.5wt^n - 0.5dl^n - 0.25em^n - fl^n + vl^n.$$

All algorithms have the same training strategy. Each agent is trained for three episodes with 3000 SUMO time steps each. At the beginning of training the agent makes random decisions to populate the road network before training begins. Each algorithm is evaluated for 5000 time steps, where the first 1000 seconds are used to randomly populate the road. For adversarial regularization, we use the ℓ_2 norm to bound the attacks δ.

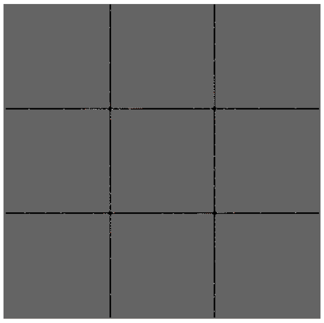


Figure 8: Traffic light control environment.

F RLHF COMPARISON

To explicitly compare RLHF with HERON, we compare the algorithms in the pendulum environment. To simulate human feedback, we rank one trajectory over another if the ground truth reward achieved by that trajectory is higher than the ground truth reward achieved by the other trajectory.

We then evaluate the performance of this simulated RLHF algorithm when varying amounts of feedback are given. The results can be seen in Figure 9. In this table we vary the number of feedbacks in RLHF, while keeping the number of feedbacks for HERON constant. In this setting HERON can perform as well as RLHF, but such good performance is not guaranteed in every environment.

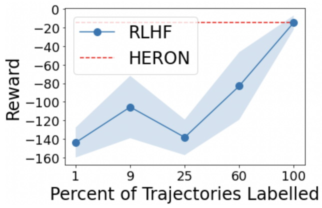


Figure 9: RLHF comparison in the Pendulum Environment.

G HERON FLEXIBILITY

In this section we evaluate how the behavior of the policies trained by HERON change when we change the reward hierarchy. We plot several hierarchies in Figure 10. The reward engineering is the thick black line. We try three factors as the most important factor (num_passed, wait time, and delay). We notice that all these observations can outperform the reward engineering reward, even though we measure the return with the reward engineering reward. One important deviation from this good performance is when wait time is not ranked highly. The wait time is a very important factor, and when we do not put this variable high up in the hierarchy, the performance becomes unstable when measured according to the reward engineering reward. This is because if we ignore the wait time of cars, the policy may make some cars wait for a long time, which is not ideal. However, this can easily be accounted for in the reward design process.

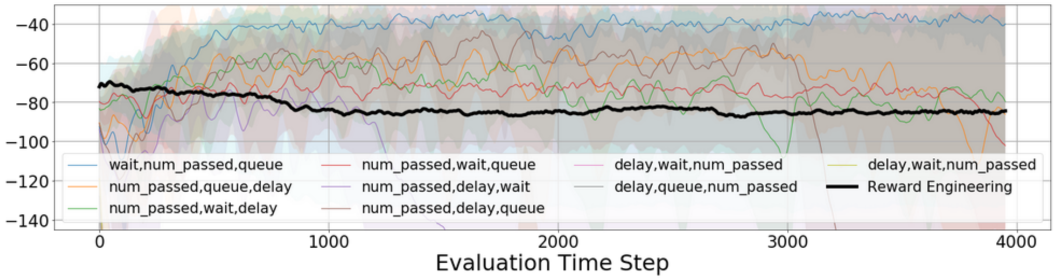


Figure 10: Different reward hierarchies in HERON.

We also show the level the decision tree induced by HERON reaches in Figure 11. This may change with different reward hierarchies (this one in particular priorities queue length, wait time, and delay), but as we can see from the figure, a relatively similar proportion of decisions are made at each level of the decision tree. We also remark different reward factors may be correlated (i.e. queue length and number passed), so the second factor may not have many decisions made with it.

H CODE GENERATION

In this section we describe details for the code generation task.

H.1 BEHAVIOR CLONING

To train the initial behavior model we use behavior cloning (supervised fine-tuning) to adapt the pre-trained CodeT5 to the APPS task. In particular, we use train with the cross-entropy loss for 12000 iterations, using a batch size of 64. We use the Adam optimizer with a learning rate of 2×10^{-5}.

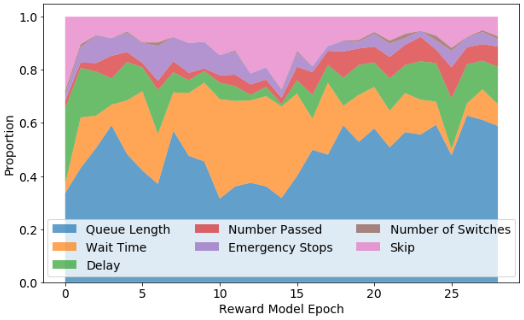


Figure 11: Different level reached by decision tree in HERON.

H.2 TEMPERATURE SELECTION

A hyperparameter that can have a large impact on generation quality is the temperature parameter, which essentially alters how greedy we are in the next-token sampling step. In all settings we follow the implementation of Le et al. (2022), using a temperature of 0.6 for APPS and 1.2 for MBPP. In addition, we sample tokens greedily to construct a baseline sample for each problem.

H.3 REWARD MODEL

It has been noted that reward models often overfit to the dataset (Ouyang et al., 2022). Therefore we use a smaller version of CodeT5 for our reward model with only 220 million parameters. We train this model for around 40000 steps with a batch size of 64. This is roughly a single epoch on the preference dataset, which is comprised of 20 samples per problem sampled from the behavior model and some expert samples provided by the APPS dataset. We use the Adam optimizer with a learning rate of 2×10^{-5}.

H.4 REINFORCEMENT LEARNING

Once we have trained the reward model, we assign a reward to each program in our preference dataset and train using reinforcement learning on this dataset. Similar to Le et al. (2022), we train on the policy gradient loss and add the cross entropy loss as a regularization term. We compare our method to two reward engineering rewards:

CodeRL reward. The first reward we compare HERON to is from CodeRL, which defines the reward as

$$r_{\text{CodeRL}}(s) = \begin{cases} -1.0 & \text{if program } s \text{ fails to compile} \\ -0.6 & \text{if program } s \text{ has a runtime error} \\ -0.3 & \text{if program } s \text{ fails a unit test} \\ 1.0 & \text{if program } s \text{ passes all unit tests.} \end{cases}$$

PPOCoder reward. The second reward we compare HERON to is based on PPOCoder, which has the insight to include syntactic similarity to expert samples in the reward. This effectively smooths the reward, and can therefore make the reward more informative. In particular, they compare the abstract syntax trees of the generated programs with the expert example programs. This is computed as

$$R_{ast}(s, \widehat{s}) = Count(AST_s, AST_{\widehat{s}})/Count(AST_s).$$

We then construct the final PPOCoder based reward as $r_{\text{PPOCoder}}(s) = r_{\text{CodeRL}}(s) + \lambda MEAN_{\widehat{s}}(r_{ast}(s, \widehat{s}))$, where $MEAN$ is the mean operator. We tune $\lambda \in \{0.001, 0.01, 0.1, 1\}$. We remark that the original PPOCoder reward contains more reward factors, but we do not use all of them due to the large tuning cost required to tune the ourselves.

For both of these rewards and the HERON reward we tune the learning rate in $\{3 \times 10^{-6}, 5 \times 10^{-6}, 8 \times 10^{-6}\}$.

H.5 EXAMPLE PROGRAMS

To further analyze the performance of HERON, we examine some of the programs generated by HERON. These programs are randomly selected. We display concatenated prompts and completions in Figure 12.

I REWARD TRAINING

In this section we detail our reward model training. For the classic control tasks and the traffic light control task we do not have a good initial behavior policy, so we must train our reward model in an iterative manner. In these settings, we iteratively update the reward model using samples from the current version of the policy. In this way the reward model is trained on samples generated from progressively better policies.

As we mentioned in our discussion on the computational costs of HERON, the cost of reward model training depends on the frequency at which the reward model is trained. For the classic control environments we simply use a linear training schedule, in which the reward model is updated every 400 steps. For traffic light control we train the reward model with an annealed frequency, where the reward model is trained every $100\upsilon^t$ steps, where υ is set 1.3 and t is the current time step.

We demonstrate the multi-step reward model training in Figure 13. The sharp drop in accuracy occurs at time step 1000, where the behavior model changes from random to a trained policy. This large change in accuracy indicates that multi-step reward model training is needed, as reward models trained on random behavior do not perform as well when the behavior changes.

I.1 THE α HYPERPARAMETER

Formal description of shaping factor: Given a trajectory τ, let us compare it with n other trajectories $\tau_1, \dots, \tau_n$. Let $F(\tau)$ denote the average level of the decision tree τ wins at. To allow us to incorporate domain knowledge into HERON, we multiply the reward assigned to τ by a factor $\alpha^{F(\tau)}$, where α is a hyperparameter. When the reward factors are categorical, $F(\tau)$ can capture which category τ lies in, and multiplying the reward by $\alpha^F(\tau)$ can control the reward separation between different categories.

Visual description of shaping factor: As mentioned in the main text, the α hyperparameter can be used to control the shape of the rewards. In Figure 14, we show how changing α changes the reward shape in the code generation task.

J COMPUTATIONAL SETUP

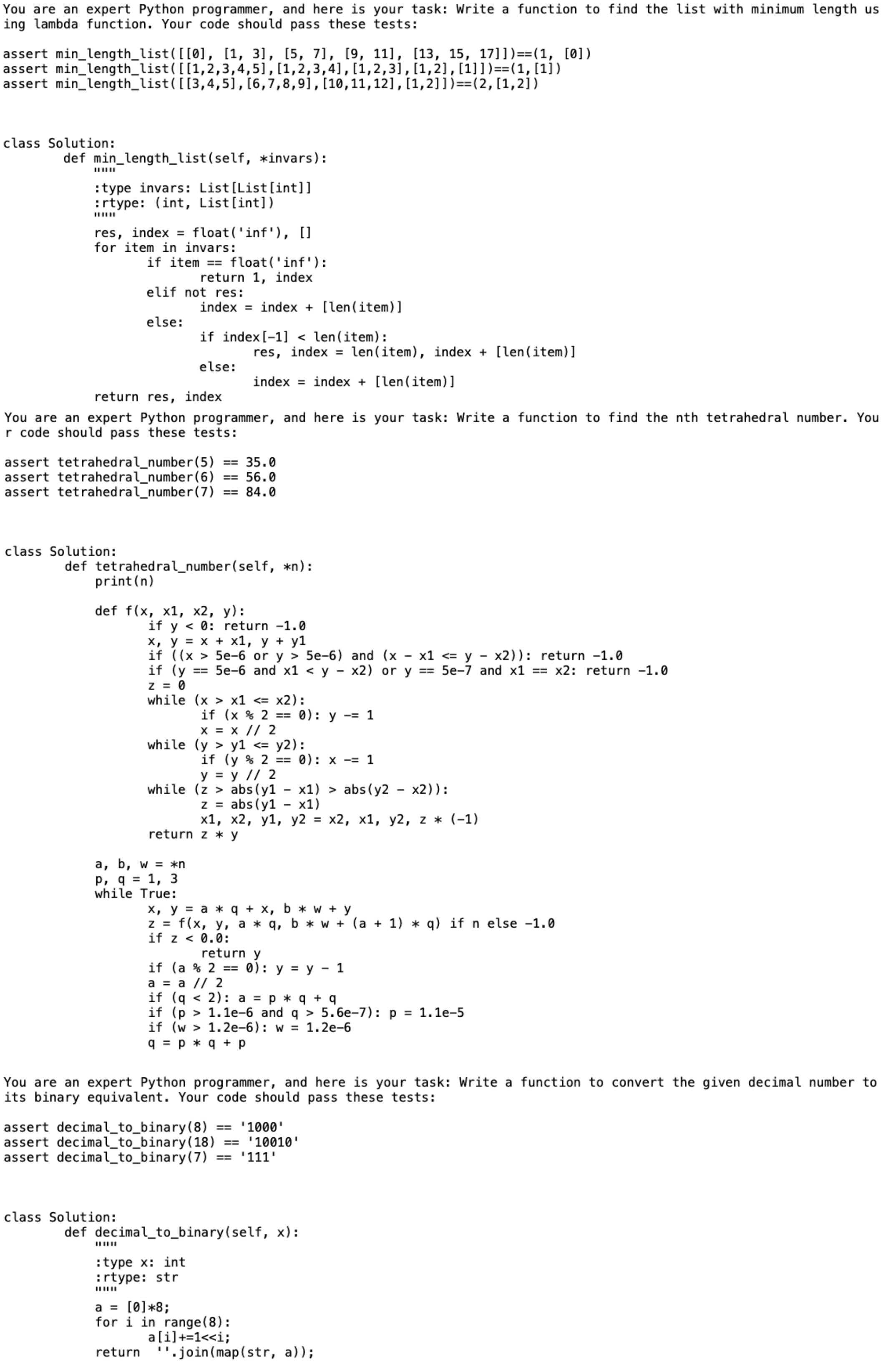
For the classic control tasks and traffic light control experiment we run experiments on Intel Xeon 6154 CPUs. For the code generation task, we train with Tesla V100 32GB GPUs.

K ROBOTICS LEARNING CURVES

In Figure 15 we display the learning curves in the robotics environments.

L LIMITATIONS

The main limitation of HERON is that not every problem will contain an obvious ranking over the reward factors, as some factors may be equally important. We propose to mitigate this limitation in future works by allowing for ties or using a randomized decision tree in the preference elicitation procedure.



```
You are an expert Python programmer, and here is your task: Write a function to find the list with minimum length us
ing lambda function. Your code should pass these tests:

assert min_length_list([[0], [1, 3], [5, 7], [9, 11], [13, 15, 17]])==(1, [0])
assert min_length_list([[1,2,3,4,5],[1,2,3,4],[1,2,3],[1,2],[1]])==(1,[1])
assert min_length_list([[3,4,5],[6,7,8,9],[10,11,12],[1,2]])==(2,[1,2])



class Solution:
        def min_length_list(self, *invars):
            """
            :type invars: List[List[int]]
            :rtype: (int, List[int])
            """
            res, index = float('inf'), []
            for item in invars:
                   if item == float('inf'):
                          return 1, index
                   elif not res:
                          index = index + [len(item)]
                   else:
                          if index[-1] < len(item):
                                 res, index = len(item), index + [len(item)]
                          else:
                                 index = index + [len(item)]
            return res, index
You are an expert Python programmer, and here is your task: Write a function to find the nth tetrahedral number. You
r code should pass these tests:

assert tetrahedral_number(5) == 35.0
assert tetrahedral_number(6) == 56.0
assert tetrahedral_number(7) == 84.0



class Solution:
        def tetrahedral_number(self, *n):
            print(n)

            def f(x, x1, x2, y):
                   if y < 0: return -1.0
                   x, y = x + x1, y + y1
                   if ((x > 5e-6 or y > 5e-6) and (x - x1 <= y - x2)): return -1.0
                   if (y == 5e-6 and x1 < y - x2) or y == 5e-7 and x1 == x2: return -1.0
                   z = 0
                   while (x > x1 <= x2):
                          if (x % 2 == 0): y -= 1
                          x = x // 2
                   while (y > y1 <= y2):
                          if (y % 2 == 0): x -= 1
                          y = y // 2
                   while (z > abs(y1 - x1) > abs(y2 - x2)):
                          z = abs(y1 - x1)
                          x1, x2, y1, y2 = x2, x1, y2, z * (-1)
                   return z * y

            a, b, w = *n
            p, q = 1, 3
            while True:
                   x, y = a * q + x, b * w + y
                   z = f(x, y, a * q, b * w + (a + 1) * q) if n else -1.0
                   if z < 0.0:
                          return y
                   if (a % 2 == 0): y = y - 1
                   a = a // 2
                   if (q < 2): a = p * q + q
                   if (p > 1.1e-6 and q > 5.6e-7): p = 1.1e-5
                   if (w > 1.2e-6): w = 1.2e-6
                   q = p * q + p

You are an expert Python programmer, and here is your task: Write a function to convert the given decimal number to
its binary equivalent. Your code should pass these tests:

assert decimal_to_binary(8) == '1000'
assert decimal_to_binary(18) == '10010'
assert decimal_to_binary(7) == '111'



class Solution:
        def decimal_to_binary(self, x):
            """
            :type x: int
            :rtype: str
            """
            a = [0]*8;
            for i in range(8):
                   a[i]+=1<<i;
            return  ''.join(map(str, a));
```

Figure 12: Example programs generate by LLMs trained with HERON.

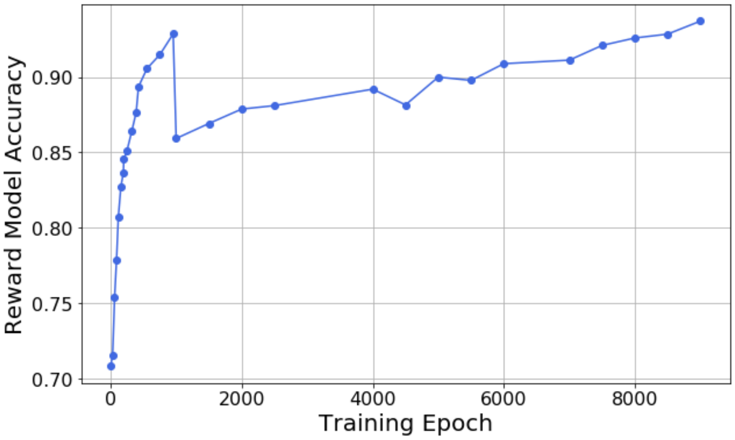
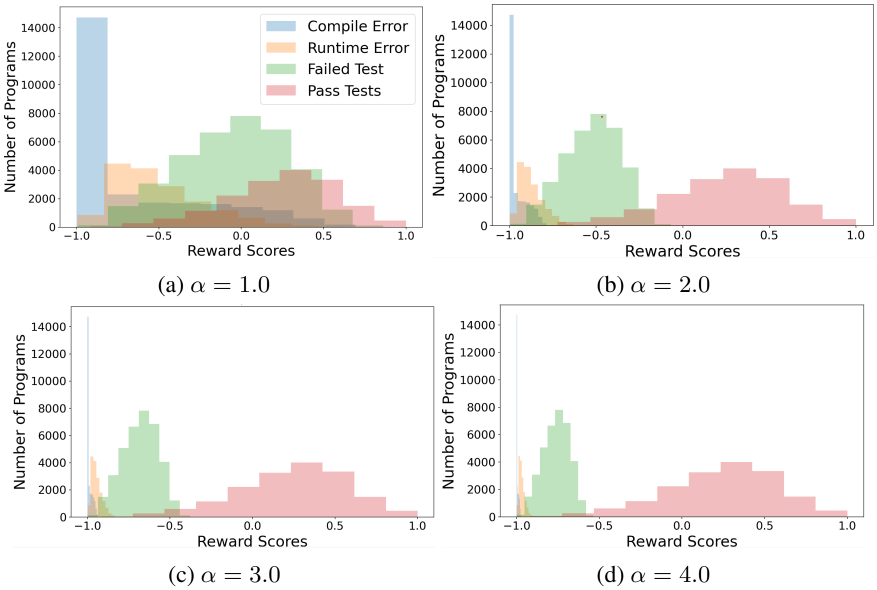


Figure 13: Reward model accuracy throughout training.



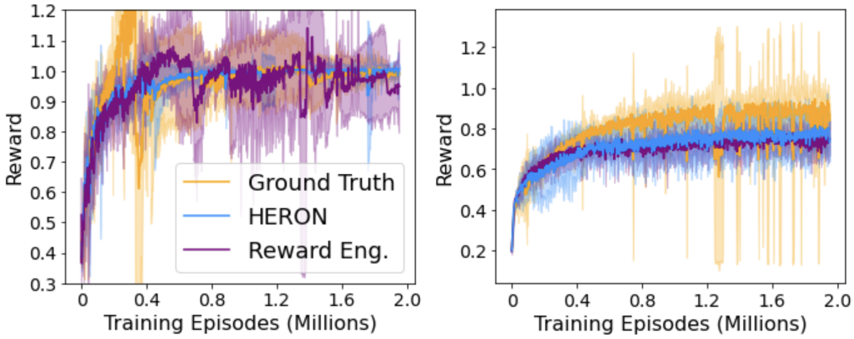
(a) $\alpha = 1.0$

(b) $\alpha = 2.0$

(c) $\alpha = 3.0$

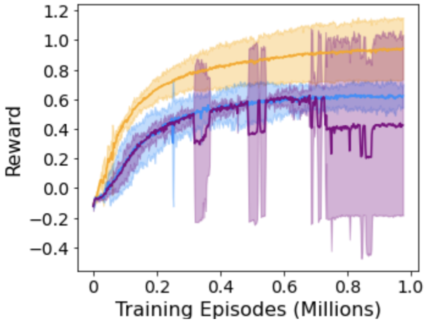
(d) $\alpha = 4.0$

Figure 14: Reward shape with different values of α.



(a) Ant

(b) Hopper



(c) HalfCheetah

Figure 15: Training curves in different robotics tasks.