Electronic Supplement

A Comparison of State-of-the-Art Reinforcement Learning Algorithms Applied to the Traveling Salesman Problem

Kenneth Schröder[(1)], Alexander Kastius[(2)], Rainer Schlosser[(3)]

[(1)]Hasso Plattner Institute, University of Potsdam, dev@kenneth-schroeder.com

[(2)]Hasso Plattner Institute, University of Potsdam, alexander.kastius@hpi.de

[(3)]Hasso Plattner Institute, University of Potsdam, rainer.schlosser@hpi.de
(corresponding author)

Contents

Appendix

1 Reproducibility

All of the implementations of this paper are available in a GitHub repository[1].

The two main branches `tianshou` and `master_bench` are used for the experiments mentioned in later sections of this paper. Of the other branches, `master` and `v1-v4` are original branches of Kool, Van Hoof, Welling's repository. Custom, experimental implementations of SAC and PPO were developed in `sac` and `ppo`. All of the results presented in this paper are created focusing on reproducibility. The respective scripts, logs, and configurations are described in the following subsections.

Server Configuration

The experiments are conducted on the `Delos` server of the Hasso Plattner Insitute. At the time of writing this paper, this server contains two 20-core Intel Xeon Gold 6148 CPUs at 2.40GHz, four NVIDIA Tesla V100 SXM2 32GB GPUs, and 1.5TB of main memory. All of the experiments are restricted to exactly one GPU each. Multiple projects were conducted on this server in parallel.

Training Reproducibility

In the `tianshou` and `master_bench` the random seed, as well as all of the relevant hyperparameters, interchangeable modules, and algorithms, are exposed as command-line arguments. If no random seed is specified, a random number between 1 and 9999 is generated and set as a random seed for `torch` and `numpy` at the beginning of a run. All of the configuration settings, including the random seed, are saved to a JSON file in the `args` folder. This automatically randomizes experiments while preserving all necessary information for full reproducibility. The command-line arguments can also be imported from a line of a comma-separated file using the `--args_from_csv` and `--csv_row` specifiers. This enables the planning of whole tables of experiment configurations. Sequential scheduling of all the lines in a CSV file is facilitated using the `scheduler.sh` bash script. Once a particular experiment is running, the accruing time series data, including performance scores, loss values, changing learning rates, etc., is logged using TensorBoard and saved to the logging directory `log_dir`. At the end of each run, the learned policy parameters are saved to the `policy_dir` folder. All of the files produced by a single experiment receive a unique name, consisting of a prefix specified via the `--run_name` command line argument and the current timestamp.

Evaluation Setup

For the evaluation of trained models, the unique name of a finished experiment can be specified. The model is automatically recreated using the logged command line arguments of the original experiment and the learned policy parameters from the `policy_dir` folder. The model is then evaluated using a batch of newly generated data, and of all trajectories, including the transition probabilities, are saved to the `eval_logs` directory.

[1] `https://github.com/Kenneth-Schroeder/attention-next-gen-rl`

Visualization Setup

The training data from `log_dir` can be analyzed using the TensorBoard browser viewer. The TensorBoard logs consist of event files, which cannot be easily accessed for custom visualizations. For the visualizations in this paper, an aggregator script and two Jupyter Notebooks are developed. The TensorBoard aggregator converts the TensorBoard event files to comma-separated files for `plotting.ipynb`. This notebook creates Plotly line charts to compare time series data between different experiments. Configurations for each visualization are saved to JSON files in the `figure_metas` folder, enabling automated adjustments to multiple visualizations without the need to redefine all of the plot parameters.

The second notebook `animating.ipynb` is used to visualize the decision process in individual problem instances from the evaluation experiments using Matplotlib. This notebook is based on code for visualizations by Kool, Van Hoof, Welling.

2 Hyperparameter Tuning

2.1 Deep Q-Learning (DQN)

The DQN implementation of Tianshou follows the DQN and Double DQN publications by [6] and [9] and is, therefore, an off-policy value-learner implementation. Figure 1 displays the constructor parameters of the class. The `estimation_step` is the n_step parameter for TD(n) in off-policy DQN and the Double DQN settings can be controlled by `target_update_freq` and `is_double`. In this article, all DQN experiments include a target network for Double DQN, which is updated with a frequency of 100 updates. Tianshou also offers possibilities for reward normalization, but this is not used in the experiments of this article. Each agent also includes decaying ϵ-greedy exploration.

The first parameter analyzed for optimizing the DQN agent is the learning rate. Multiple runs for each learning rate are conducted to test this hyperparameter's limits and usable range. Figure 2 shows representative graphs for runs experiments with a learning rate of $1e-2$ to $1e-6$. The corresponding loss developments are plotted in Figure 3 with a logarithmic y-axis.

These graphs show that a learning rate of $1e-2$ is too high for the DQN agent. The performance decreases at the beginning of training but fluctuates quite a lot due to the substantial updates to the network parameters. At around 9 million training steps, the loss explodes, and the performance collapses, which indicates an unlucky batch of data with a too-large learning rate, causing a major destructive update to the network. The graphs corresponding to the experiments with learning rates $1e-5$ and $1e-6$ show slightly above-random performance but fail to improve further. One likely reason for this early performance stagnation could be that the optimization process gets stuck in a bad local minimum of the loss function, with a learning rate too small to escape. The best learning rates for the DQN agent seem to lie between $1e-3$ and $1e-4$. Using these learning rates, the agents' performance continuously approaches the optimum throughout these shorter experiments. Notably, the learning stability of the agent with a learning rate of $1e-3$ seems to be better than with a learning rate of $1e-4$, especially at the beginning of training. This is counterintuitive, as a lower learning rate usually corresponds to more minor changes to the policy and less drastic performance fluctuations. One possible reason for this phenomenon is a combination of the random network initialization and the highly undirected exploration (randomly-initialized deterministic policy and ϵ-greedy) of the DQN agent at the beginning of training. The Q-values are likely very close together at the start. Without significant updates by high learning rates or directed exploration, the agent might take

time to escape this volatile state.

For the experiments of this article, two types of learning rate schedules are compared. Figure 4 shows their change over time. With exponential learning rate decay, the current learning rate is multiplied by a value $d \in (0, 1)$ after each policy update. In the plotted example the batch size is $64 \cdot 20 \cdot 2 = 2560$, the learning rate starts at $5e-4$ and decays over $\frac{12800000\,\text{samples}}{2560\,\text{batch size}} = 5000$ batches with a decay factor of 0.9992 to a value of $\approx 1e-6$. The second learning rate schedule used in this article is PyTorch's cyclic learning rate of mode "triangular2", which means that the learning rate oscillates linearly between two defined learning rate values, halving in amplitude with each cycle. The purpose of a schedule like this is to prevent the process from getting stuck in suboptimal local minima. By increasing the learning rate from time to time, the agent gets a chance for more significant steps in the loss space. This can temporarily lead to performance drops but allows better performance after recovery.

Figure 5 displays the performance of the DQN agent with the two learning rate schedules presented above. The first exponential run, "exp #1", starts with a learning rate of $5e-4$, and "exp #2" begins with a learning rate of $1e-3$. Otherwise, all configurations are identical. All three runs are characterized by increasing stability throughout training and good performance. The first exponential setup seems temporarily stuck at a bad local minimum, which can result from the short training in combination with the quickly decreasing learning rate. The example with the cyclic learning rate recovers well from some early instability caused by the increasing learning rate. What was previously discovered with the fixed learning rates also shows in the experiments with learning rate schedules: The configuration with the highest starting learning rate seems to stabilize first, as seen in Figure 6.

The final optimization experiments for the DQN agent compare the effects of n-step updates in Figure 7. Notably, only fixed learning rates are used for these runs. The 5-step and 10-step versions are characterized by fast convergence and high stability, likely due to the reduced overestimation bias and only slightly increased update variance.

2.2 Policy Gradient (PG)

The PG algorithm in Tianshou is an implementation of the REINFORCE algorithm, which means that it uses Monte Carlo reward-to-go estimates to reinforce actions. In comparison to the PG of Kool, Van Hoof, Welling, it does not reinforce each action by the whole trajectory reward, but Tianshou's PGPolicy does not support baselines. The agent is trained on-policy, and the constructor parameters are listed in Figure 1. The `dist_fn` distribution is used to convert the model's logit outputs to action probabilities. Since discrete actions characterize combinatorial optimization problems, a categorical distribution is used in the following experiments. By setting `action_scaling` to `False`, the resulting actions would be bound to the interval $[-1, 1]$ by the `action_bound_method`. For this article, it is set to `True` so that the actions remain in $[\text{action\_spaces.low}, \text{action\_spaces.high}]$. Learning rate schedules can be included by `lr_scheduler` and deterministic evaluation is possible by the `deterministic_eval` parameter.

Similar to the DQN experiments, fixed learning rates are examined first for the PG optimization. Figure 8 shows representative PG performance developments throughout training for learning rates between $1e-3$ and $1e-6$ with the corresponding logarithmic loss graphs in Figure 9. According to these results, the most effective range of learning rates for the PG agent is between $1e-4$ and $1e-5$, as the example with a learning rate of $1e-3$ is characterized by exploding loss and consistently bad performance. All the other runs show consistent performance improvements with convergence speed proportional and stability slightly inversely proportional to the learning rate.

The loss values for the PG optimization are negative because of the strictly negative returns in TSP and the loss function $-\log \pi_\theta\left(a_t|s_t\right) G_{t:T}$. Although this loss is unbounded towards negative infinity, it is expected to converge towards 0 in training, as actions that produce high negative loss (by their low probability or large negative returns) become even less likely. Compared to strictly positive returns, the intuition behind the network updates changes with strictly negative returns. Instead of all probabilities being *increased* proportionally to their return, with strictly negative returns, all probabilities are *decreased* proportionally to their negative return with each network update. Suppose two large negative returns with different probabilities in the loss calculation. The probability of the improbable sample will be decreased more drastically than the probability of the probable sample because of the large log-probability factor in the loss calculation. Intuitively, it would make sense to decrease the probability of probable large negative returns more. This thought process is discussed on Stack Exchange (see `https://ai.stackexchange.com/questions/2405/how-do-i-handle-negative-rewards-in-policy-gradients-with-the-cross`
and a user suggests adjusting the loss calculation for negative losses to $-\log(1-\pi_\theta\left(a_t|s_t\right)) G_{t:T}$. In the experiments of this article, this loss calculation led to worse-than-random results, as presented in Figure 10. For an intuitive explanation of why this fails, suppose two (almost optimal) *small* negative returns with different probabilities. With the adjusted loss formula, the probability of the probable sample will be decreased drastically, which is destructive, as the probabilities of almost optimal decisions should be kept high. Furthermore, the PG optimization objective and gradient formulas are not derived with any restrictions to positive numbers and can therefore be applied with negative returns without adjustments.

Figure 11 shows the result of the PG experiments with learning rate decay. Two representative graphs are plotted for each configuration. The starting learning rate for the exponential runs is $5e-5$ with a decay factor of 0.9992. The two cyclic runs have a lower LR bound of $1e-5$ and an upper bound of $3e-4$. All of the configurations are characterized by fast early

improvements and better stability with lower learning rates. The cyclic runs show some performance drops at points of higher learning rates but, in comparison to their DQN counterparts, do not achieve much better results after recovery.

As discussed in Section 3.1.4, the Transformer architecture of the experiments of this article is based on the implementations of [4]. They include options for batch and instance normalization in the encoder. With the transition to Tianshou, batch normalization can lead to exploding loss values and a collapse of training performance. This can happen if the number of training environments is smaller than the number of episodes used per network update. In this case, the batch normalization leads to different results at the time of gradient calculation compared to the time of data collection (Section 3.1.4). Because of this, the log probabilities can explode, which leads to the performance collapse as verified by the corresponding experiments with batch normalization in Figure 12. All other Tianshou setups in this article use instance normalization.

A similar issue occurs when using the PGPolicy's `repeat_per_collect` option to use a single batch of data for multiple network updates. With each update, the policy's probabilities change. When recalculating the log probabilities in-between updates, significant differences can result in exploding losses and performance collapses, as visualized in Figure 13.

2.3 Soft Actor-Critic (SAC)

Tianshou's SAC implementation follows the second version of SAC by [2], which uses only Q-learning critics and no state-value learners. It adds support for learning the entropy temperature parameter `alpha`. Special constructor parameters of DiscreteSACPolicy are `tau`, `alpha` and `exploration_noise` as displayed in Figure 1. Each critic in SAC has an additional target network used to compute the critics' gradients which by [2] has shown to stabilize training. The parameters of the target networks are exponential moving averages of their respective critics, and `tau` defines the update factor for these target networks. Noise can be added to the actions to solve the hard-exploration problem by using the `exploration_noise` parameter. `Alpha`, the entropy temperature parameter, can be set to a fixed value or to a tuple of a `target_entropy`, a `log_alpha` variable and an `alpha_optim` torch optimizer. If it is set to a tuple, SAC entropy temperature learning (Section 2.2.5) will be applied.

Figure 14 compares the performance developments during training of SAC agents with fixed learning rates. All of the runs are conducted with entropy temperature learning. None of the configurations cause performance collapses in these experiments, and all learning rate settings show continuously improving policies. The agents with lower critic learning rates of $1e-5$ perform worst and are characterized by lower stability in the second half of the training process. Notably, the graphs indicate a very low hyperparameter sensitivity of the SAC algorithm for combinatorial optimization. This is not true in general, and the low sensitivity in Figure 14 is only achieved after carefully selecting the learning rate for the temperature optimizer. Just by lowering this *temperature learning rate* from $1e-3$ to $3e-4$, the results are significantly worse, as depicted in Figure 15.

The training characteristics of the well-performing agents with fixed learning rates from Figure 14 can be analyzed in detail by looking at the development of the temperature parameter alpha and the other losses of the agents. During the first few updates of the training process, the critics' losses decrease significantly as they start to learn the values of state-action pairs (Figure 16). Simultaneously, the actor losses in Figure 17 decrease slightly as the loss calculations become more accurate due to the improving critics. Afterward, the actor losses increase because the negative entropy term in the loss calculations becomes less pronounced with the

rapidly decreasing alpha graphed in Figure 18. If the critics' learning rates are too low, they cannot keep up with the changes in the soft Q-targets caused by the change of entropy of the policy. This increases the corresponding critic losses in the second half of training, as shown in Figure 16.

The application of learning rate schedulers to the actors' and critics' learning rates does not affect the learning progress significantly, which is expected considering the few differences between the fixed learning rates. Figure 19 shows the results of the corresponding experiments. Although it is not recommended, SAC can also be used with fixed temperature parameters because the best alpha value is problem specific [2]. Figure 20 confirms that lower alpha values perform best on the TSP, which resulted from temperature learning as well.

2.4 Advantage Actor-Critic (A2C)

The A2C implementation of Tianshou is a synchronous version of A3C [5]. The A2CPolicy class is derived from PGPolicy, and the constructor parameters are listed in Figure 1. The advantage-values in A2C are estimated using Generalized Advantage Estimation. The corresponding lambda can be set by the `gae_lambda` parameter, and `max_batchsize` is used to stay within the memory constraints when estimating batches of state-values using the critic. The loss in A2C is a sum of three components: the actor loss, the critic loss weighted by `vf_coef` and the entropy loss weighted by `ent_coef`. For optimizing the performance of the A2C agent, different fixed learning rates for actor and critic are examined in Figure 21. These results are characterized by high stability and convergence to good results. The learning rate of the critic does not show significant effects, but the setups with a lower actor learning rate converge slightly slower. Because of the similarities between A2C and PPO, fixed learning rates from a broader range are compared in the PPO section.

The fast convergence of the A2C algorithm is also apparent in the experiments with learning rate schedules in Figure 22. For the exponential schedules, starting learning rates of $1e-4$ and $5e-5$ are used for the actor and critic, respectively, with a decay factor of 0.9992. In the cyclic configurations, the lower bound for both learning rates is $1e-5$, and the upper bound is $5e-4$. When the cyclic learning rates reach their maximum values, slight performance drops can be observed. Although the agent recovers well from these more substantial network updates, no significantly better minima are achieved.

Representative experiments on the settings of the critics are conducted on the A2C agent with exponential learning rate decay in Figure 23, 24, and 25. Figure 23 analyzes critics of architecture `v3` with different embedding sizes, i. e. the embeddings of the input nodes learned by the encoder have different lengths. The conducted runs on TSP with 20 nodes show no significant differences. Larger embedding sizes might be beneficial for larger problem sizes.

As described in Section 3.1.4, the `v1` critic architecture appends 0/1 encoded values to each input node to mark them as visited or unvisited. In the original implementation, unvisited nodes get a value of 0. Depending on the specific architecture, neural networks can have problems efficiently processing inputs of value 0, as they cannot be manipulated by multiplication. They might cancel out other values in attention calculations, for example. An alternative implementation is tested as the unvisited nodes are most relevant for the critic. This alternative inverts the visitation values so that unvisited nodes are marked by ones and visited nodes are marked by zeros. Figure 24 visualizes the results of these experiments and includes a run with `v3` architecture for comparison. In the setup of this article, both critic architectures have similar performance, and inverting the visited values shows no improvements.

Another possible problem for neural networks is producing negative outputs, as activation functions like ReLU[2], for example, commonly clip intermediate results to the positive range. To potentially simplify the learning of negative values for the critics, negations of the critic outputs are added in the experiments in Figure 25 and leaky ReLU[3] is tested as alternative. As can be seen in comparison to the reference runs without negation and with ReLU activation, there are no significant improvements by these adjustments.

[2] $\mathrm{relu}(x) = \max(0, x)$

[3] $\mathrm{leaky\_relu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$

2.5 Proximal Policy Optimization (PPO)

PPO in Tianshou is derived from the A2CPolicy but includes additional mechanisms to limit policy changes between updates as proposed by [8]. The complete list of parameters is shown in Figure 1. The epsilon used by PPO to clip the policy changes is defined by `eps_clip`. For off-policy PPO training environments, [10] propose dual-clipped PPO to prevent drastic policy updates from negative advantages with large ratios between behavioral and training policy. Dual-clipped PPO is supported by Tianshou by the `dual_clip` parameter, although Tianshou's PPO is designed as an on-policy algorithm.

The `value_clip` parameter makes it possible to train the value function with a PPO-like objective function which prevents drastic changes in the value estimates, as proposed by [3].

When using one batch of data for multiple policy updates, it can be beneficial to recompute the advantage-values in-between each step [1]. This can be done by setting the `recompute_advantage` parameter. advantage-values in each mini-batch can optionally be normalized by `advantage_normalizat`. The PPO extensions dual-clipped PPO, value function clipping and advantage recalculation are are not applied in this article.

The effects of different fixed learning rates on the PPO agent are visualized in Figure 26. According to these results, learning rates for actors and critics of $1e-3$ are too high, which causes an early performance collapse. As for the A2C agent, learning rates between $1e-4$ and $1e-5$ show fast convergence and increased stability and seem to be the best choice for the PPO agent. As in the A2C experiments, the actor's learning rate influences the convergence speed the most. Prolonged but stable performance improvements characterize a learning rate of $1e-6$ for both actor and critic.

The comparisons of learning rate schedulers are conducted with the same settings as for the A2C agent. For the exponential schedule, starting learning rates of $1e-4$ and $5e-5$ are used for actor and critic, respectively, with a decay factor of 0.9992. In the cyclic runs, the lower bound of both learning rates is $1e-5$, and the upper bound is chosen as $5e-4$. All of the runs demonstrate a fast convergence and high stability. Similar to the A2C experiments on learning rate schedules from Figure 22, the cyclic learning rates show performance drops at the peaks of the learning rate schedule in Figure 27, but those are much less pronounced, which is likely a result of the objective clipping of PPO.

Variance in PG approximations can be reduced by increasing the batch size. The configurations of the results in Figure 28 only differ in batch size, learning rate decay, and training length. In this magnified section of the training performances, the graphs of the setups with larger batch sizes fluctuate less and improve stability. Larger batch sizes can benefit extended and more precise training but come at the cost of higher memory requirements and slower convergence w. r. t. the number of samples trained, as there are fewer updates per fixed n samples.

Additional experiments have been conducted to analyze the effect of different GAE lambda values. Figure 29 shows that the Monte Carlo advantage approximation by $\lambda = 1.0$ achieves the best results with decreasing performance and stability towards lower lambdas. So in the case of TSP with problem size 20, a more considerable variance with lower bias in the advantage estimations seems preferable to a larger bias with lower variance. The relatively short trajectory lengths with a relatively small variance of realized returns are likely the main reason for this observation.

2.6 PG by Kool et al.

As mentioned in Section 3.1.4, Kool, Van Hoof, Welling use a vanilla PG algorithm for optimizing their policy network and include multiple options for baselines. Their training process

is adjustable by many training parameters, and some are analyzed in this section to get insights into their effects on hyperparameter sensitivity, convergence rates, and optimality.

Similar to the experiments on the RL algorithms above, fixed learning rates are examined first for Kool, Van Hoof, Welling's implementation. Figure 30 shows the effects of different learning rates on the performance developments of the PG agent. The training graphs look similar to the Tianshou PG results in Figure 8. A fixed learning rate of $1e-3$ is too large and causes performance collapse. All other runs are characterized by continuous performance improvements, with faster convergence by experiments with higher learning rates and increased stability by runs with lower learning rates. According to these graphs, starting learning rates between $1e-4$ and $1e-5$ are likely to produce the best results with learning rate schedules.

Regarding learning rate schedules, Kool, Van Hoof, Welling only offer exponential decay, which in this case is applied after each epoch (after 256k trained samples). Because of this, the decay factors considered in Figure 31 are adjusted compared to the Tianshou experiments. With starting learning rate of $1e-4$ and decay factors of 0.99, 0.97 and 0.95, the learning rates decay to values of $\approx 4e-5$, $5e-6$ and $6e-7$ respectively, over 100 epochs. All configurations are characterized by fast convergence and stability proportional to the learning rate decay. Considering the fixed learning rates results, a decay factor of 0.97 is the best tradeoff between stability and continuous learning improvements over the presented training length.

The three options for baselines provided by Kool, Van Hoof, Welling are a rollout baseline, an exponential moving average, and a critic baseline. As Kool, Van Hoof, Welling implement the vanilla PG variant, their baselines only need to provide one baseline value per problem instance, as opposed to one baseline value per partially solved instance. The exponential moving average baseline is initialized by the first realized solution length and updated with a moving average of subsequent results (i.e. $b \leftarrow \beta \cdot b + (1-\beta) \cdot l$, where b denotes the current baseline value, l is the most recent average solution length and β defines the decay). The critic baseline uses a neural network to predict the expected tour length of an unsolved instance and the rollout baseline is a unique proposition by Kool, Van Hoof, Welling inspired by self-critical training [7]. The value of the rollout baseline is the solution length achieved by a deterministic, greedy version (rollout) of the best policy so far. Figure 32 graphs the magnified results of all setups with each of the three baselines and includes a reference configuration with no baseline for comparison. According to these experiments, the rollout baseline outperforms their alternatives but might require computation overhead for computing solutions on each training sample.

High learning rates with unlucky batches of training data can lead to exploding loss and performance collapse for all algorithms. With too-small learning rates, agents tend to get stuck in bad local minima of the loss function, which causes performance stagnation. For the DQN algorithm on TSP, relatively high learning rates at the beginning of training seem to help the agent to reach stable policy parameterizations faster. Cyclic learning rate schedules can help algorithms escape bad local minima of the loss functions and discover better policies, but agents have difficulties recovering from phases of high learning rates. Exponential learning rates let the algorithms take more precise steps towards local minima but can cause performance stagnation. Adjusting the loss function in PG for environments with strictly negative rewards as suggested on Stack Exchange is unsuccessful, and the regular loss formula should hold for negative rewards as well. SAC's performance with learned entropy temperature is susceptible to the temperature learning rate. A2C and PPO perform almost identically on the TSP, and the benefits of PPO's objective clipping are only visible with high learning rates. For methods with Generalized Advantage Estimation, lambda values of 1.0 are most effective for TSP of size 20, likely because of the relatively short trajectories and resulting low variance.

References

[1] Andrychowicz M, et al. 2020. What Matters in On-Policy Reinforcement Learning? A Large-Scale Empirical Study. *arXiv preprint arXiv:2006.05990*.

[2] Haarnoja T, et al. 2018. Soft Actor-Critic Algorithms and Applications. *arXiv preprint arXiv:1812.05905*.

[3] Ilyas A, et al. 2018. Are Deep Policy Gradient Algorithms truly Policy Gradient Algorithms? *arXiv preprint arXiv:1811.02553*. URL: `https://arxiv.org/pdf/1811.02553v3.pdf`.

[4] Kool W, Van Hoof H, Welling M. 2018. Attention, learn to solve Routing Problems! *arXiv preprint arXiv:1803.08475*.

[5] Mnih V, et al. Asynchronous Methods for Deep Reinforcement Learning. In: *International Conference on Machine Learning*. PMLR. 2016, pp. 1928–1937.

[6] Mnih V, et al. 2015. Human-Level Control through Deep Reinforcement Learning. *Nature* 518(7540): 529–533.

[7] Rennie SJ, Marcheret E, Mroueh Y, Ross J, Goel V. Self-Critical Sequence Training for Image Captioning. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 7008–7024.

[8] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.

[9] Van Hasselt H, Guez A, Silver D. Deep Reinforcement Learning with Double Q-Learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 2016.

[10] Ye D, et al. Mastering Complex control in Moba Games with Deep Reinforcement Learning. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 6672–6679.

DQNPolicy

model: `torch.nn.Module`
optim: `torch.optim.Optimizer`
discount_factor: `float` = 0.99
estimation_step: `int` = 1
target_update_freq: `int` = 0
reward_normalization: `bool` = False
is_double: `bool` = True
****kwargs**: `Any`

PPOPolicy

actor: `torch.nn.Module`
critic: `torch.nn.Module`
optim: `torch.optim.Optimizer`
dist_fn: `torch.distributions.Distribution`
discount_factor: `float` = 0.99
eps_clip: `float` = 0.2
dual_clip: `float` = 5.0
value_clip: `bool` = True
advantage_normalization: `bool` = True
recompute_advantage: `bool` = False
vf_coef: `float` = 0.5
ent_coef: `float` = 0.01
max_grad_norm: `float` = None
gae_lambda: `float` = 0.95
eps_clip: `float` = 0.2
reward_normalization: `bool` = False
max_batchsize: `int` = 256
action_scaling: `bool` = True
action_bound_method: `str` = "clip"
action_space: `gym.Space` = None
lr_scheduler: `torch.optim.lr_scheduler` = None
deterministic_eval: `bool` = False
****kwargs**: `Any`

PGPolicy

model: `torch.nn.Module`
optim: `torch.optim.Optimizer`
dist_fn: `torch.distributions.Distribution`
discount_factor: `float` = 0.99
action_scaling: `bool` = True
action_bound_method: `str` = "clip"
action_space: `gym.Space` = None
lr_scheduler: `torch.optim.lr_scheduler` = None
deterministic_eval: `bool` = False
****kwargs**: `Any`

DiscreteSACPolicy

actor: `torch.nn.Module`
actor_optim: `torch.optim.Optimizer`
critic1: `torch.nn.Module`
critic1_optim: `torch.optim.Optimizer`
critic2: `torch.nn.Module`
critic2_optim: `torch.optim.Optimizer`
tau: `float` = 0.005
gamma: `float` = 0.99
alpha: `Union[float, Tuple[...]]` = 0.2
reward_normalization: `bool` = False
exploration_noise: `tianshou.[...].BaseNoise` = None
action_scaling: `bool` = True
action_bound_method: `str` = "clip"
action_space: `gym.Space` = None
lr_scheduler: `torch.optim.lr_scheduler` = None
deterministic_eval: `bool` = False
****kwargs**: `Any`

A2CPolicy

actor: `torch.nn.Module`
critic: `torch.nn.Module`
optim: `torch.optim.Optimizer`
dist_fn: `torch.distributions.Distribution`
discount_factor: `float` = 0.99
vf_coef: `float` = 0.5
ent_coef: `float` = 0.01
max_grad_norm: `float` = None
gae_lambda: `float` = 0.95
reward_normalization: `bool` = False
max_batchsize: `int` = 256
action_scaling: `bool` = True
action_bound_method: `str` = "clip"
action_space: `gym.Space` = None
lr_scheduler: `torch.optim.lr_scheduler` = None
deterministic_eval: `bool` = False
****kwargs**: `Any`

Figure 1: Class constructor parameters with types and default values for all Tianshou policies applied in this paper.

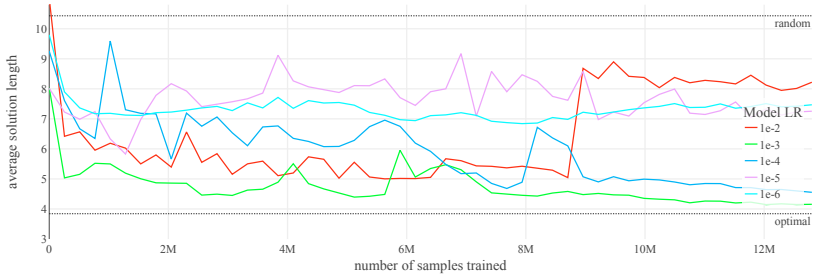

Figure 2: Performance throughout the training of 5 DQN agents with different learning rates.

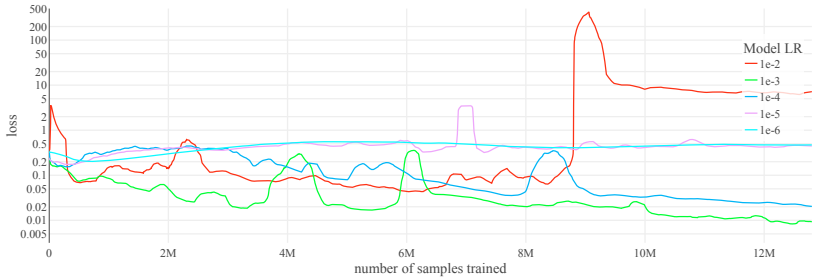

Figure 3: Model loss throughout the training of five DQN agents with different learning rates.

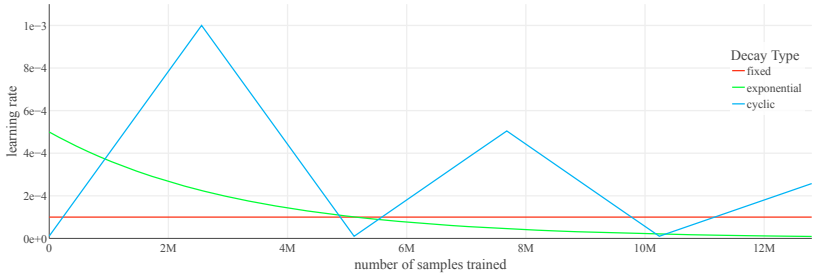

Figure 4: Comparison of learning rate developments throughout training for different learning rate schedules.

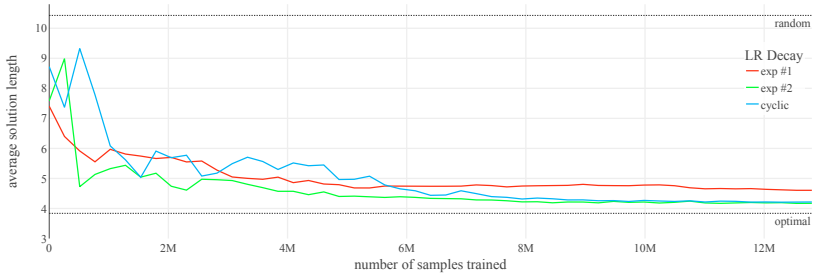

Figure 5: Performance throughout the training of three DQN agents with different learning rate schedules.

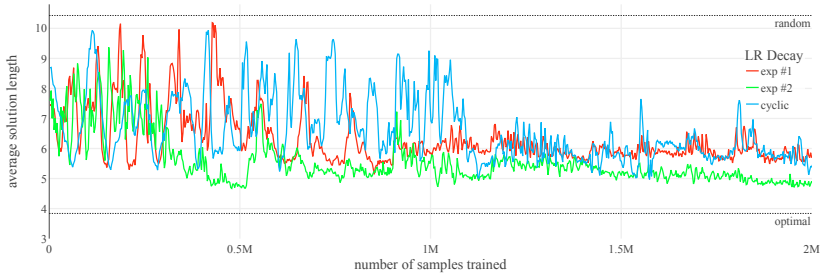

Figure 6: Section of the performance over the first two million training samples of three DQN agents with different learning rate schedules. Each data point represents the average solution length over 128 training problem instances.

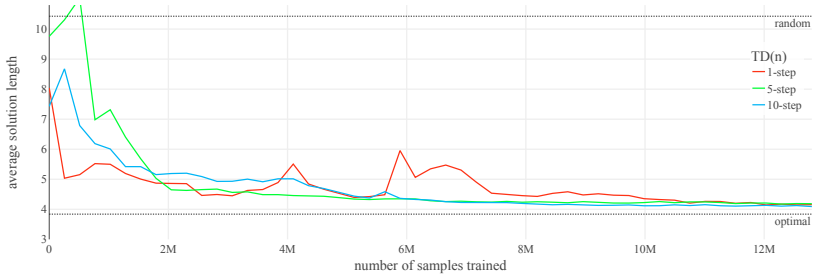

Figure 7: Performance throughout the training of three DQN agents with different n-step updates.

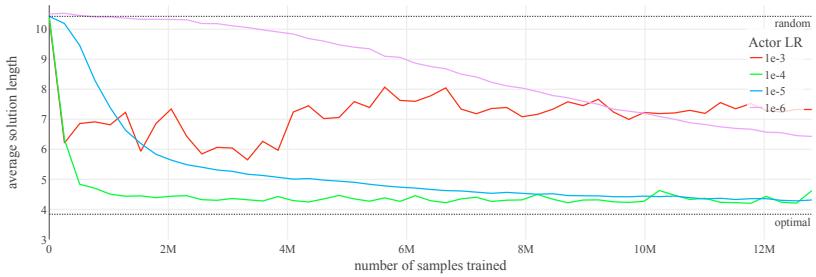

Figure 8: Performance throughout the training of four PG agents with different learning rates.

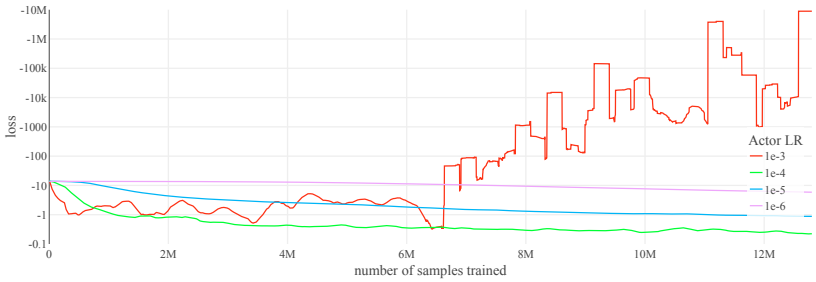

Figure 9: Actor loss throughout the training of four PG agents with different learning rates.

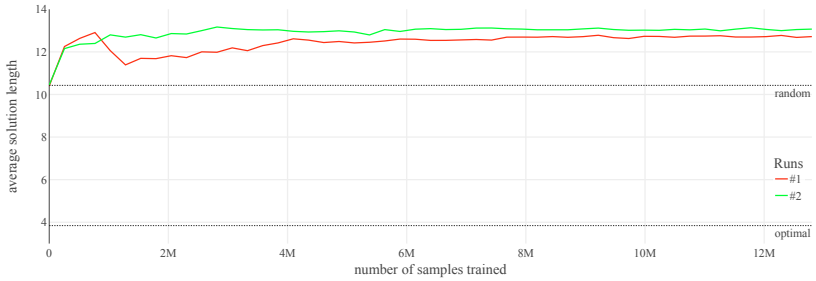

Figure 10: Performance throughout training of two PG agents adjusted objective for strictly negative rewards as suggested on Stack Exchange.

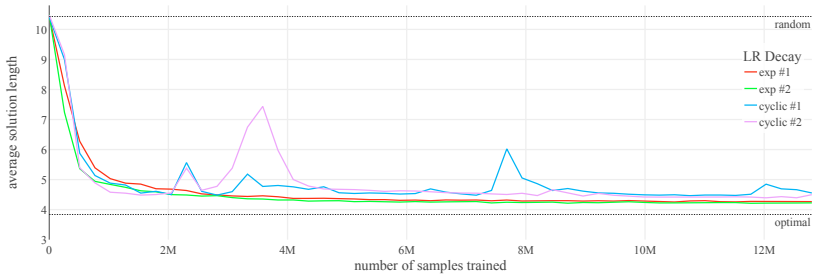

Figure 11: Performance throughout the training of four PG agents with different learning rate schedules.

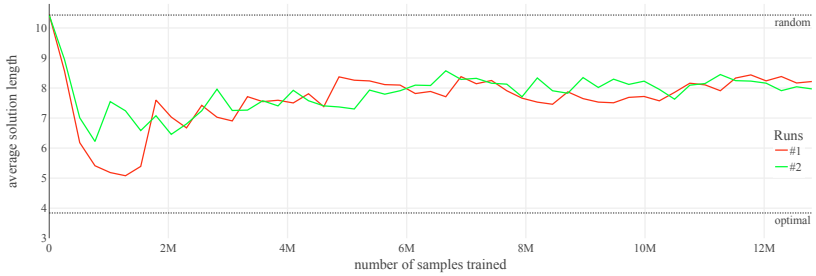

Figure 12: Performance throughout the training of two PG agents with batch normalization and the number of training environments set smaller than the number of episodes considered per network update.

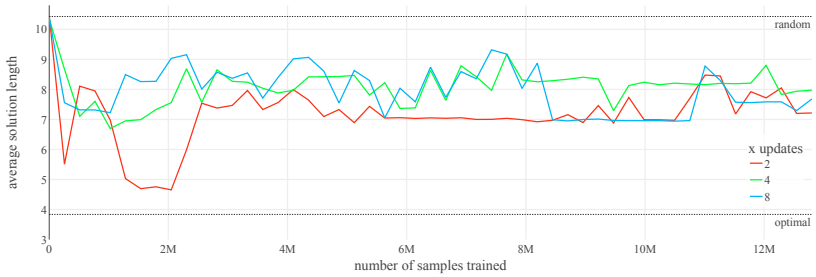

Figure 13: Performance throughout the training of three PG agents with different `repeat_per_collect` values, which determine the number of network updates per batch of data.

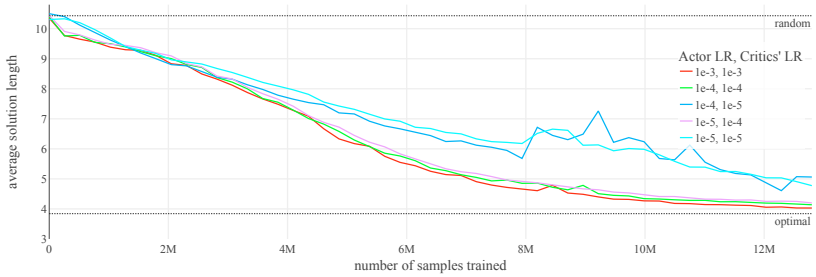

Figure 14: Performance throughout the training of five SAC agents with different learning rates.

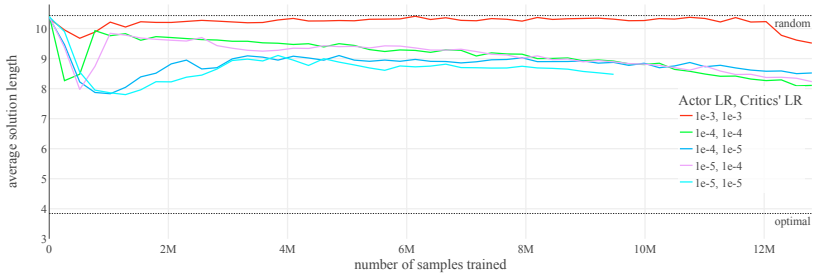

Figure 15: Performance throughout the training of five SAC agents with differing actor and critic learning rates. Entropy temperature learning is enabled with a fixed, low learning rate.

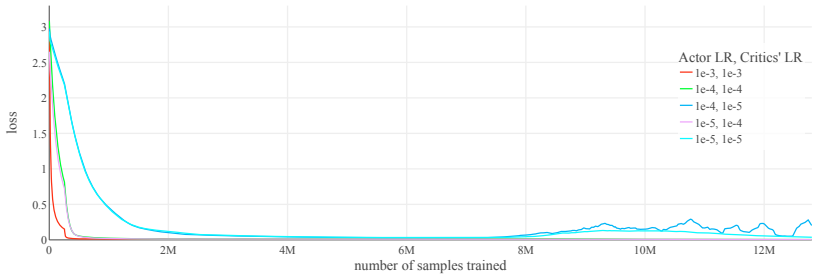

Figure 16: Critic losses throughout the training of five SAC agents with different learning rates.

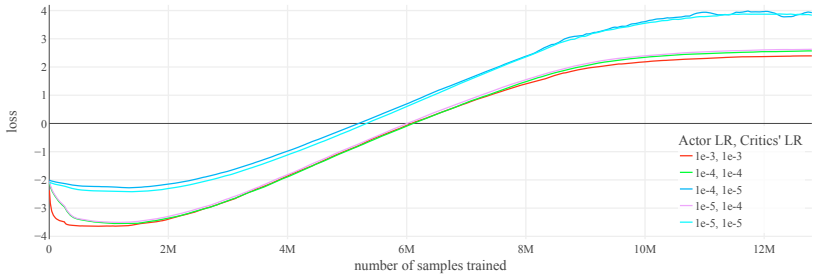

Figure 17: Actor loss throughout the training of five SAC agents with different learning rates.

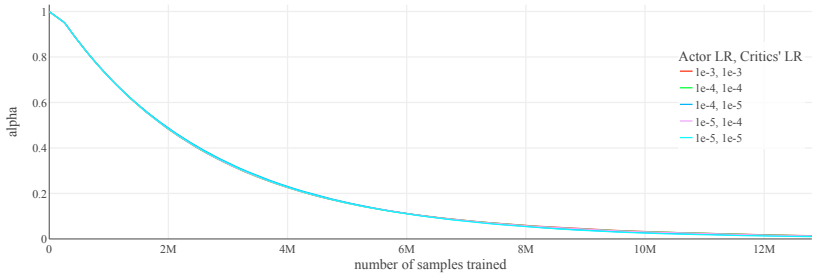

Figure 18: Almost identical development of the entropy temperature parameter alpha throughout the training of five SAC agents with different learning rates and temperature learning.

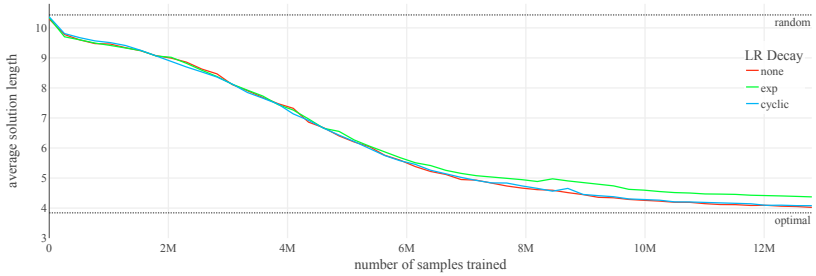

Figure 19: Performance throughout the training of three SAC agents with different learning rate schedules.

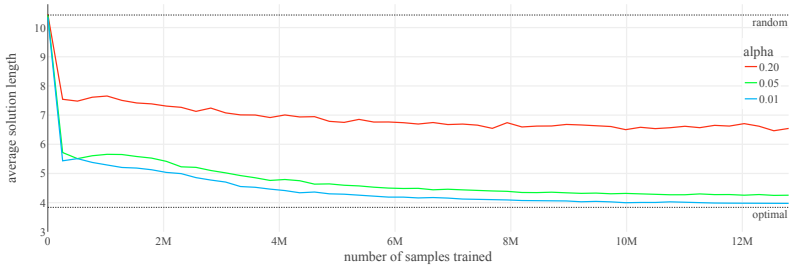

Figure 20: Performance throughout the training of three SAC agents with fixed entropy temperature values (alpha).

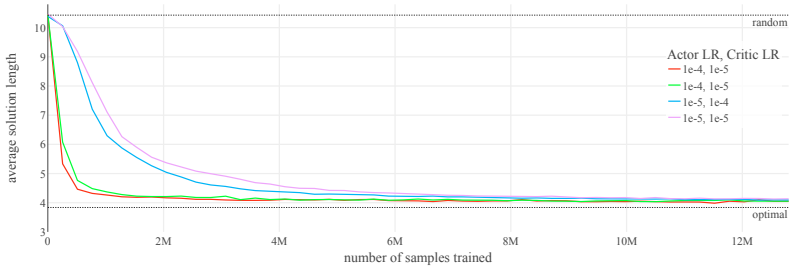

Figure 21: Performance throughout the training of four A2C agents with different learning rates.

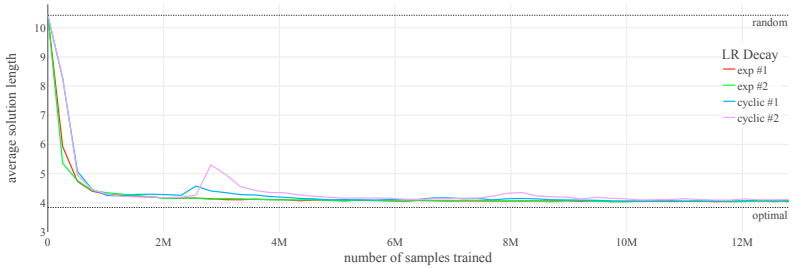

Figure 22: Performance throughout the training of four A2C agents with different learning rate schedules.

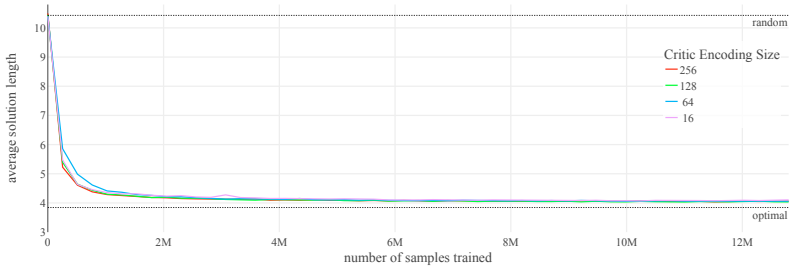

Figure 23: Performance throughout the training of four A2C agents with different embedding sizes specified for the critics.

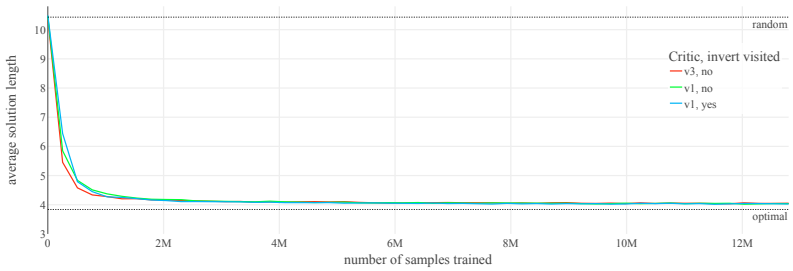

Figure 24: Performance throughout the training of three A2C agents, comparing the effect of inverting the 0/1 encoded vector of visited nodes.

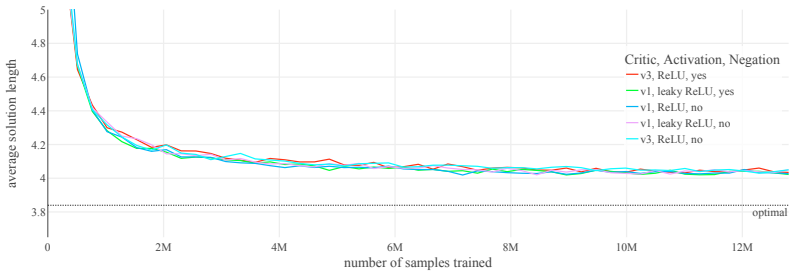

Figure 25: Performance throughout the training of five A2C agents, comparing the effect of different critic architectures, activation functions, and output negation.

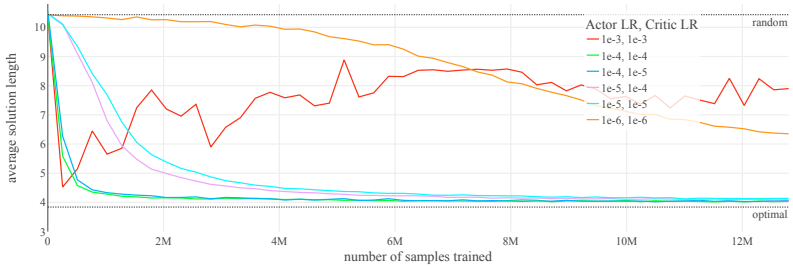

Figure 26: Performance throughout the training of six PPO agents with different learning rates.

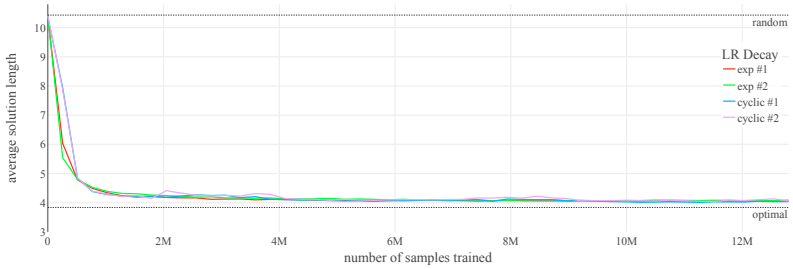

Figure 27: Performance throughout the training of four PPO agents with different learning rate schedules.

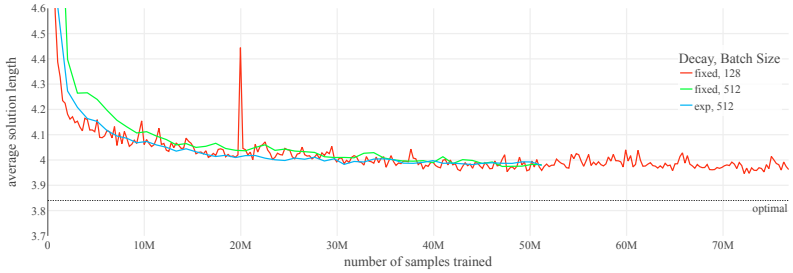

Figure 28: Magnified performance throughout the training of three PPO agents with different learning rate decays and batch sizes.

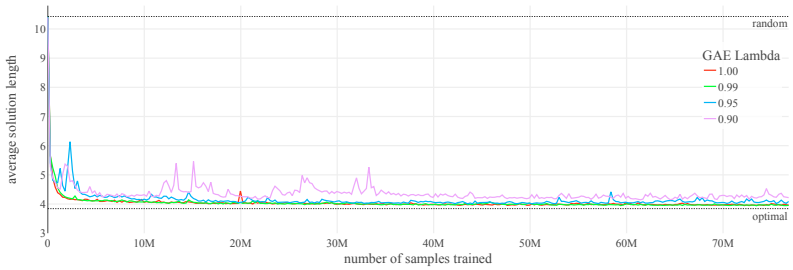

Figure 29: Performance throughout the training of four PPO agents with different GAE lambda values.

Figure 30: Performance throughout the training of four of Kool, Van Hoof, Welling's agents with different learning rates.

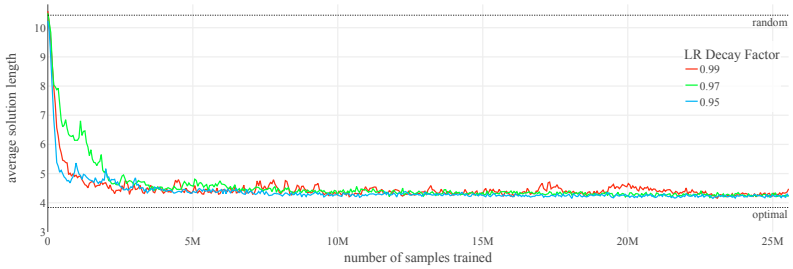

Figure 31: Performance throughout the training of three of Kool, Van Hoof, Welling's agents with different exponential learning rate decays.

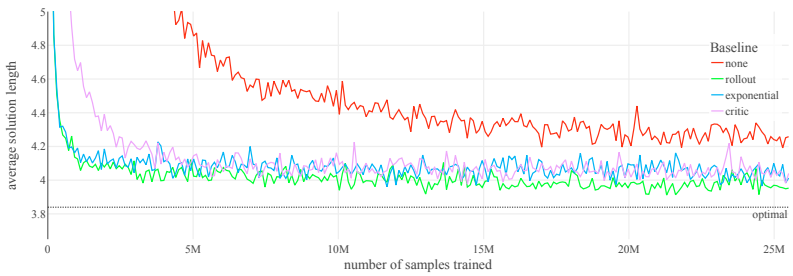

Figure 32: Magnified performance throughout the training of four of Kool, Van Hoof, Welling's agents with different baselines.