Federated Reinforcement Learning Has Linear Speedup

Michael Einhorn

July 17, 2023

1 Abstract

Whether Federated Reinforcement Learning has linear speedup is an important question for determining how to allocate resources when training models. Previous work has shown that Federated Supervised Learning has linear speedup, and that Federated Reinforcement Learning works, but have not measured the speedup for Federated Reinforcement Learning empirically. We tested the relationship between speedup, the learning rate, the number of clients, and the synchronization frequency, partially confirming prior theoretical predictions for Federated Tabular Q-learning. These results empirically showed that Federated Tabular Q-learning and Federated PPO have linear speedup.

2 Introduction

2.1 Reinforcement learning

Reinforcement Learning is a paradigm of Machine Learning where an agent learns from experience on making decisions in an environment. Q-learning is a simple algorithm where the agent learns a q value function for every action in every state. To learn the agent either takes the action with maximum q value in its state, or takes a random action, and then updates its q function using the reward it received and the next state. The q function can either be a table, or a function approximation such as a linear model or a neural network [13]. The update rule for Q-learning is $q(s,a) \leftarrow q(s,a) + \alpha(r + \gamma \max_{a'} q(s',a') - q(s,a))$. The advantages of Q-learning are it has been studied for a while, is easy to implement and explain, though there are newer methods that perform better. In a small discrete environment, and tabular Q values, the optimal q values can be computed using the Bellman Optimality Equation, and can be compared with the learned q values. $q_*(s,a) = \Sigma_{s',r} p(s',r|s,a)(r + \gamma \max_{a'}(q_*(s',a'))$ [13].

Another class of methods are Policy Gradient algorithms. Instead of acting greedily based on Q value, this method can use a neural network to approximate the policy. The policy network outputs action selection probabilities for an input state, which lets it model stochastic policies. A gradient of the policy w.r.t. to the reward can be estimated using sample trajectories, and then used to optimize the policy [11]. PPO (Proximal Policy Optimization) is based on this Policy Gradient approach which uses a policy function and estimate of the state value function. The state value function is optimized to predict expected discounted returns, and the policy is optimized from the value function. These methods alternate between acting in the environment to collect data and optimizing a new policy on the data collected by the old policy, with constraints on how different the policies can be [12].

2.2 Federated Learning

Most of the work in Federated Learning is on the supervised learning paradigm. Federated Learning trains one large model from distributed data and compute without sharing the dataset with other clients. Clients train their model, and the server aggregates the models and sends updates back to clients. Two important considerations are stability of convergence which is if the global model will still find a local optimum, and if the speedup is linear, which means for N clients working in parallel the model will converge N times faster. Unbalanced data between clients and bandwidth limits can cause issues for Federated Learning. Bandwidth needed can be reduced with structured and sketched updates. Structured updates limit learning to a subset of features each round to reduce communication, and sketched updates learn with the whole model but compress the update before communicating [6]. These considerations of federated supervised learning seem like they will also apply to federated reinforcement learning. The advantages of federated learning are preserving data privacy, speedup of training models, and makes deep learning more accessible for edge devices and individuals [9]. FedAvg is a method where this aggregation is averaging weights by clients to construct the global model, and it is the most used federated learning algorithm. It is both theoretically proven and empirically tested that FedAvg convergence is sped up linearly by the number of active participants [9]. To our knowledge there is not a similar study on linear speedup for federated reinforcement learning.

2.3 Federated Reinforcement Learning

Federated Reinforcement Learning is a combination of these two ideas, and there is little existing work in this area. One approach is FedRL, which maintains data privacy where sharing datasets or individual models is not an option [14] There are multiple agents which can't share their state, and all using their own model. Agents receive gaussian differentials of the Q network outputs from other agents. This allows a global Q network to be constructed without any agent learning about the individual Q network of another agent. Each agent views the other agents' Q network outputs as a constant to update its own. After communication, there is a federated Q network for each agent [14]. It does not appear there is a central server maintaining a global aggregated network unlike FedAvg, so it has stronger privacy. Two Categories of Federated Reinforcement Learning are Horizontally Federated Reinforcement Learning (HFRL) and Vertically Federated Reinforcement Learning (VFRL). HFRL is where each agent has its own independent environment to act in, and information is shared between agents to explore more possible states. Each agent has similar actions and environment, and aggregate into a global model. VFRL is where there is a global environment that each agent partially observes, and may have different actions in. Actions of agents affect each other, and the areas they are able to observe may partially intersect. While each agent acts differently and pursues different rewards, sharing information gives them a more general model of the environment [8]. It seems FedRL method used above is a VFRL approach, though Zhuo et al. do not use this term in the paper [14]. HFRL sounds more similar to the FedAvg method used for supervised learning. This paper does describe that there is a speedup, but not if it is linear [8].

2.4 Contribution

We tested Federated Reinforcement Learning for linear speedup on a variety of problems from tabular Q-learning in a Markov Decision Process (MDP) to PPO on small neural networks in Coin Run. We experimentally tested a theoretical proof of linear speedup in simple tabular Q-learning cases, and if this extends to more complex environments and PPO. The result of this research is general knowledge on how Federated Reinforcement Learning behaves and will aid in designing large scale reinforcement learning experiments to forecast required resources and time, make the most efficient use of federated training.

3 Methods

3.1 Toy Models

In an MDP, there are a finite number of states and actions, a transition matrix with the probability of getting to each next state S' after taking action A in state S, and a reward for each state. The MDP used in this experiment has 4 states and 3 actions (Fig. 1) and has an optimal policy of a q table that can be calculated with the Bellman Optimality Equation. $q_* = \Sigma_{s',r} p(s', r|s, a)(r + \gamma \max_{a'} q_*(s', a'))$ [13].
A random transition matrix will tend to have an optimal policy which is barely better than randomly selecting actions, so the probability distribution needs to be less uniform. The method we used was generating normally distributed values, and raising them to a power. The higher the power is, the less uniform the transition matrix will be. One of the reasons for using this specific method is that it is continuous, instead of making a subset of the state action pairs have deterministic transitions. Softmax is then used to turn the values into a valid probability distribution.

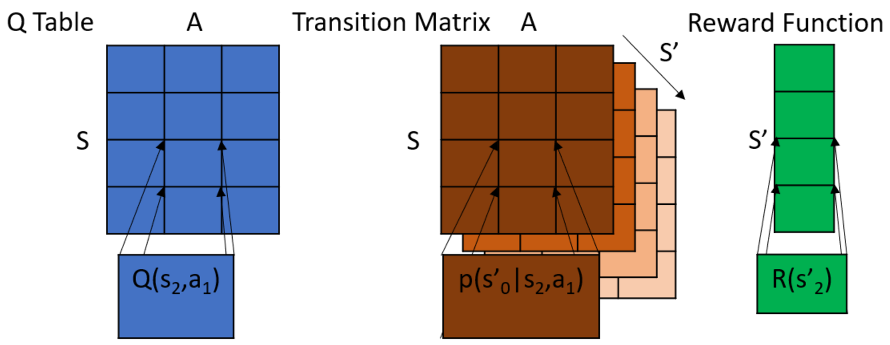


Figure 1: Q Table and MDP Environment with 4 States and 3 Actions

Tabular Q-learning is an reinforcement learning algorithm, which takes actions exploring in the environment and updates a q table based on the observed outcome [13]. For an environment with n states and m actions, the model has n * m parameters. The policy can either be deterministic or stochastic by using argmax or softmax. Epsilon greedy exploration is taking a random action with probability epsilon, otherwise use the policy. Training is stopped when the learned model is within a certain distance of the optimal model, or the distance stops improving. Our implementation is by running Q-learning separately on each CPU core, and averaging the weights together after a fixed number of steps K, similar to FedAvg [9]. The main hyperparameters are N, the number of agents running in parallel, K the synchronization frequency which is how often the weights are averaged, and α the learning rate. In the limit of decreasing the synchronization frequency this is equivalent to training multiple models independently. While this does not simulate some of the challenges with networking such as communication time, and different clients running at different speeds, the number of both computation and communication steps are counted for comparison. The final model is the thread which stopped first. This is tested on the PACE cluster for different combinations of these hyperparameters [7].

There will be some tradeoff between communicating as frequently as possible, and the time for communication. Other questions are if learning converges to a better optimum regardless of the number of steps, or can tolerate a higher learning rate and still get to the same optimum.

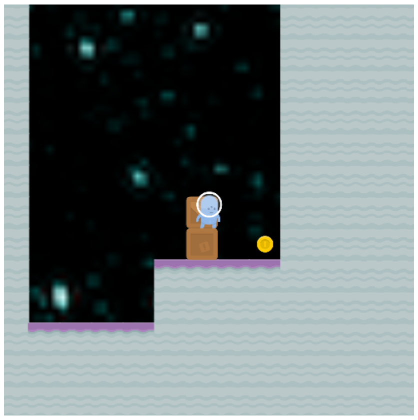
If there is linear speedup with the number N of models, then there should be a decrease of a factor of N of the computation steps to reach a fixed error. This may only happen at good values for the learning rate alpha, and a high communication frequency. To test if this relationship is linear the data is graphed on a log log plot with linear regression. On a log log plot, the slope b of the line represents the exponent of the power law ax^b. If the slope of the line in log log space is 1, that means the relationship is linear, and if it is -1 it is linear to the inverse.

$$D^2 \propto (1-\alpha)^T + \alpha/N + (K-1)\alpha^2 \tag{1}$$

For tabular Q-learning in an MDP, the expected error bound is (EQ 1) [5]. where D is distance to the bellman q table, T is time, α is the learning rate, N is the number of agents, and K is the synchronization frequency. Linear speedup with N is caused by being able to increase α by a factor of N while keeping the α/N term constant. For more complex environments reward is used as the metric since there is not an error from a known optimal solution.

3.2 PPO in Procgen

PPO is the state of the art reinforcement learning algorithm for environments with discrete action spaces. It is an actor-critic method which learns both a state value function, and policy function. One of the main advantages is that it gives a loss function which can be used by standard first order optimizers such as Adam [12]. Coin Run and other procedurally generated games in Procgen, are common baseline environments to test neural network models [1].


(a) Coin Run [1]

(b) Bigfish [1]

The model is given the image for the current frame and outputs the action that it takes. Tested both IMPALA [3] and Vision Transformers [2] on coin run, but both were getting stuck at around 0.5 reward per episode, corresponding to getting the coin half of the time. The models were learning to run right to the coin and jump over the terrain, but were not learning to avoid the enemies so did not get the coin on the harder levels. Bigfish is an easier environment to learn because there are small rewards for each fish, in addition to the winning reward. IMPALA trained on Bigfish was able to learn and smoothly increase in reward, so this model and environment was used.
Actions are sampled from the softmax of the logits the model outputs. For exploration, an entropy bonus loss of 10^{-2} is used. 64 environments are run in

parallel, for 256 steps for every epoch.
The implementation is by running N copies of the model each with their own 64 environments on the same GPU, and averaging the weights every K epochs. Similar to the experiment with tabular Q-learning the network is simulated, and the number of communications and the amount of data transferred are measured. Every model has it's own Adam optimizer object, but the forward and backward passes are run in parallel with the model index like a second batch dimension. Due to limited GPU time, the hyperparameter range tested on procgen is smaller than for tabular Q-learning and only for a single value of $\alpha = 2.5 * 10^{-4}$. $N = 1, 2, 4, 8$ and $K = 1, 10, 100$.
Unlike tabular Q-learning on simple MDPs, there is no known optimal solution for PPO with a neural network on procgen environments. Optimizers like Adam find local minimums, and it was infeasible to run every experiment until it reached steady state reward. Because of this, the limiting behavior of N and K on reward as time goes to infinity was not tested, and instead only the time to reach a reward threshold was measured for different N and K.

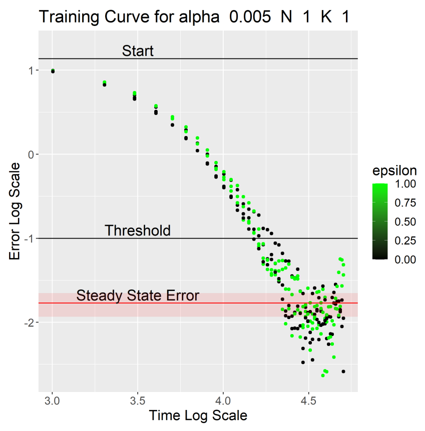
4 Results

4.1 Tabular Q-learning

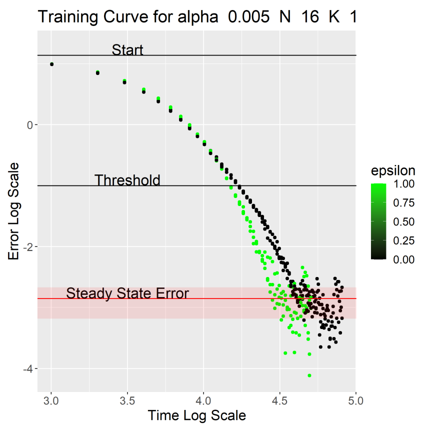
4.1.1 Training Curves

Data for each training run includes the number of computation and communication steps to convergence. These are a few of the learning curves of distance from the reference model. N is the number of agents, Alpha is the learning rate, K is the synchronization frequency, and epsilon is the epsilon greedy exploration parameter. Each graph shows 6 separate learning curves, 3 with epsilon 0 and 3 with epsilon 1. These learning curves show a smooth decline in error until reaching a steady state error where it will then fluctuate. Steady state error is measured by the average of the last 10 data points in a learning curve, the red line and region above are the mean and standard deviation of the steady state error for the 6 learning curves on each graph. The red region looks asymmetric because the average error was calculated in linear space while the graph is in log space. The learning curves for alpha = 0.005, all reach the steady state error at about $10^{4.5}$ steps, but for $N = 1, K = 1$ (Fig. 3a) and $N = 16, K = 10^4$ (Fig. 3c), the steady state error is about $10^{-1.8}$, and for N = 16, K = 1, the steady state error is around $10^{-2.9}$ (Fig. 3b). Note that with $K = 10^4$ by $10^{4.5}$ steps there will have only been about 3 synchronizations. The top horizontal line is at the value of the error at time step 0. The theshold line at 0.1 error is used to evaluate linear speedup, since most training runs reach this threshold before the steady state error region. Raising the learning rate by a factor of 16 for $N = 16$ (Fig. 3d) reaches the a similar steady state error of $10^{-1.7}$ as the run with $N = 1$ (Fig. 3a), but much faster.

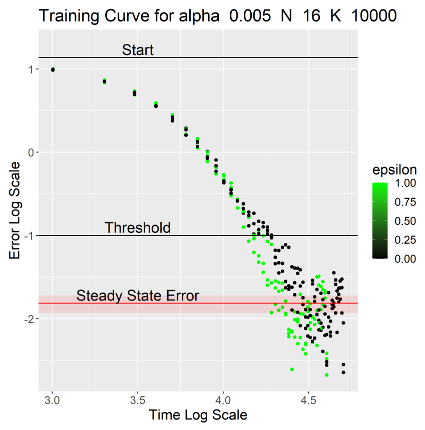
To validate the environment the optimal policy from the Bellman Optimality Equation is compared with a random policy. If the optimal policy is not



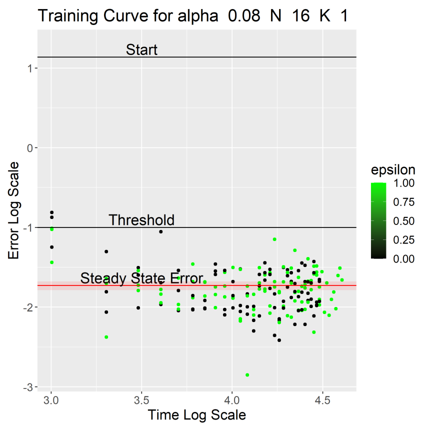
(a)



(b)



(c)



(d)

Figure 3

significantly better than a random one, then it is not a good environment to test learning on. For the MDP used here, the random action policy gets a cumulative reward of 1275, and the optimal policy's cumulative reward is 3913, when using an argmax policy to select actions. This means a learned policy can significantly improve in performance from initialization.

4.1.2 Steady State Error α/N Term

To test the α/N Term, in (EQ 1), steady state error is used eliminating the 1st term, and K=1 is used eliminating the 3rd term. The bound now has the form $D^2 \propto \alpha/N$. A linear regression between $\log(D^2)$ and $\log(\alpha/N)$ gives a slope of 0.997, confirming that the relationship between D^2 and α/N is linear. This line is graphed with a 95% confidence interval of the regression's estimate (Fig.

4). In a separate regression between $\log(D^2)$ and $\log(\alpha)+\log(N)$, the slope for the α term was 0.994, and the slope for the N term was -1.012, confirming the linear relation with α and inverse relation with N.

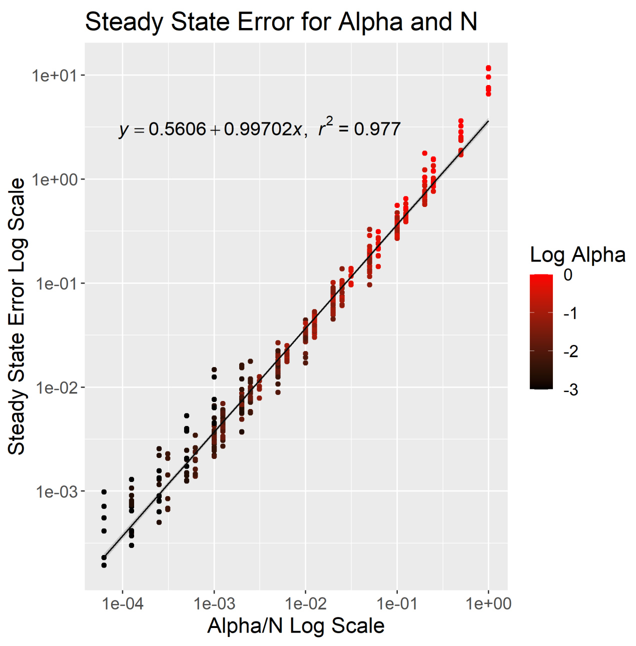


Figure 4

4.1.3 Steady State Error $(K-1)\alpha^2$ Term

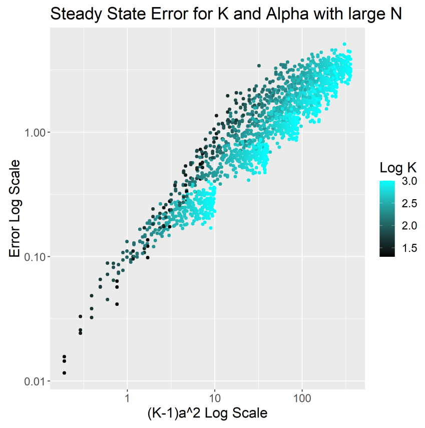
When N is 256 and alpha is large, the α/N term becomes relatively small compared to $(K-1)\alpha^2$, and using steady state error eliminates the 1st term.

$$D^2 \propto \alpha/N + (K-1)\alpha^2$$
$$\log D^2 \propto \log(\alpha/N + (K-1)\alpha^2)$$

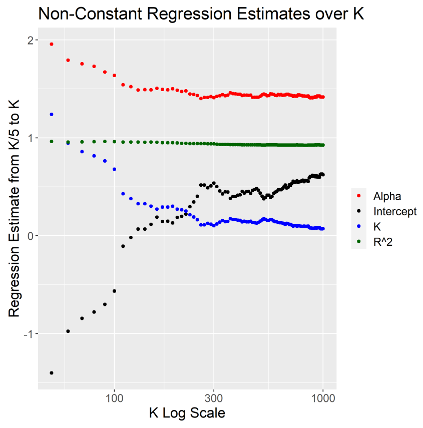
$$\log D^2 \propto \log(K-1) + \log(\alpha) + \log(1 + 1/(N\alpha(K-1)) \qquad (2)$$

Factoring out $(K-1)\alpha^2$ gives a different form of the bound (EQ 2) allowing for the estimation of the exponents for K and α in the $(K-1)\alpha^2$ term with linear regression.

This graph is for data with $N = 256$, α from 0.1 to 0.6 with increments of 0.1, and K from 20 to 1000 with increments of 10. The slope for $\log(K-1)$ is 0.207, and the slope for $\log(\alpha)$ is 1.378 (Fig. 5a). Though these slopes are not constant with K. When taking multiple regressions of subsets of the data between $K/5$ and K, the slope for $\log(K-1)$ starts at a bit over 1, and decreases to close to 0. The slope of $\log(\alpha)$ starts at just under 2, and decreases to a bit under 1.5. For all of these regressions R^2 remains good at above 0.9. (Fig. 5b). The $(K-1)\alpha^2$ term has lower exponents than expected for both $(K-1)$, and α, and these exponents vary with K, so a power law may not be the correct relation.



(a)



(b)

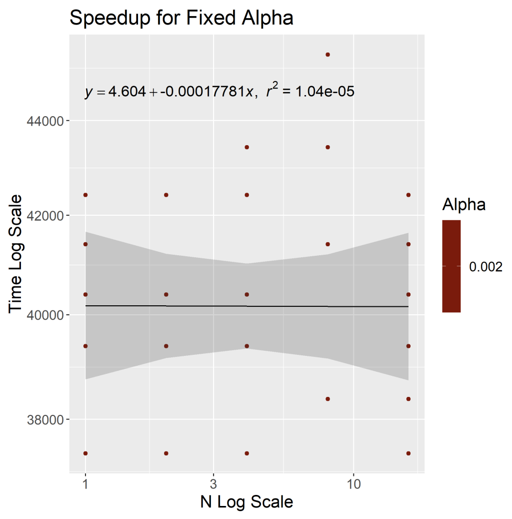
Figure 5

4.1.4 Linear Speedup

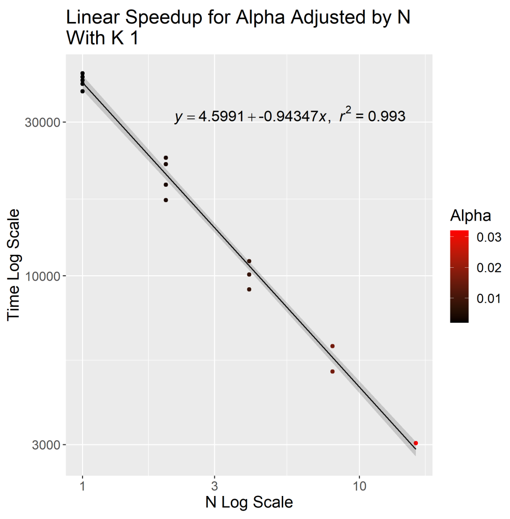
To test for linear speedup, data for a single value of α and different values of N is used. As expected since the first term in the bound (EQ 1) is independent of N, there is no speedup from increasing N when α is fixed, however it does reduce steady state error. Linear speedup in this case is dependent on scaling α by N while getting the same steady state error.

$$D^2 \propto (1–\alpha_1 N)^T + \alpha_1 + (K-1)(\alpha_1 N)^2 \tag{3}$$

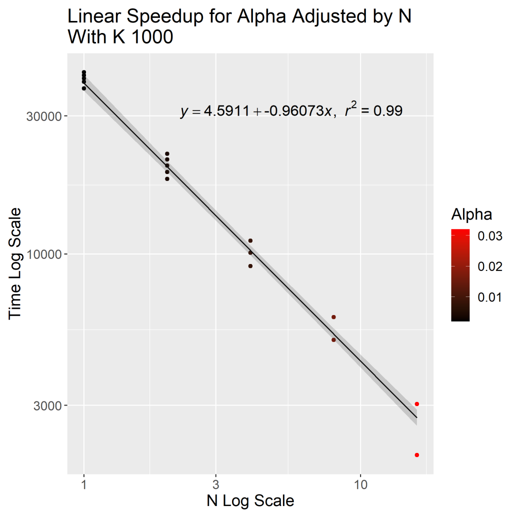
Using a learning rate $\alpha = a_1 N$ makes the the bound have this form (EQ 3). Unlike the original form of the equation, the first term is now dependent on N. The steady state error is independent of N when $K = 1$, and when K is larger than 1, the steady state error will increase with N. For higher K this will reduce the scale factor of α that federation is able to tolerate while maintaining a constant steady state error, though it will reduce the communication complexity. For this graph with $\alpha = 0.002N$, and $K = 1$, the slope of the line is -0.944, which is almost linear to $1/N$ (Fig. 6c). When $K = 10^4$, this slope drops to -0.855 (Fig. 6d). These lines are graphed with a 95% confidence interval of the regression's estimate.



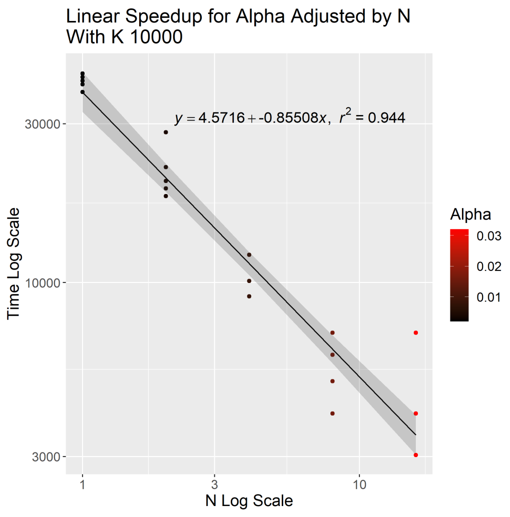
(a)



(b)

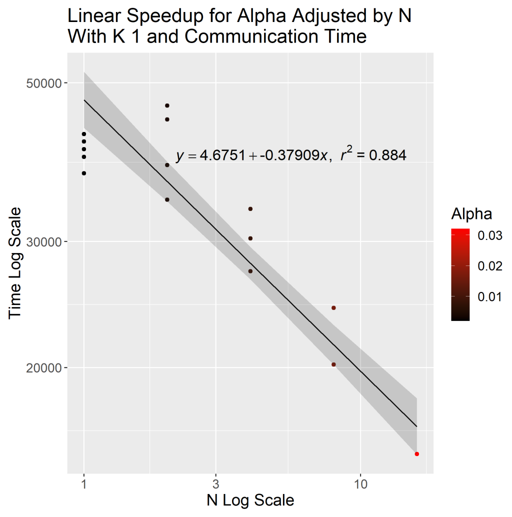


(c)

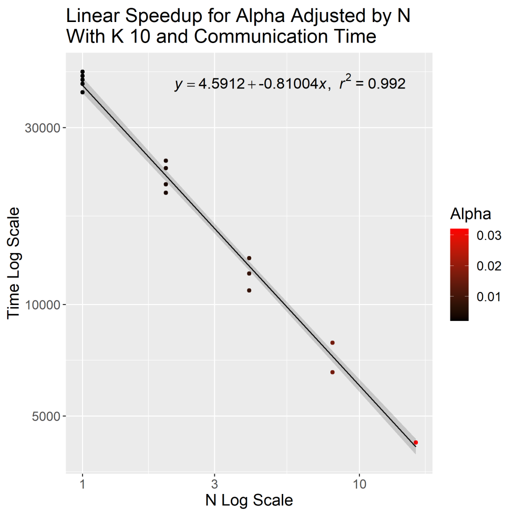


(d)

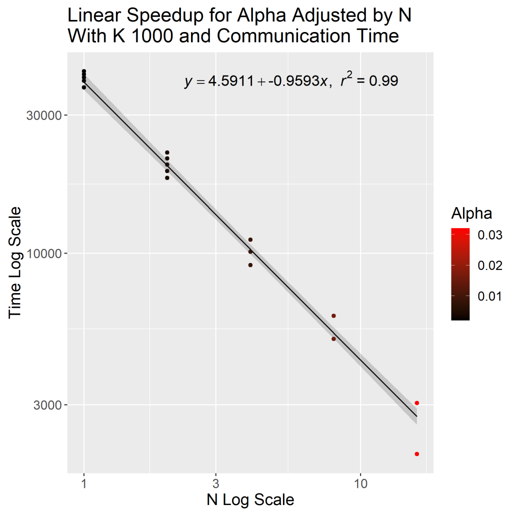
Figure 6



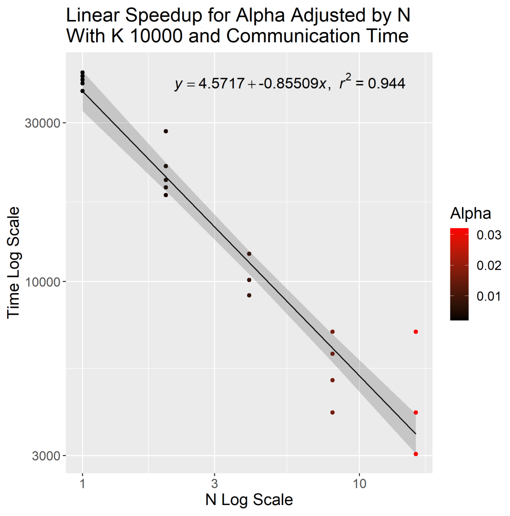
(a)



(b)



(c)



(d)

Figure 7: Speedup including $\log_2(N)$ communication complexity.

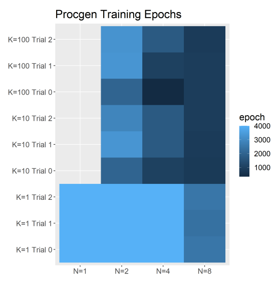
4.1.5 Linear Speedup with Communication Time

While this shows a linear speedup in the number of computation steps, it comes at a trade off with the number of communication steps, and increasing N increases the communication complexity of each step. The complexity of all reduce is $O(log(N))$, so the total communication time is $O((T//K) * \log_2(N))$, the graphs below use a formula for total time including communication of $T_c = T + (T//K) * \log_2(N)$. The first term is the computation time and the second is communication time where both are weighted equally, but the real weight will depend on the hardware used, and how hard to sample the environment is. With $K = 1$ (Fig. 7a) the slope is reduced to only 0.38 which is a significantly sublinear speedup. This increases for $K = 10$ (Fig. 7b) and $K = 100$, up to $K = 1000$ (Fig. 7c) for a slope of 0.96, which is almost linear, and reduces to 0.86 for $K = 10000$ (Fig. 7d).

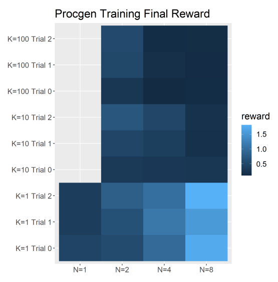
4.2 PPO in procgen

4.2.1 Dataset

For each value of N and K, there are 3 trials. These training runs did not run for a consistent quantity of time so the final rewards are not directly comparable. Training runs with lower N were run for a larger number of epochs, both because running multiple models is slower on 1 GPU, and there is a speedup expected for the number of epochs. For most training runs this was not an issue, but especially for $K = 100$, the training runs for larger N could have been run for longer (Fig. 8).



(a)



(b)

Figure 8: Procgen Dataset Visualization

4.2.2 Linear Speedup for N

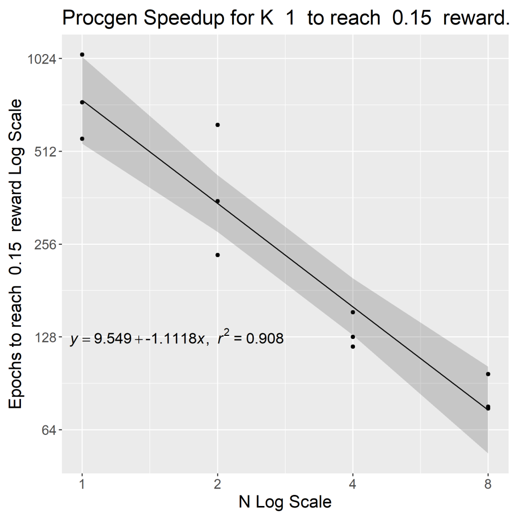
To test for linear speed up different values of N are graphed against time to reach a reward threshold on a log log plot for a fixed value of K. The slope of the linear regression corresponds with the exponent of the scaling in real space. If the exponent of N is -1, then speedup is linear.

On the following graphs, black datapoints are the time where the model reached the reward threshold for the graph. Red datapoints are if the model never reached the reward threshold how long was the model trained for. If the red datapoints are above the trendline it is safe to say that the trendline does not predict the result, but if it is below the trendline then the model may have just needed to train for longer to reach the threshold.

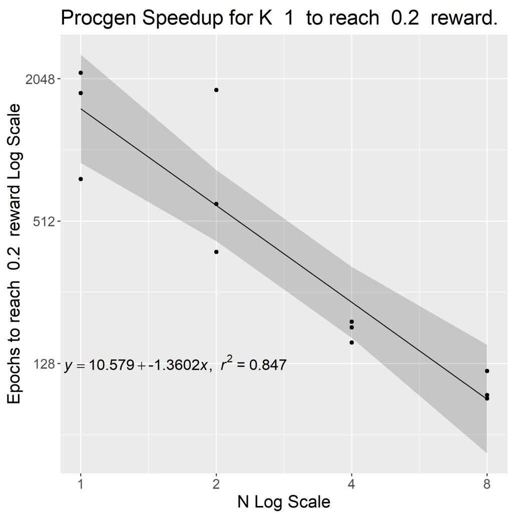
For $K = 1$, the speedup to reach different thresholds of reward is super-linear, with estimates for the slope ranging from -1.11 to -1.47 (Fig. 9). For the threshold of 0.1 reward speedup is sublinear at 0.33, but this is likely because 0.1 is too close to the reward at initialization.

For $K = 10$, speedups are sublinear with slopes ranging from -0.28 to -0.78 (Fig. 10). For the threshold of 0.5 reward there are too many missing points marked in red to make conclusions (Fig. 10d). For all of these thresholds, the data at $N = 8$ is clustered above the trendline, and $N = 4$ is clustered below. This suggests that there is a steeper trend which reverses once N is too large. At a threshold of 0.3, all of the trials for $N = 8$ did not reach the threshold, and are above the trendline (Fig. 10c). When excluding all $N = 8$ points the slopes range from -0.59 to -0.78.

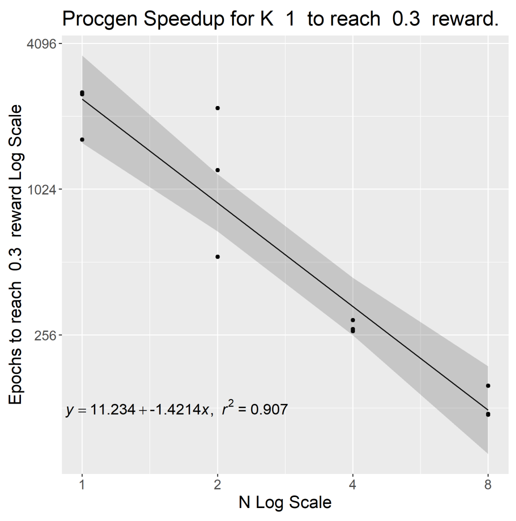
For $K = 100$, there is no speedup, and potentially there is a slowdown (Fig. 11).



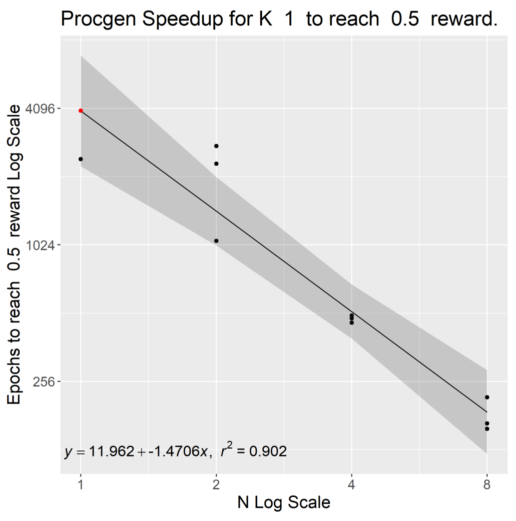
(a) Slope -1.1118, Std. 0.1122



(b) Slope -1.3602, Std. 0.1828

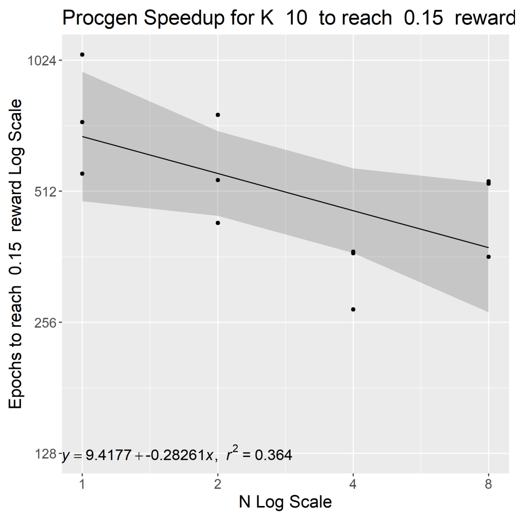


(c) Slope -1.4214, Std. 0.1440

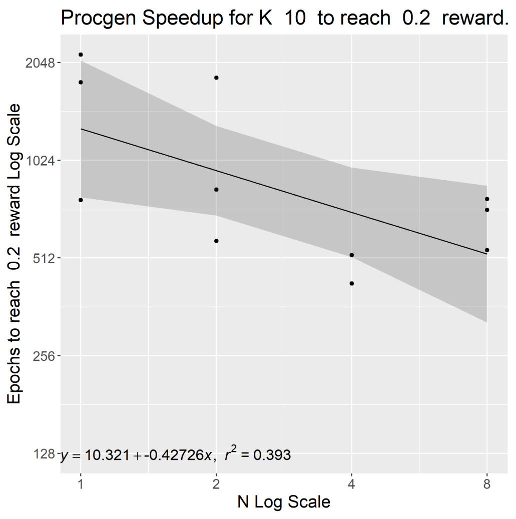


(d) Slope -1.4706, Std. 0.1717

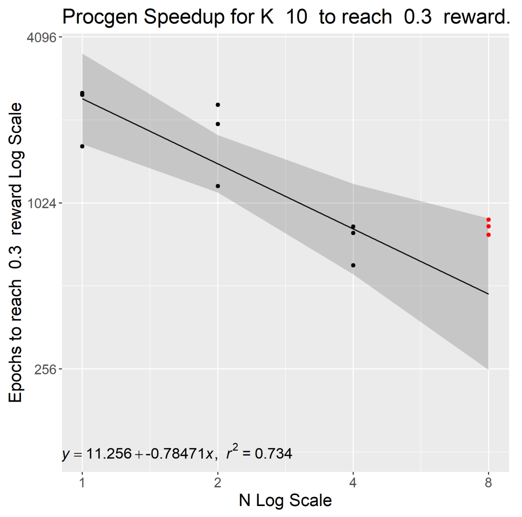
Figure 9: Speedup over N for K 1



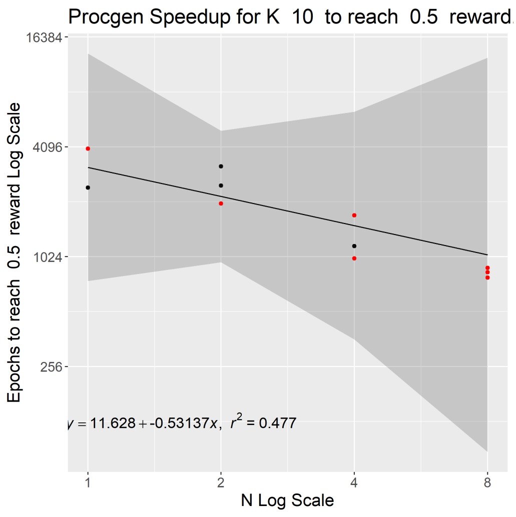
(a) Slope -0.2826, Std. 0.1182



(b) Slope -0.4273, Std. 0.1679

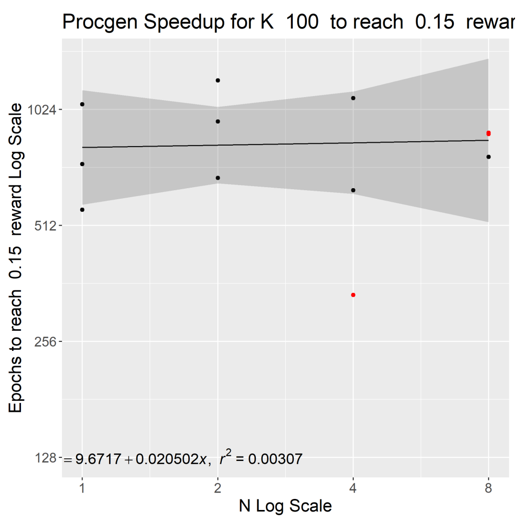


(c) Slope -0.7847, Std. 0.1786

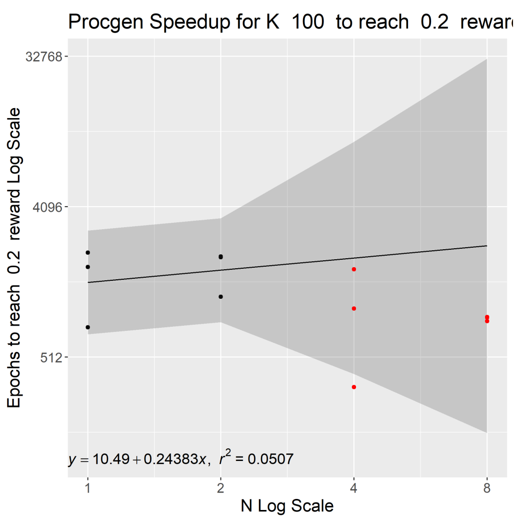


(d) Slope -0.5314, Std. 0.3933

Figure 10: Speedup over N for K 10



(a) Slope 0.0205, Std. 0.1397

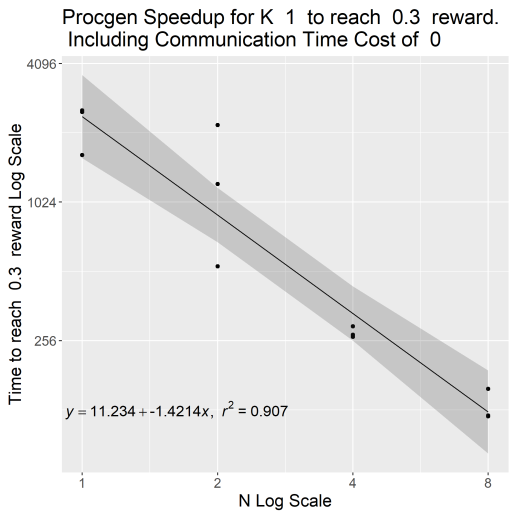


(b) Slope 0.2438, Std. 0.5275

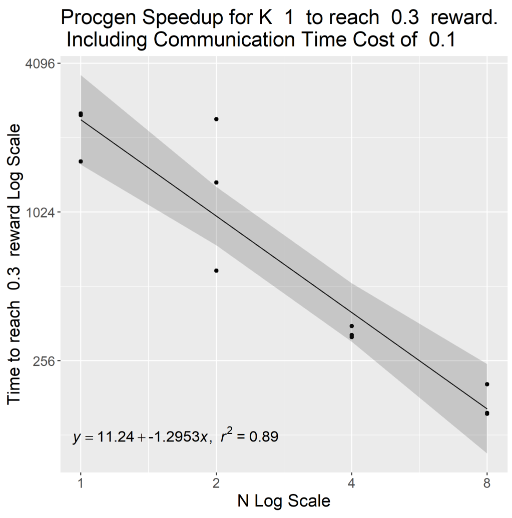
Figure 11: Speedup over N for K 100

4.2.3 Linear Speedup for N with Communication Time

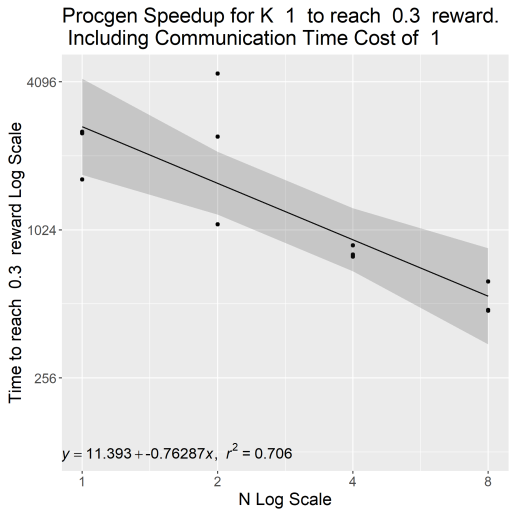
Since even a moderate K of 10 epochs results in a slowdown as N increases from 4 to 8, synchronizing frequently is likely essential to get a speedup in realistic situations, though this may have a significant communication overhead. The following graphs are for different relations to the cost of a communication relative to the cost of an epoch, using $O(log(N))$ complexity for all reduce. The formula for total time including communication is $T_c = E + C * (E//K) * \log_2(N)$, where C is the communication cost, and E is the number of epochs.
A communication cost of 0.1 may be representative of a robotics situation where sampling the environment is more expensive than running the model or synchronizing. A communication cost of 10 may be representative of learning in a simulation on a GPU where local computations can run faster than network communications.



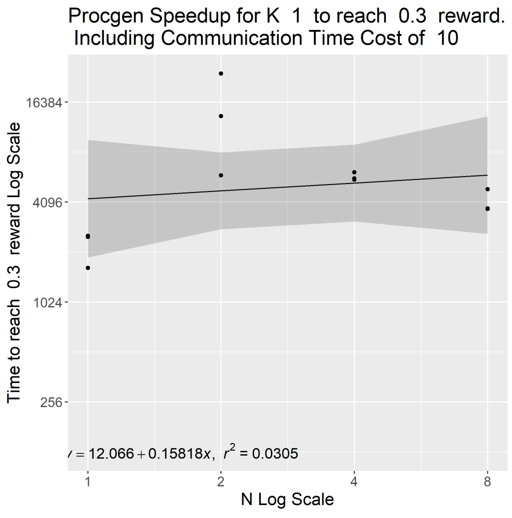
(a) Slope -1.4214, Std. 0.1440



(b) Slope -1.2953, Std. 0.1442



(c) Slope -0.7629, Std. 0.1556

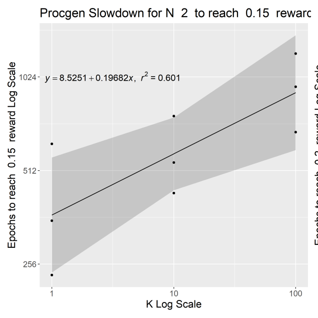


(d) Slope 0.1582, Std. 0.2820

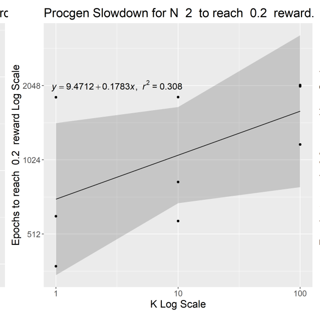
Figure 12: Speedup over N for K 1 Including Communication Time

4.2.4 Slowdown for K

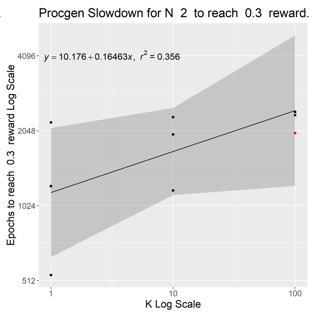
To test for slowdown different values of K are graphed against time to reach a reward threshold on a log log plot for a fixed value of N. The slope of the linear regression corresponds with the exponent of the scaling in real space.
For $N = 2$, the slope ranges from 0.164 to 0.196 for different thresholds. At $N = 4$ the range is 0.398 to 0.442, and at $N = 8$ the range is 0.554 to 0.824 (Fig. 13). The slope for K is increasing as N gets larger, though is still sub-linear at all values tested.



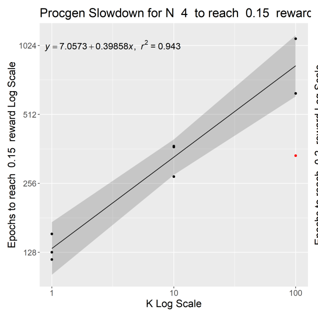
(a) Slope 0.19 Std. 0.06



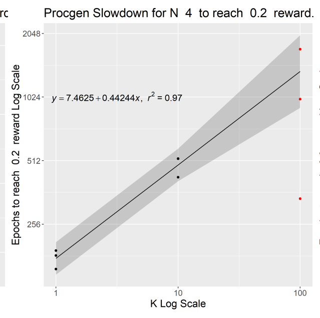
(b) Slope 0.17 Std. 0.10



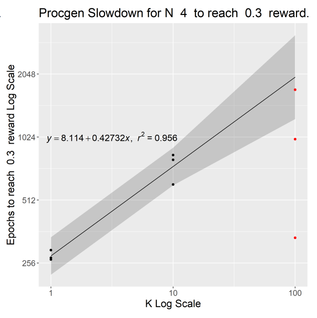
(c) Slope 0.16 Std. 0.09



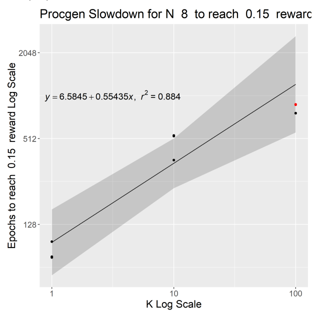
(d) Slope 0.39 Std. 0.04



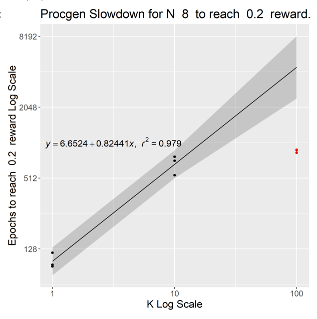
(e) Slope 0.44 Std. 0.04



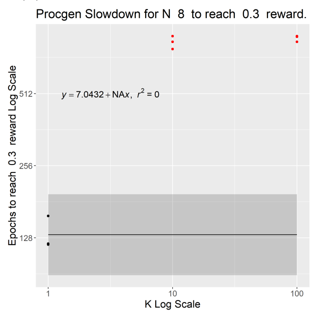
(f) Slope 0.43 Std. 0.04



(g) Slope 0.55 Std. 0.09



(h) Slope 0.82 Std. 0.06



(i) Slope NA

Figure 13: Slowdown over K for fixed N

5 Discussion

5.1 Tabular Q-learning

5.1.1 Steady State Error

For tabular Q-learning, we showed that (EQ 1), accurately predicts the relationship between the α/N term and steady state error D^2. This bound does not fully predict the relationship between K and the steady state error, with the measured exponents in $(K-1)\alpha^2$ lower than predicted, and decreasing as K increases. In the limit as $K \to \infty$, learning is equivalent to N independently trained models, but (EQ 1). fails to capture this and instead predicts infinite error. Since the experiments had some values of K which exceeded the time a single model takes to reach steady state error, this limiting behavior was likely relevant.

5.1.2 Linear Speedup

For tabular Q-learning, we demonstrated a linear speedup with N when scaling α by N, and there is no linear speedup for a fixed α. This is predicted by 1) since the only term dependent on Time, $(1-\alpha)^T$ is related to α, but not N. The linear speedup is entirely caused by the ability to increase the learning rate without increasing the steady state error as N increases which may not be true for more more complex environments.

5.2 Generalization of Toy Models

In tabular Q-learning there is a single optimal answer that Q-learning will eventually converge to. For a convolutional network trained on complex environments there are symmetries in the architecture allowing the same function to be implemented with many different sets of parameters, and different local optimum for different strategies which all get maximum reward. If the agents in the setting are learning different strategies or even representing the same strategy using different directions, averaging the models together will result in interference that was not an issue in tabular Q-learning. Another difference is that in the small MDP environment for tabular Q-learning exploration was easy since a random policy (epsilon = 1) learned at about the same rate as sampling from the learned policy (epsilon = 0). In more complex environments random exploration is much less efficient than both exploring and exploiting using the current learned model. This creates a serial dependency in learning which was not present in tabular Q-learning with the small MDP environment. For these reasons speedup for using federated learning with convolutional networks in a more complex environment may have a different relationship with α, N and K.

5.3 PPO in Procgen

For the Bigfish Procgen environment, Federated PPO showed superlinear speedup, but this degraded as K increased and even reversed the speedup trend with larger N.

5.3.1 Superlinear Speedup

The data showed a superlinear speedup over N for $K = 1$, a sublinear speedup for $K = 10$ and no speedup for $K = 100$.
While it is not expected to be able to achieve superlinear speedup in general with optimal hyperparameters, it may be possible with fixed hyperparameters since there may be a different optimum for different N and K. Some potential differences in fixed hyperparameter's behavior as N and K vary are:
At $K = 1$, the models are averaged together every epoch so effectively the learning rate is reduced by a factor of N. Unlike in tabular Q-learning where larger learning rates always learned faster, but converge at higher error, in Neural Networks it depends on the shape of the loss landscape whether a larger or smaller learning rate will learn faster.
There are N times as many total samples per epoch across all the models. PPO alternates between collecting samples and a training epoch on that data, and the loss is clipped to make sure the model does not move too far within one epoch [12]. Even with only 1 model and no federation, 8 epochs with S samples is not the same as 1 epoch with $8 * S$ samples.
In supervised learning with convolutional networks, batching can improve final accuracy on the dataset [10]. When parallelized, this could cause a real superlinear speedup. While federation is not the same thing as batching this shows that parallel steps are not always just an optimization of sequential steps with the same result.
If the fixed hyperparameters used are more optimal for a large N, then there could be a superlinear speedup.

5.3.2 Slowdown with larger K

The data showed a sublinear slowdown over K at all values of N. At $K = 1$, the models always synchronize before generating new data, while at $K = 10$, the models will generate diverging data before synchronizing. This could be a threshold where interference becomes a major problem and not just a smooth decrease in performance with increasing K.
This may be the cause of the trend reverse where at $K = 10$, there is a speedup up to $N = 4$, but performs worse at $N = 8$.
Methods like FedAvg rely on linear mode connectivity between the models for the average model to be an effective model. When models have learned different generalization strategies they may not be linearly mode connected, and the average model may be on a high loss wall in between the low loss basins [4].

Since PPO is an online RL algorithm, models with diverging strategies will generate further diverging samples. If the strategies of each model become distinct enough, then they may no longer be linearly mode connected, and averaging would decrease performance.

6 Conclusion

We showed that Federated RL has linear speedup in practice both for tabular Q-learning, and with PPO and a Convolutional Network in Procgen. In Tabular Q-learning the results confirm the theoretical prediction for the α/N term but not the $(K-1)\alpha^2$ term from (EQ 1). Some experiments in Procgen showed superlinear speedup, but this is most likely caused by non-optimal hyper parameters and not expected in general. The speedup for Procgen environments is more brittle than on tabular Q-learning, so it will work best in contexts where communication is relatively cheap compared to sampling the environment and running the model such as real world robotics.

References

[1] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. December 2019.

[2] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. October 2020.

[3] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. February 2018.

[4] Jeevesh Juneja, Rachit Bansal, Kyunghyun Cho, João Sedoc, and Naomi Saphra. Linear connectivity reveals generalization strategies. May 2022.

[5] Sajad Khodadadian, Pranay Sharma, Gauri Joshi, and Siva Theja Maguluri. Federated reinforcement learning: Linear speedup under markovian sampling. June 2022.

[6] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. October 2016.

[7] PACE. *Partnership for an Advanced Computing Environment (PACE)*, 2017.

[8] Jiaju Qi, Qihao Zhou, Lei Lei, and Kan Zheng. Federated reinforcement learning: techniques, applications, and open challenges. *Intelligence &amp Robotics*, 2021.

[9] Zhaonan Qu, Kaixiang Lin, Zhaojian Li, Jiayu Zhou, and Zhengyuan Zhou. A unified linear speedup analysis of stochastic fedavg and nesterov accelerated fedavg. July 2020.

[10] Pavlo M. Radiuk. Impact of training set batch size on the performance of convolutional neural networks for diverse datasets. *Information Technology and Management Science*, 20(1), jan 2017.

[11] Satinder Singh Yishay Mansour Richard S. Sutton, David McAllester. Policy gradient methods forreinforcement learning with functionapproximation. 2000.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. July 2017.

[13] Barto A. G. Sutton, R. S. *Reinforcement Learning: An Introduction.* The MIT Press, 2018.

[14] Hankz Hankui Zhuo, Wenfeng Feng, Yufeng Lin, Qian Xu, and Qiang Yang. Federated deep reinforcement learning. January 2019.