Thesis Draft

Michael Einhorn

January 2023

1 Introduction

1.1 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 [8]. 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.** $q_*(s,a) = \Sigma_{s',r} p(s',r|s,a)(r + \gamma \max_{a'}(q_*(s',a'))$**. [8]**

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 [6]. 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. [7]

1.2 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 [2]. 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 [5]. 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 [5]. To our knowledge there is not a similar study on linear speedup for federated reinforcement learning.

1.3 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 [9] 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 [9]. 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 [4]. It seems FedRL method used above is a VFRL approach, though Zhuo et al. do not use this term in the paper [9]. 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. [4]

1.4 Contribution

We will test Federated Reinforcement Learning for stability of convergence and 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. This will experimentally test a theoretical proof of linear speedup in simple tabular Q learning cases, and test 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, and ensure the model will converge to a solution.

1.5 Revision

When I have some results I think I will include some of the general ideas in the introduction. Such as if the results support the theory, and if there is linear speedup for PPO.

2 Methods

2.1 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. $q_* = \Sigma_{s',r} p(s', r|s, a)(r + \gamma \max_{a'} q_*(s', a'))$ [8] .

A random transition matrix will tend to have an optimal policy which is barely better than randomly selecting actions, so the probability

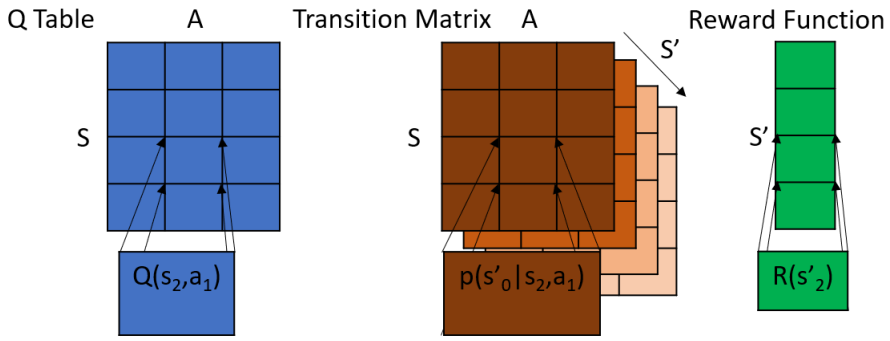


Figure 1: Q Table and MDP Environment with 4 States and 3 Actions

distribution needs to be less uniform. The method I 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.

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 [8]. 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.

My implementation is federated by running Q learning separately on each CPU core, and averaging the weights together after a fixed number of steps K, similar to FedAvg [5]. 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 [3].

There will be some tradeoff between communicating as frequently as possible, and the time for communication. While I am not measuring what this time would be on a realistic network, I can calculate how low it would need to be for federating to have an advantage. Other interesting things are if Federated 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 federated 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 will be 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.

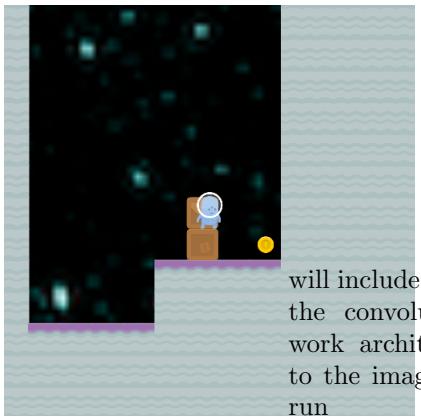
For tabular Q learning in an MDP, the expected relationship is $D^2 \propto (1-\alpha)^T + \alpha/N + (K-1)\alpha^2$ where D is distance to the bellman q table, T is time, α is the learning rate, N is the number of agents, and K is the synchronization frequency. Linear speedup with N is caused by being able to increase α by a factor of N while keeping the α/N term constant. For more complex environments reward will need to be used as the metric since there is not an error from a known ground truth.

2.2 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 [7]

Coin Run and a few other procedurally generated games in Procgen, are common baseline environments to test neural network models [1]. A small convolutional net can learn to play these games with PPO. The model is given the image for the current frame and outputs the action that it takes.

I'm not sure how I will federate the implementation, some options I have thought of are running multiple processes with torch distributed,



will include a graphic of the convolutional network architecture next to the image from coin run

(a) Coin Run [1] (b) Model Architecture

or adding a model index dimension to the parameters of the model allowing them to be trained separately on the same GPU process.

3 Results

3.1 Tabular Q Learning

3.1.1 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

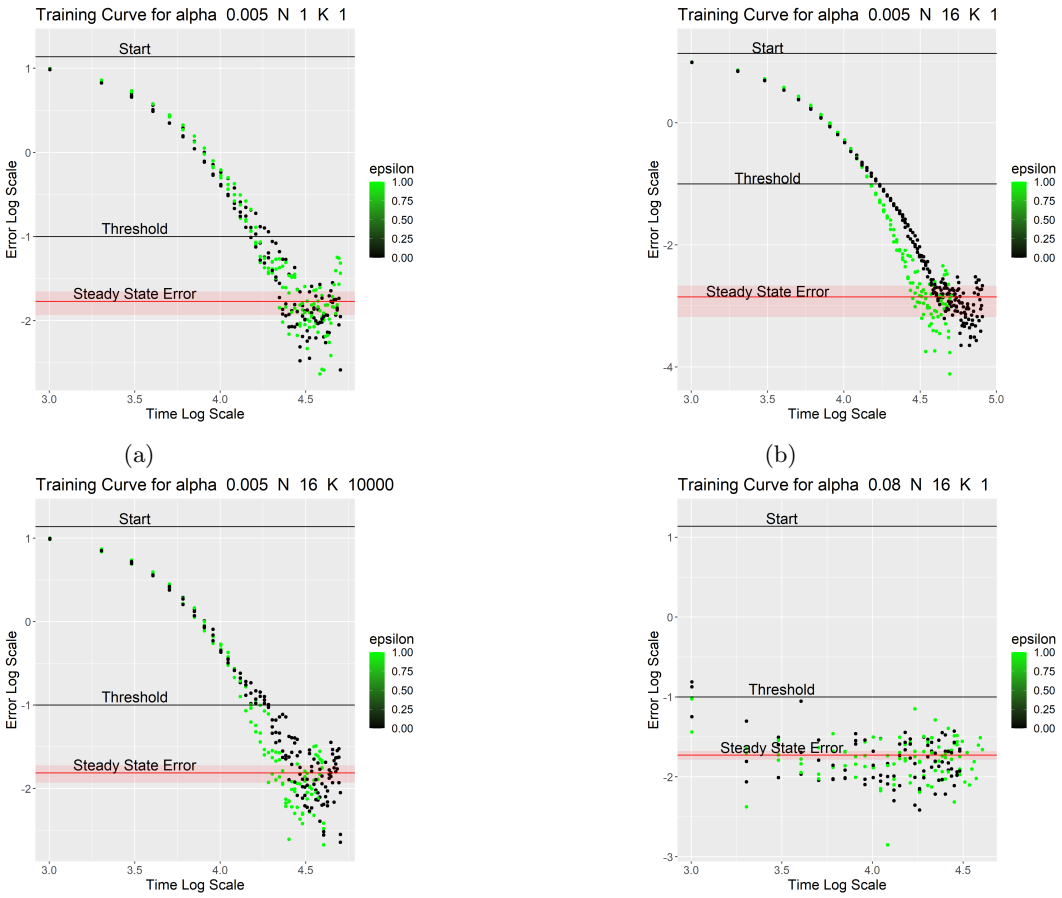


Figure 3: Example Training Curves

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 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.

3.1.2 Steady State Error α/N Term

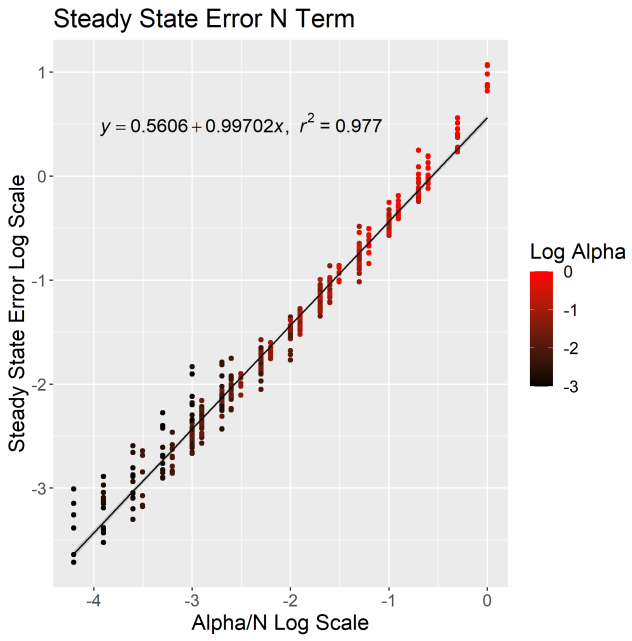


Figure 4

To test the α/N Term, in the theoretical equation, $D^2 \propto (1-\alpha)^T + \alpha/N + (K-1)\alpha^2$, steady state error is used eliminating the 1st term dependent on T, and K=1 is used eliminating the 3rd term dependent on K. The equation 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.

3.1.3 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 T term.

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

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

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

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

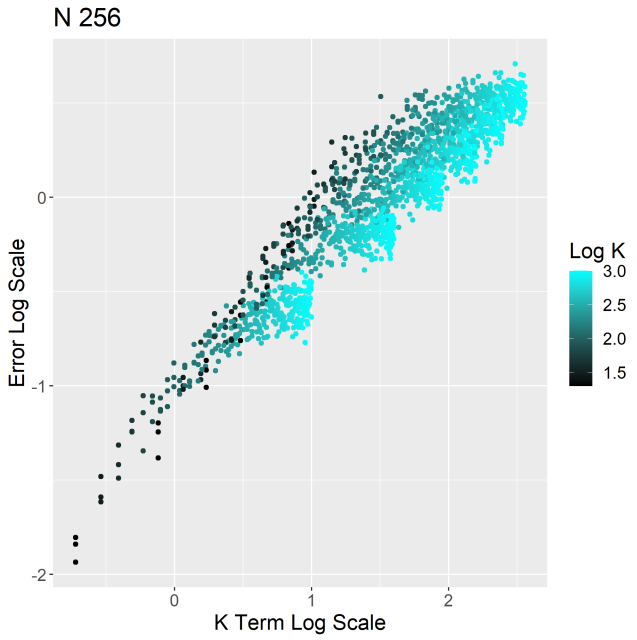


Figure 5

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. 5). 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. 6). 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.

3.1.4 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 time term in the equation

Figure 6

$D^2 \propto (1-\alpha)^T + \alpha/N + (K-1)\alpha^2$ **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.**

Using a learning rate $\alpha = a_1 N$ **makes the theoretical equation** $D^2 \propto (1-\alpha_1 N)^T + \alpha_1 + (K-1)(\alpha_1 N)^2$**. Unlike the original form of the equation, the time 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. 8a). When** $K = 10^4$**, this slope drops to -0.855 (fig. 8b). These lines are graphed with a** 95% **confidence interval of the regression's estimate.**

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. 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 this will depend on the hardware used. With $K = 1$ (fig. 9a) the slope is reduced to only 0.38 which is a significantly sublinear speedup. This increases for $K = 10$ (fig. 9b) and $K = 100$, up to $K = 1000$ (fig. 9c) to a slope of 0.96, which is almost linear, and reduces to 0.86 for $K = 10000$ (fig. 9d).**

3.2 PPO in procgen

4 Discussion

4.0.1 Federated Tabular Q Learning Steady State Error

For federated tabular Q learning, we showed that the theoretical equation $D^2 \propto (1-\alpha)^T + \alpha/N + (K-1)\alpha^2$, accurately predicts the relationship between the α/N term and steady state error D^2. This equation 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 $K \to \infty$, federated learning is equivalent to N independently trained models, but the theoretical equation 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.

4.0.2 Federated Tabular Q Learning Linear Speedup

For federated 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 the theoretical equation 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.

4.1 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 coinrun there are rotational 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 federated 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.

4.2 PPO in Procgen

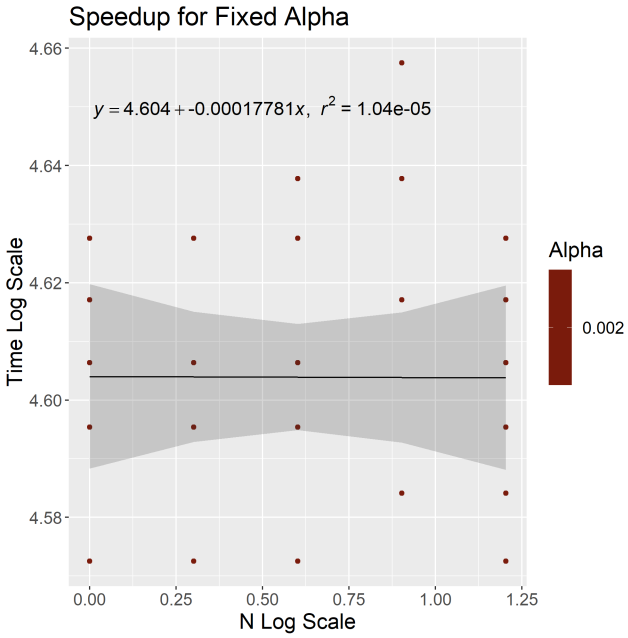
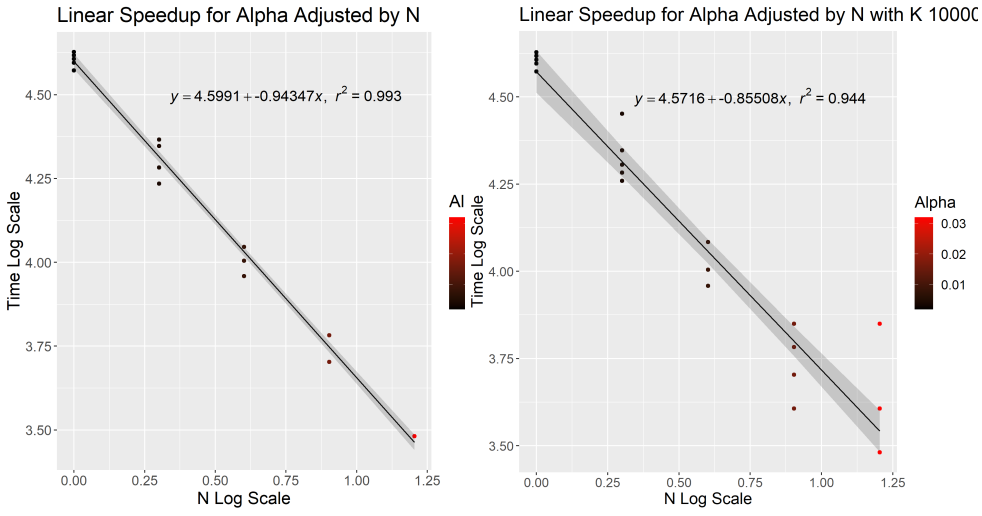


Figure 7



(a) (b)

Figure 8: Speedup scaled α

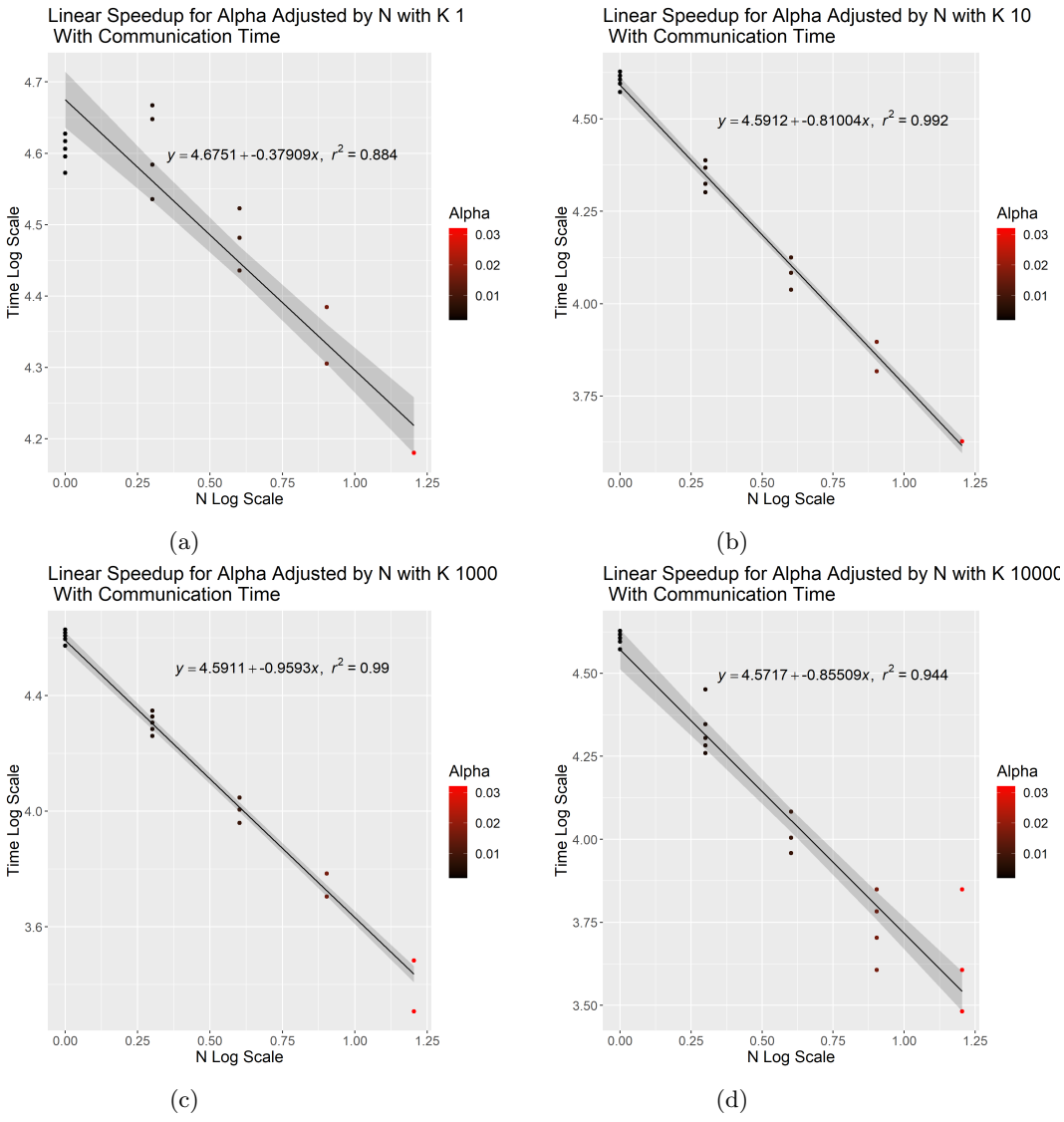


Figure 9: Speedup including $\log 2(N)$ communication complexity.

4.3 Revision

several parts of this methods section are work in progress because I am still planning how I will do these things. I will fill in with more details later. I am currently working on the analysis for comparing tabular q learning to the theoretical model, and will update the section with more details as I finish that.

Convolutional network methods and data still in progress. Planning to end the discussion with how the prediction from the proof for the simple tabular q learning case extend or don't extend to complex models and environments.

The theoretical equation is from the grad student I am working with it is not published yet. Will cite this later.

Latex formatting and positioning with the graphs is kindof weird, so I am only planning to fix this once I have finalized the text around it.

References

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

[2] 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.

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

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

[5] 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.

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

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

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

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