Capsule Network based Distributed Reinforcement Learning

Suraj Panwar, Mtech DESE, 14644
Dola Ram, Mtech DESE, 15197

Abstract

The project undertakes the task of creating a new modified architecture for Deep Q-Learning using Capsule neural networks and comparing it with current existing model architectures. The Capsule network used in this project has been modified for applicability to a Reinforcement Learning application and the appropriate model architecture is selected on basis of the experimentation. The comparison is carried out for both Single Machine/ Local Model as well as for a Multi-Machine/ Distributed Model. The model performance with the worker nodes is tested for scalability as well as the model performance with model parameters especially the effect of Experience replay pool size and the capsule network routing iteration values. Finally, the different model architectures are compared on various parameters such as Scalability, Data Requirement, Training Time, etc.

1. Introduction

Reinforcement learning is an approach to machine learning that is inspired by behaviorist psychology. It is similar to how a child learns to perform a new task. Reinforcement learning contrasts with other machine learning approaches in that the algorithm is not explicitly told how to perform a task, but works through the problem on its own. As an agent, which could be a self-driving car or a program playing chess, interacts with its environment, receives a reward state depending on how it performs, such as driving to destination safely or winning a game. Conversely, the agent receives a penalty for performing incorrectly, such as going off the road or being checkmated. the agent over time makes decisions to maximize its reward and minimize its penalty using dynamic programming.

However, even though the current nature of Reinforcement learning is largely limited to Games with a fixed number of states, the algorithm is being explored in the area of Navigation, trading, etc and have seen huge developments in the recent past regarding the way algorithm is process with the development of TD learning and shift from the traditional Q table based method to a more general algorithm such a Deep Q learning where a CNN is used as a function approximator for the state variable.

However, Reinforcement Learning suffers from a critical problem which limits its liberal application, namely the problem of generalizing over a situation. As, the nature of problems become more complex, the possible states or conditions one can encounter in the environment increases many folds which limits the Reinforcement Learning application as to generalize for a real-world application i.e Autonomous Navigation the algorithm requires a large amount of data and then the data has to be processed by the algorithm which severely limits the applicability of the Reinforcement Learning. Using a Distributed Architecture, one which efficiently deals with data as well as provides a time feasible training for the algorithm is a possible solution to all that ails the algorithm. In this project, we have undertaken a task of implementing a Distributed architecture using a modified Capsule network architecture and comparing it with the conventional network architectures for both a standalone as well as a distributed architecture.

However, the problem of excessive data requirement is just as bad if not worse than a large training time, and is something that we have tried to find a solution to in this modified Capsule network algorithm.

2. Brief Theory

2.1. Deep Reinforcement Learning

Reinforcement Learning tasks are in general tasks in which the agent interacts with an environment through a sequence of observations, actions, and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. More formally, the model follows a set policy like the maximization of the Q function given below.

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \middle| s_t = s,\ a_t = a,\ \pi\right],$$

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q function). This instability has several causes: the correlations present in the sequence of observations, the fact that small updates may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values.

To solve this we employ a data pool known as action replay which stores the data, the samples for training are randomly sampled from the pool and used for training the network. As the sampled variables are randomly selected, this removes the problem of correlation between the variables and allows us to use a Neural Network approximator such as CNN for the Q value approximation.

During learning, we apply Q-learning updates, on samples (or minibatches) of experience (s,a,r,s') , U(D), drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)}\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

Employing the above-mentioned methods as was stated in [2], the neural configuration can then be used with the existing Reinforcement Learning application for Q Value approximation and is what we have used for comparison between the different architectures.

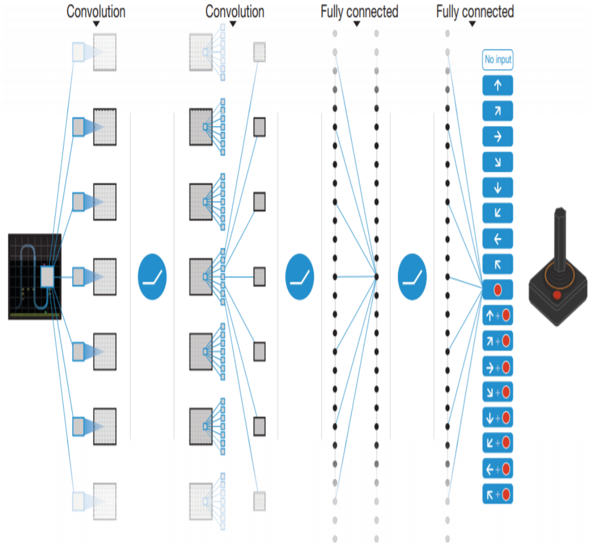

Figure 1. Convolutional Neural Network for Reinforcement Learning application

2.2. Capsule Network

Capsule network in our model is used as a function approximator in place of CNN. A capsule is a group of neurons whose activity vector represents the instantiation parameters of a specific type of entity such as an object or an object part. We use the length of the activity vector to represent the probability that the entity exists and its orientation to represent the instantiation parameters. Active capsules at one level make predictions, via transformation matrices, for the instantiation parameters of higher-level capsules. When multiple predictions agree, a higher level capsule becomes active.

Discriminatingly trained, multi-layer capsule system achieves state-of-the-art performance on MNIST and is considerably better than a convolutional net at recognizing highly overlapping digits[2]. To achieve these results an iterative routing-by-agreement mechanism is used.

The activities of the neurons within an active capsule represent the various properties of a particular entity that is present in the image. These properties can include many different types of instantiation parameter such as pose (position, size, orientation), deformation, velocity, albedo, hue, texture, etc. The fact that the output of a capsule is a vector makes it possible to use a powerful dynamic routing mechanism to ensure that the output of the capsule gets sent to an appropriate parent in the layer above. Initially, the output is routed to all possible parents but is scaled down by coupling coefficients that sum to 1. For each possible parent, the capsule computes a prediction vector by multiplying its own output by a weight matrix. If this prediction vector has a large scalar product with the output of a possible parent, there is top-down feedback which increases the coupling coefficient for that parent and decreasing it for other parents. This increases the contribution that the capsule makes to that parent thus further increasing the scalar product of the capsules prediction with the parents output.

This type of routing-by-agreement should be far more effective than the very primitive form of routing implemented by max-pooling, which allows neurons in one layer to ignore all but the most active feature detector in a local pool in the layer below. We demonstrate that our dynamic routing mechanism is an effective way to implement the explaining away that is needed for segmenting highly overlapping objects.

We want the length of the output vector of a capsule to represent the probability that the entity represented by the capsule is present in the current input. We therefore use a non-linear ”squashing” function to ensure that short vectors get shrunk to almost zero length and long vectors get shrunk to a length slightly below 1.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

$$\boxed{\mathbf{s}_j} = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}\,, \qquad \hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij} \mathbf{u}_i$$

total input to a capsule sj is a weighted sum over all prediction vectors uji. where the cij are coupling coefficients that are determined by the iterative dynamic routing process.

In convolutional capsule layers, each capsule outputs a local grid of vectors to each type of capsule in the layer above using different transformation matrices for each member of the grid as well as for each type of capsule.

2.3. CapsNet architecture

Conv1 has 256, 8 8 convolution kernels with a the stride of 4 and ReLU activation. This layer converts pixel intensities to the activities of local feature detectors that are then used as inputs to the primary capsules. The primary capsules are the lowest level of multi-dimensional entities and, from an inverse graphics perspective, activating the primary capsules corresponds to inverting the rendering process. This is a very different type of computation than piecing instantiated parts together to make familiar wholes, which is what capsules are designed to be good at.

The second layer (Primary Capsules) is a convolutional capsule layer with 32 channels of convolutional 8D capsules (i.e. each primary capsule contains 8 convolutional units with a 9 9 kernel and a stride of 2). Each primary capsule output sees the outputs of all 256 81 Conv1 units whose receptive fields overlap with the location of the center of the capsule. In total Primary Capsules has [32 6 6] capsule outputs (each output is an 8D vector) and each capsule in the [6 6] grid is sharing their weights with each other. One can see Primary Capsules as a Convolution layer with Eq. 1 as its block non-linearity.

3rd layer (Digit Caps) has one 16D capsule per digit class and each of these capsules receive input from all the capsules in the layer below. We have routing only between two consecutive capsule layers (e.g. Primary Capsules and Digit Caps). Since the Conv1 output is 1D, there is no orientation in its space to agree on. Therefore, no routing is used between Conv1 and Primary Capsules. All the routing logits (bij) are initialized to zero. Therefore, initially, a capsule output (ui) is sent to all parent capsules (v0...v9) with equal probability (cij).

Digit caps layer is followed by a Fully connected layer with 30 hidden units. Output layer is FC layers with linear outputs equal to the number of actions. Output layer gives the Q-values for the state corresponding to each action.

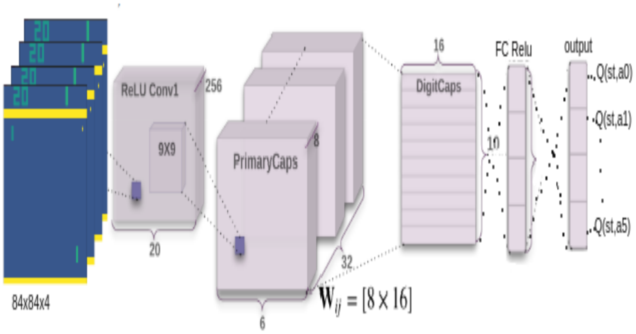

Figure 2. Capsule Network

3. Datasets

Reinforcement learning inherently learns by interacting with a fixed environment based on a fixed set of rules, which provides it with corresponding states to an action and rewards achieved by taking that action. The virtual environment used in this project is PYGAME which provides an easily configurable environment which can rapidly generate the states and present us with the corresponding rewards.

The emulator is used for generating the data, which is then stored in the experience replay pool. The actions are based on an epsilon-greedy policy, which provides the emulator steps based on the current state.

Other than PYGAME, we have also worked with OpenAI gym, which is a python library and provides with the same functionality as the PYGAME, however, we have found through experiments that the PYGAME emulator is more resolute and harder to train at.

4. Baseline

For model performance comparison, we have compared our model performance with that of OpenAI baseline models architecture. OpenAI has open sourced a set baseline, for different Reinforcement Learning architectures for model comparisons. The baseline can be found at Open-AI Baseline.

From the different model baselines that one can select, we have chosen to work on Pong environment as due to its easier learning curve, this model is adequate for rapid testing but not small as to be trivial. This environment has a modest training time and hence is used in this project. The baseline of OpenAI Pong model architecture is thus compared to our model performance.

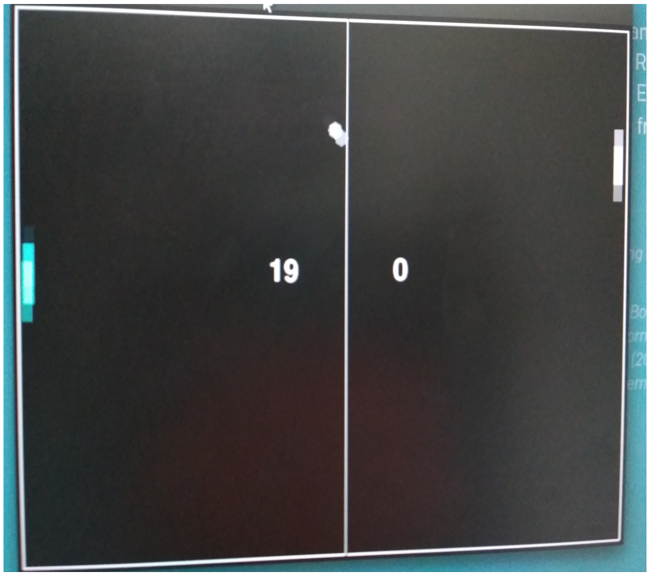

Figure 3. Sample Pong environment screen shot after fully trained with perfect score.

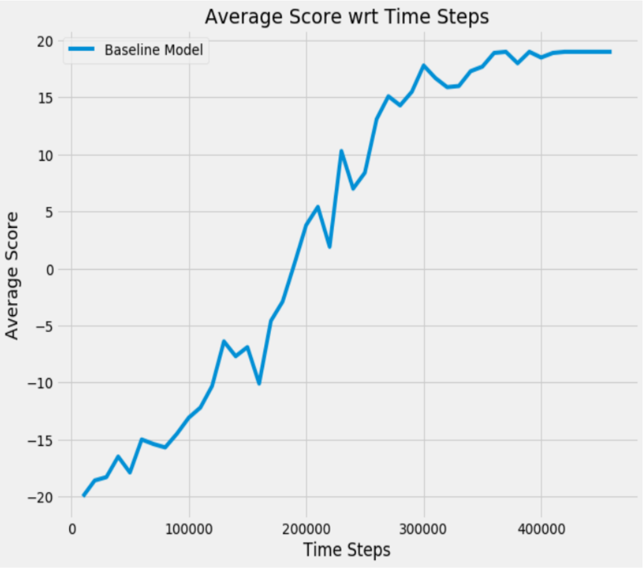

Figure 4. Average Episode Score wrt Time Steps for CNN model

5. Brief Overview of PYGAME and OpenAI

PYGAME is an open source library for python which allows to rapidly generate frames corresponding to an environment with given action and also returns the corresponding reward. This is used for populating the experience replay pool and also to generate rewards when finally testing the model.

The package returns an array of pixel intensities and rewards for the corresponding action, these values are used as the corresponding states and used for model input to the convolution layer, which further gives us the corresponding Q values which can be used for dictating the further actions.

6. Local Implementation

The following exercise was conducted on Turing cluster on a single node in standalone mode. The values then generated were recorded and are used for plotting the final curves.

6.1. Framework

For this project, for creating the Convolutional Neural Network and Capsule Network, we have used Tensorflow for generating the above-mentioned model configuration. For the emulator purpose, we have used PYGAME which is used for state, reward generation for training and testing purposes.

6.2. Single Machine Training

For the above mentioned Convolutional Neural Network model which is employed in the OpenAI baseline architecture is as follows:

- Input size: 84 x 84 x 4
- Conv Layer 1 : 8 x 8 filter
- Conv Layer 2 : 4 x 4 filter
- Fully connected layer : 30
- Fully connected layer : 512
- Output/Q layer : 6

The input image from the emulator is cropped in 84x84 chunk and is appended with the last three frames to give us the input for the Convolutional layer. The output layer consists of 6 nodes, which correspond to the 6 action Q values which could be taken for the given environment.

6.3. Capsule Network Training

The capsule network is used in more or less similar fashion to the CNN architecture, the network configuration is as follows:

- Input size: 84 x 84 x 4
- Conv Layer 1 : 8 x 8 filter
- Primary Capsules: 8D capsule with 9 x 9 filter
- Digit Caps: 10 x 16 D
- Fully connected layer: 30
- Output/Q layer : 6

The capsule network is also utilized as a function approximator and the input is fed with a stacked input image group, which utilizes the Convolutional layer and have the final output of Q values of the model. The iterative value for the capsule network is kept at 1 for this case.

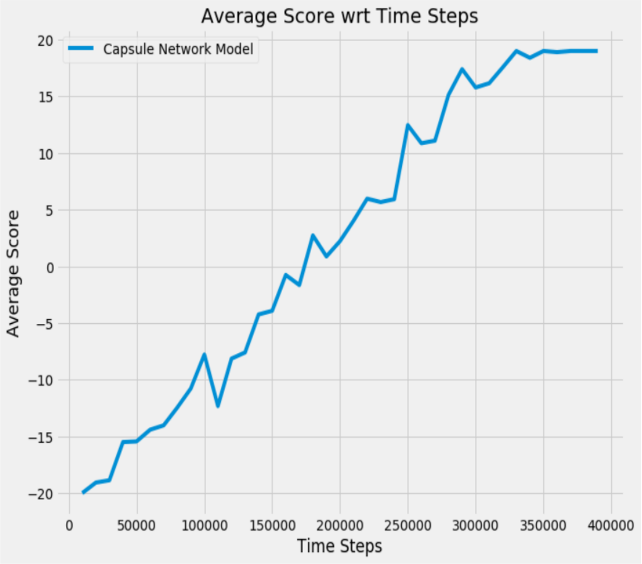

Figure 5. Average Episode Score wrt Time Steps for Capsule Model

Model	Time-steps
Convolutional Model	330411
Capsule Model	320005

Table 1. Model Time-step performance.

6.4. Comparison with Baseline

The capsule model comparison with the standard CNN architecture is shown below.

7. Inference

Even though the network architecture for Capsule network takes more time-steps to train, it however took 20 times fewer update steps compared to CNN architecture, which has to be updated at every time step. The capsule network for the above configuration of iteration value of 1 has to be updated 20 times fewer than the corresponding CNN architecture.

8. Distributed Implementation

The distributed implementation for the model architectures is carried on multiple nodes of the Turing cluster. The values are generated from there and the plots have been shown in the following sections.

8.1. Framework

For training the models, we are using the parameter server approach where we are using the Distributed Tensorflow for creating a parameter server and the worker nodes. The emulator used is PYGAME and the model frame generation and action values are decided individually on each worker.

8.2. Experiments

8.2.1. Model performance variation with Experience Replay size

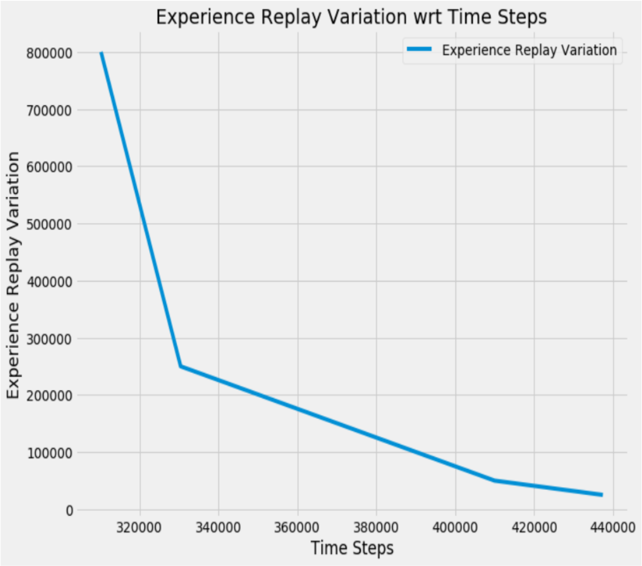

Figure 6. Experience Replay Size wrt Time Steps for Capsule Model

Experience replay is one of the most crucial parameter in the Deep Q learning algorithm. The variation of the experience replay is shown along with the change in the time steps needed for complete model training in the baseline OpenAI model architecture. From the plot we can see that more size of experience replay expedites the process of training, however, it comes at a cost of increased storage requirement, this is specially true for distributed architectures, where multiple workers have to store there own copy of experience pool and so can cause a substantial increase in memory requirement. We have taken a compromise between the two and have 250000 for the rest of the executions.

8.2.2. Model performance variation with epsilon

The Epsilon is the parameter which decides how much the model is willing to exchange the greedy execution for exploring different avenues which could or could not result in increased future rewards. The variation of the model parameter along with the time-steps required for training the baseline model is plotted. From it we can see that the increase in epsilon from the 5% mark resulted in substantial time step requirement increase and hence we will use 5% as

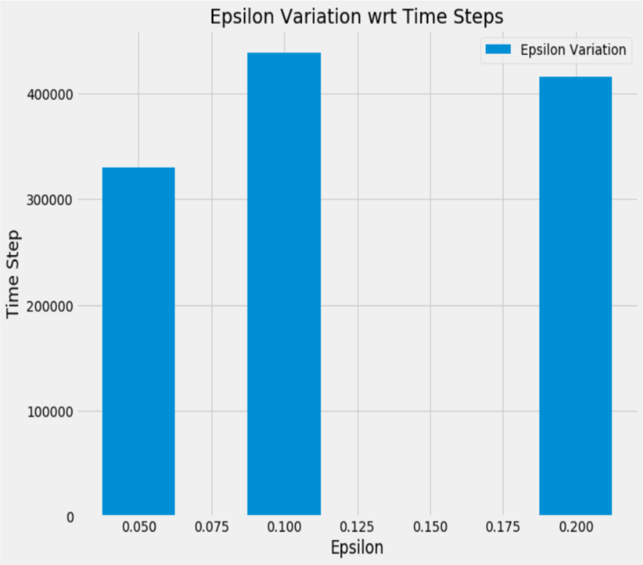

Figure 7. Epsilon wrt Time Steps for Capsule Model

the final epsilon.

8.2.3. Model performance variation with gamma

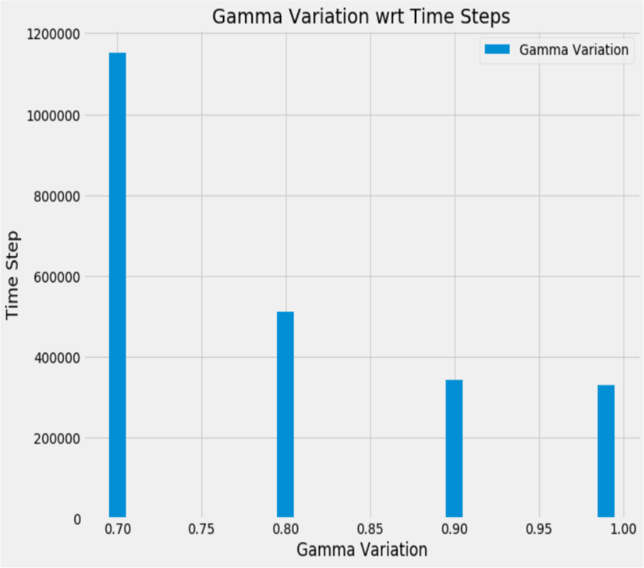

Figure 8. Gamma wrt Time Steps for Capsule Model

The gamma parameter defined the model expectation for the future reward and how much it prefers the other time step rewards compared to the present reward. The variation of gamma along with the time-steps required for the final baseline model training is plotted. This shows that a decrease in gamma causes a substantial increase in training steps as the model has lesser and lesser memory of other rewards. So, we have chosen a value of 0.99 for gamma.

8.2.4. Model performance variation with routing iteration for capsule network

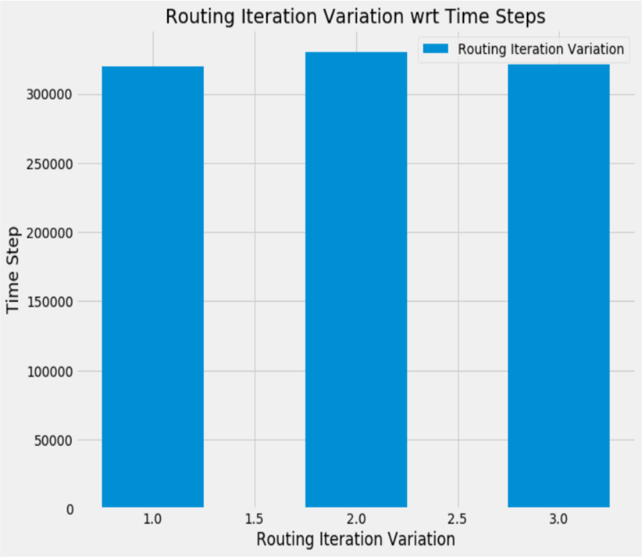

Figure 9. Routing Iterations wrt Time steps for Capsule Model

The variation of the routing layer for the Capsule network has been reported here, from the observations, we can see that the routing iterations do not have much effect on the model training performance. As, the routing iteration increases, the model can learn more complex features but as the requirement for our application is not there, we do not see a effect of routing iteration on the time step requirement for model training.

8.3. Network Architecture

The network architecture for all the models in a distributed environment is essentially the same, the workers maintain an individual copy of the game running while maintaining a separate Experience replay pool. The workers than interact with the Parameter servers while training to fetch parameter, compute gradients and then return the parameters for updating. The various configurations of the parameter server have been implemented for both the CNN and the Capsule model and the architecture and result has been discussed here.

8.4. Distributed Implementation for Baseline Model

For the distributed implementation of the OpenAI baseline model using CNN configuration, we are employing a Asynchronous, Synchronous and a Stale gradient update method, there results have been compared and the plots have been shown.

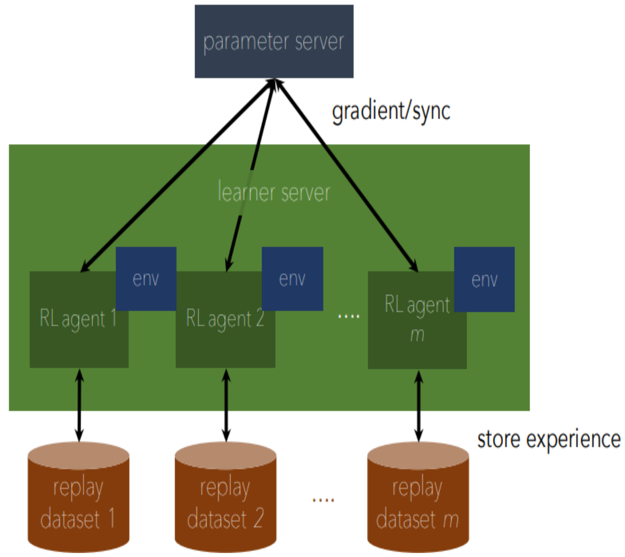

Figure 10. Distributed Reinforcement Learning Architecture

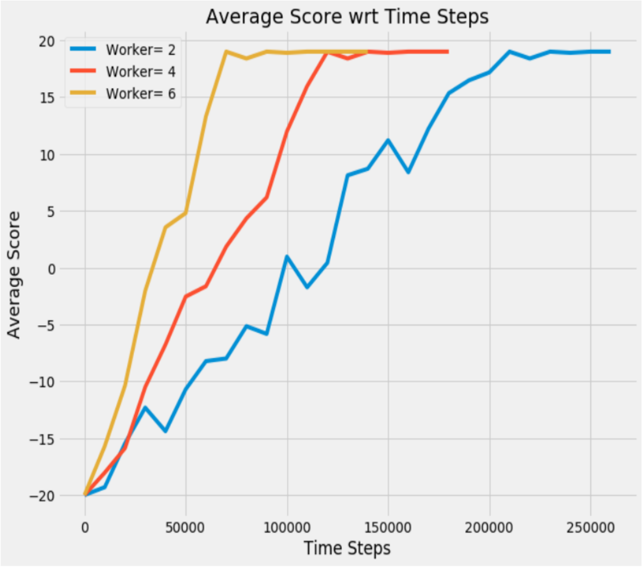

Figure 11. Average Episode Score wrt Time steps for different workers for Asynchronous Model for Baseline Model

8.4.1. Asynchronous

The Asynchronous gradient update scheme has been implemented for Baseline model and the result is shown in the following figure. The speed up for the Asynchronous configuration for various number of worker nodes have also been shown, it is noticed that as the number of workers are increased, the speed ratio vs the worker nodes starts to flatten out. For the configurations, the parameter server nodes and the worker nodes were both made on Turing cluster and the Time Steps were recorded for the algorithm to reach a score of 19 for the first time, at which it is assumed that the episode is over and the next episode is started.

As for the average score, at every 1000 time steps, the average score of the last 3 episodes is used as the current average episode error. From the curves, it can be seen that the more worker process achieves the target score fastest and once achieved, the variation in score is very minute which shows that the algorithm has a good event memory to maintain a almost perfect score.

8.4.2. Synchronous

The Synchronous gradient update scheme has been implemented for Baseline model and the result is shown in the mentioned figure. Similar to the Asynchronous scheme the increase in the number of workers resulted in the decrease in the number of time steps required to achieve a score of 19 for the Pong game.

The variation of the workers also results in rapid increase in the Average episode score, which tells the model about how much the score it has attained in the last 3 episodes averaged.

8.4.3. Stale Model

The model staleness for the network configuration is kept at 50 replica models besides the worker nodes. The model training time steps are recorded for the different worker nodes and the average episode score is used as metric of algorithm performance from the graph, we can see that as the model worker nodes increase, the model becomes faster to train, requiring fewer time steps to train and retains the memory as after training the score holds steady for the rest of training period at the maximum score achievable.

8.5. Distributed Capsule Network

For the distributed implementation of the OpenAI baseline model using CNN configuration, we are employing an Asynchronous, Synchronous and a Stale gradient update method, their results have been compared and the plots have been shown.

8.5.1. Asynchronous

The Asynchronous gradient update scheme has been implemented for the capsule network model and the result is shown in the following figure. The speed up for the Asynchronous configuration for various number of worker nodes have also been shown, it is noticed that as the number of workers are increased, the speed ratio vs the worker nodes starts to flatten out. For the configurations, the parameter server nodes and the worker nodes were both made on

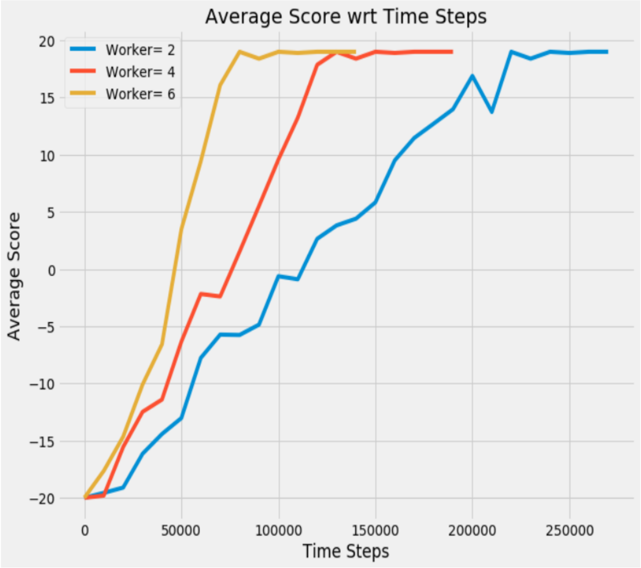

Figure 12. Average Episode Score wrt Time steps for different workers for Synchronous Model for Baseline Model

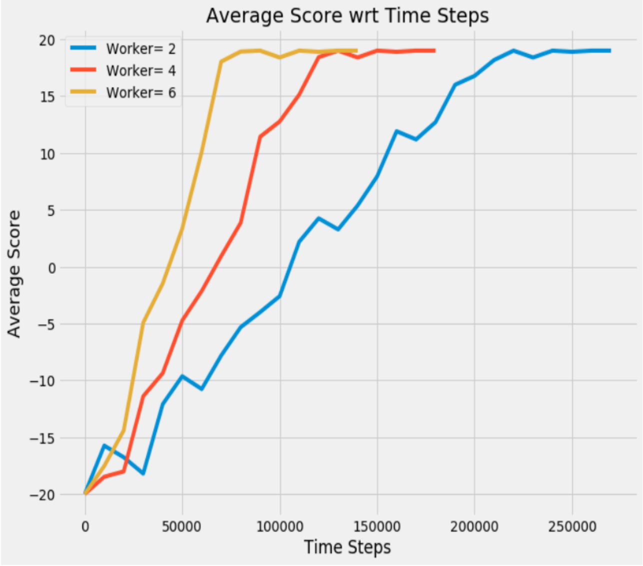

Figure 13. Average Episode Score wrt Time steps for different workers for Stale Model for Baseline Model

Turing cluster and the Time Steps were recorded for the algorithm to reach a score of 19 for the first time, at which it is assumed that the episode is over and the next episode is started.

As for the average score, at every 1000 time steps, the average score of the last 3 episodes is used as the current average episode error. From the curves, it can be seen that the more worker process achieves the target score fastest and once achieved, the variation in score is very minute which shows that the algorithm has a good event memory to maintain a almost perfect score.

8.5.2. Synchronous

The BSP SGD model was trained in Tensorflow using the hyper parameters decided on. The number of workers was varied on the nodes of the Turing cluster. The training step steps for different workers were noted and have been reported here.

The worker nodes had a separate copy of the model and maintained the experience pool for the implementation. It can be seen that a increase in the number of worker nodes results in decrease in the number of time steps required to train the model

8.5.3. Stale Model

The model staleness for the network configuration is kept at 50 replica models besides the worker nodes. The model training time steps are recorded for the different worker nodes and the average episode score is used as metric of algorithm performance from the graph, we can see that as the model worker nodes increase, the model becomes faster to train, requiring fewer time steps to train and retains the memory as after training the score holds steady for the rest of training period at the maximum score achievable.

Model	**2 Wrk**	**4 Wrk**	**6 Wrk**
Asynchronous	206506	122374	88354
Synchronous	213168	124683	89300
Stale Gradient	218815	132164	89785

Table 2. Baseline Model Time step comparison for different workers.

The capsule network model requires less amount of time steps for training compared to the conventional model using Convolutional architecture.

9. Model Scalability

The model performance in lieu of time step requirement for training the model with respect to the number of the worker nodes is shown in the above tables for both Capsule network and the conventional CNN model. From the tables we can observe that the model requires less amount of time steps for training as the number of worker nodes are increased, but also that the Capsule network requires lesser number of time steps for training compared to the CNN based model, which shows an advantage of capsule network over traditional convoluted architecture.

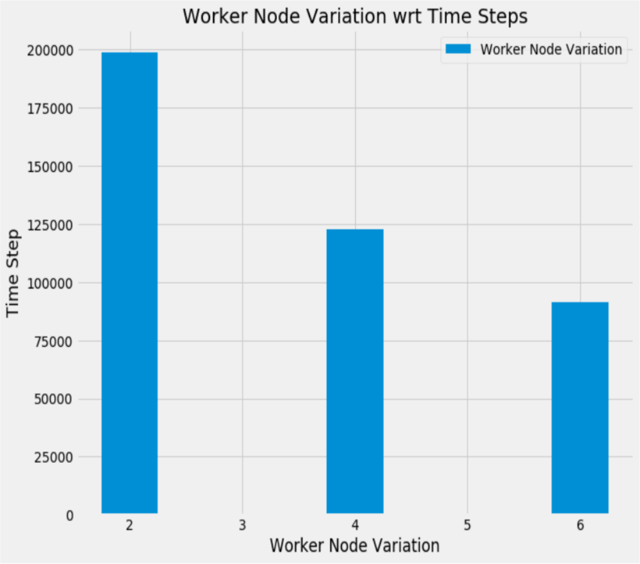

Figure 14. Time steps for different workers for Asynchronous Model for Capsule Model

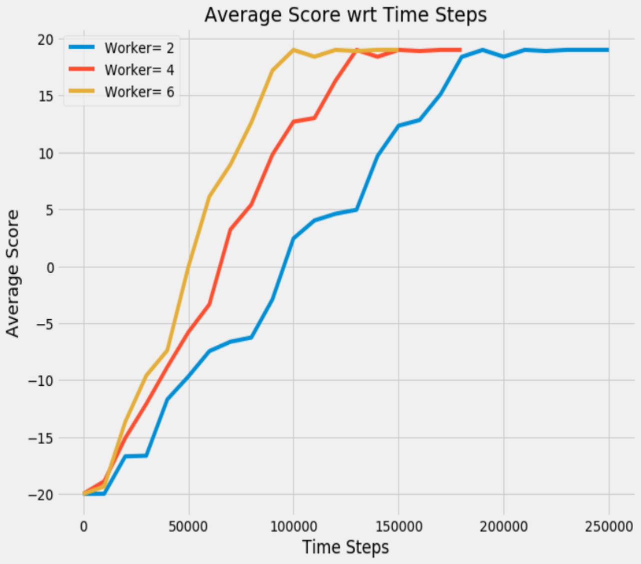

Figure 15. Average Episode Score wrt Time steps for different workers for Asynchronous Model for Capsule Model

Model	**2 Wrk**	**4 Wrk**	**6 Wrk**
Asynchronous	198760	122607	91430
Synchronous	209153	125968	92755
Stale Gradient	213336	133335	93843

Table 3. Capsule Model Time step comparison for different workers.

10. Inference

The Capsule network from our study shows that the data requirement or the time step requirement of the capsule network is less than that of the CNN architecture, but because of the routing layers is more computationally intensive and hence requires more time for implementation.

The capsule network is substantially smaller in size compared to the CNN model that was employed here but provided almost a equal level performance compared to the CNN architecture. This holds huge promise for the applicability to this algorithm to Distributed architectures where whole model has to be loaded on the main memory during execution, but the data can be shared, i.e Data Parallel applications because of its small memory requirement.

The capsule network also holds more potential towards more complex applications as we saw in our project, that the program ram satisfactorily on routing setting set to 1 which can however be increased to increase the feature learning capability of the model and hence increase the applicability of the model for model complex tasks.

Overall we have found that the capsule network is a good replacement for the current CNN architecture due to its small parameter count, less time step requirement and ability to learn complex features using iterative routing.

11. Future Work

There are multiple avenues and recent papers such as [3] that have introduced concepts in reinforcement learning that deal with experience replay. As mentioned before, the experience replay mechanism is a major bottleneck in the reinforcement learning task as the size of experience replay has a magnified impact on distributed architectures by increasing data requirement manifold. The effective learning using Experience replay is also slow as the samples have no priority parameter and the system might train on examples where are stale. The prioritized experience replay negates this problem by assigning priority to the samples which allows the model to pick a high priority sample for training compared to a stale sample. This can be used in conjugation with Capsule network and will be considerably more effective as the update step is carried out sparsely compared to a traditional CNN model.

As mentioned in [5], we can also employ EM routing in place of dynamic routing which can further provide better results compared to the current capsule configuration as well as standard CNN architectures.

Also, the Capsule network architecture can possibly be modified for better applicability to reinforcement learning by decreasing the iteration time and increasing the update frequency which could have a favorable outcome on the training duration.

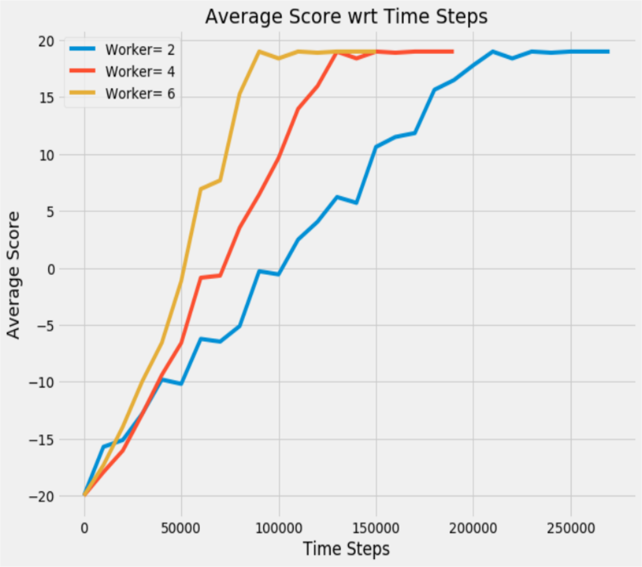

Figure 16. Average Episode Score wrt Time steps for different workers for Synchronous Model for Capsule Model

12. Reference

[1] Human-level control through deep reinforcement learning, *Volodymyr Mnih, Koray Kavukcuoglu, David Silver*, Nature/14236, 2015

[2] Dynamic Routing Between Capsules, *Geoffrey E. Hinton, Sara Sabour, Nicholas Frosst*, arXiv:1710.09829v2 [cs.CV] 7 Nov 2017

[3] Prioritized Experience Replay, *Tom Schaul, John Quan, Ioannis Antonoglou, David Silver*, ICLR 2016

[4] Distributed Prioritized Experience Replay, *Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, David Silve*r, arXiv:1803.00933

[5] Matrix capsules with EM routing, *Geoffrey E Hinton, Sara Sabour, Nicholas Frosst*, ICLR 2018 (Open Review)

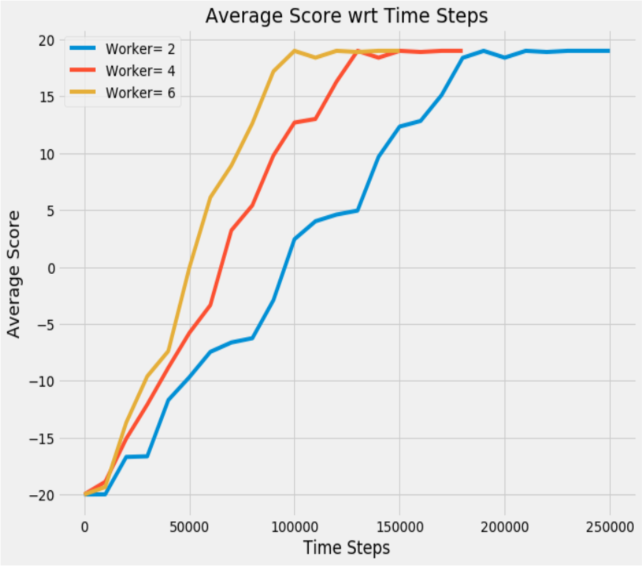

Figure 17. Average Episode Score wrt Time steps for different workers for Stale Model for Capsule Model