Deep Reinforcement Learning Algorithms to Play Atari Games and its Applications

M.E Project

Rohit R.R

Indian Institute of Science, Bangalore

Reinforcement Learning

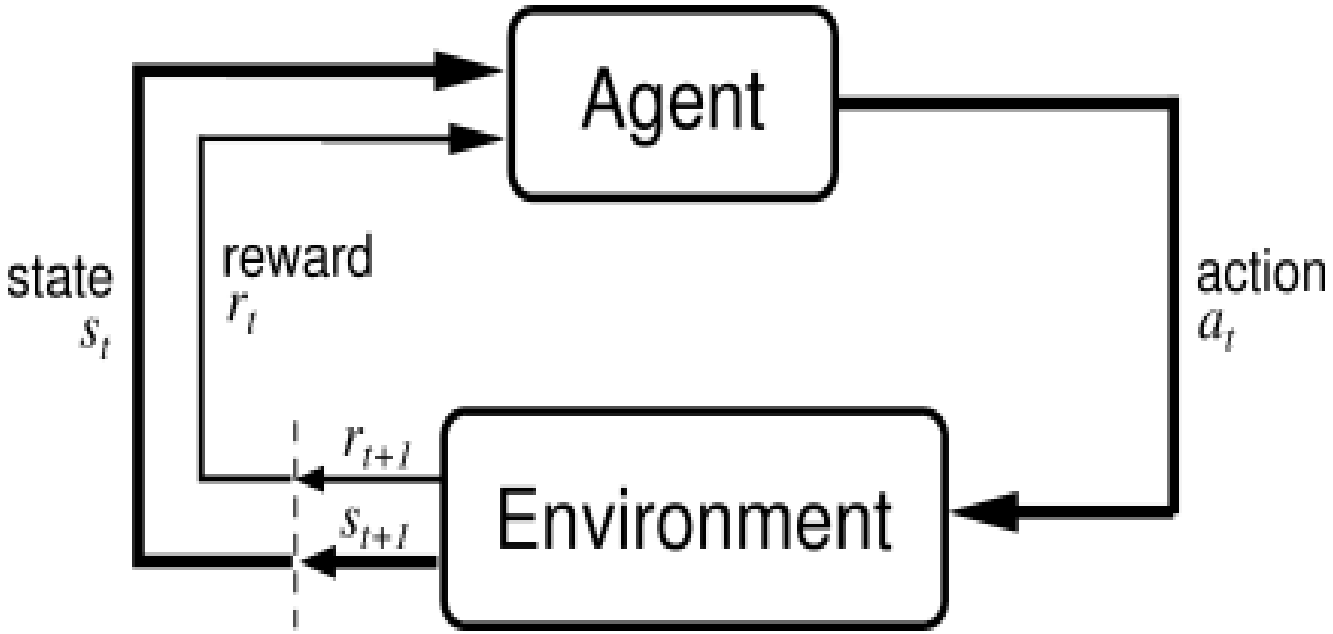


Figure: The Basic RL scenario

Introduction

- Most sensory inputs like images and audio are high dimensional data.
- Usage of Reinforcement Learning Algorithms have always been plagued by high dimensional large state spaces which make learning impractical.
- Even in cases where we can apply,
  1. State Space reduction techniques like aggregation
  2. Simple function approximators like linear function approximators

  the results would be unsatisfactory for complex tasks and would most likely needed to applied in a problem specific way.

Deep Learning Effects

- Enables us to get general AI agents that can learn variety of tasks in different environments.
- With Deep Neural Networks like Convolutional Neural Networks we are able to learn complex representations of the real world through images.
- Memory property of Recurrent Neural Networks help to capture temporal aspects of data.

Deep Reinforcement Learning

Learning ability of Deep Neural Networks is used along with reinforcement learning algorithms to handle complex tasks in high dimensional state spaces.

Atari Games and Benchmark for Deep RL

- Atari games are a set of computer games which we want to learn to play.
- Use only raw screen images for learning.
- Atari games provide an excellent way to benchmark and test the Deep RL algorithms.
  1. Considering 4 grayscale screen images of size 84x84 with 256 gray levels to be a state, we would $256^{84x84x4} \approx 10^{67970}$ possible game states.
  2. Simulations are easy and emulators are available for multiple languages.

Deep RL algorithms

1. Deep Q-Learning
2. Asynchronous Actor Critic
3. Trust Region Policy Optimization
4. Deep Recurrent Q-Learning

General Notations and definitions

At each time t$\epsilon\{1,2,..\}$

State $s_t \ \epsilon S$

Action $a_t \ \epsilon A$

Reward $r_t \ \epsilon R$

The environment's dynamics are characterized by

State transition probabilities

$P^a_{ss'} = P_r\{s_{t+1} = s' | s_t = s, a_t = a\}$

The total discounted reward we get over T time steps,

$R_t = \sum_{t'=t}^{T} \gamma^{t-1} r_t$

Q-value function,

$Q(s,a) = E\{R_T | s_t = s, a_t = a\}$

Policy,

$\pi(s,a) = P_r\{a_t = a | s_t = s\}$

Q Learning

Q-function is parameterized to form a function approximator - $Q(s, a; w)$

Algorithm

For every step t,

1. ϵ-greedy policy
   Choose a random action with probability ϵ
   otherwise, choose action $= \underset{a}{argmax} Q(s, a)$
2. Policy evaluation step:
   Reward target : $r_t + \gamma \left[\max_{a'} Q_{old}(s', a'; w^-) \right]$
3. Policy Improvement step:
   $L_i(w_i) = E_{s,a,r,s'} \left[(r_t + \max_{a'} Q_{old}(s', a'; w_i^-) - Q(s, a; w_i))^2 \right]$
   The gradient w.r.t to parameters $\nabla_{w_i} L(w_i) =$
   $E_{s,a,r,s'} \{ (r_t + \max_{a'} Q_{old}(s', a'; w_i^-) - Q(s, a; w_i)) \nabla_{w_i} Q(s, a; w_i) \}$
   Use stochastic gradient descent to update the w parameters.

Deep Q-Learning

Use a deep neural network as a function approximator in Q-Learning.

Issues in using Deep Neural Networks

- Unlike in supervised setting the "Data" we get to train the DNN is not IID ,so making DNNs to converge is a big challenge.
- RL algorithms generally are "slow" iterative algorithms and addition of DNNs further increase computational requirements and the time to train, so scaling and training fast is difficult.

Experience replay and **Fixed target networks** to avoid correlations in the data.

Deep Q-Network Architecture

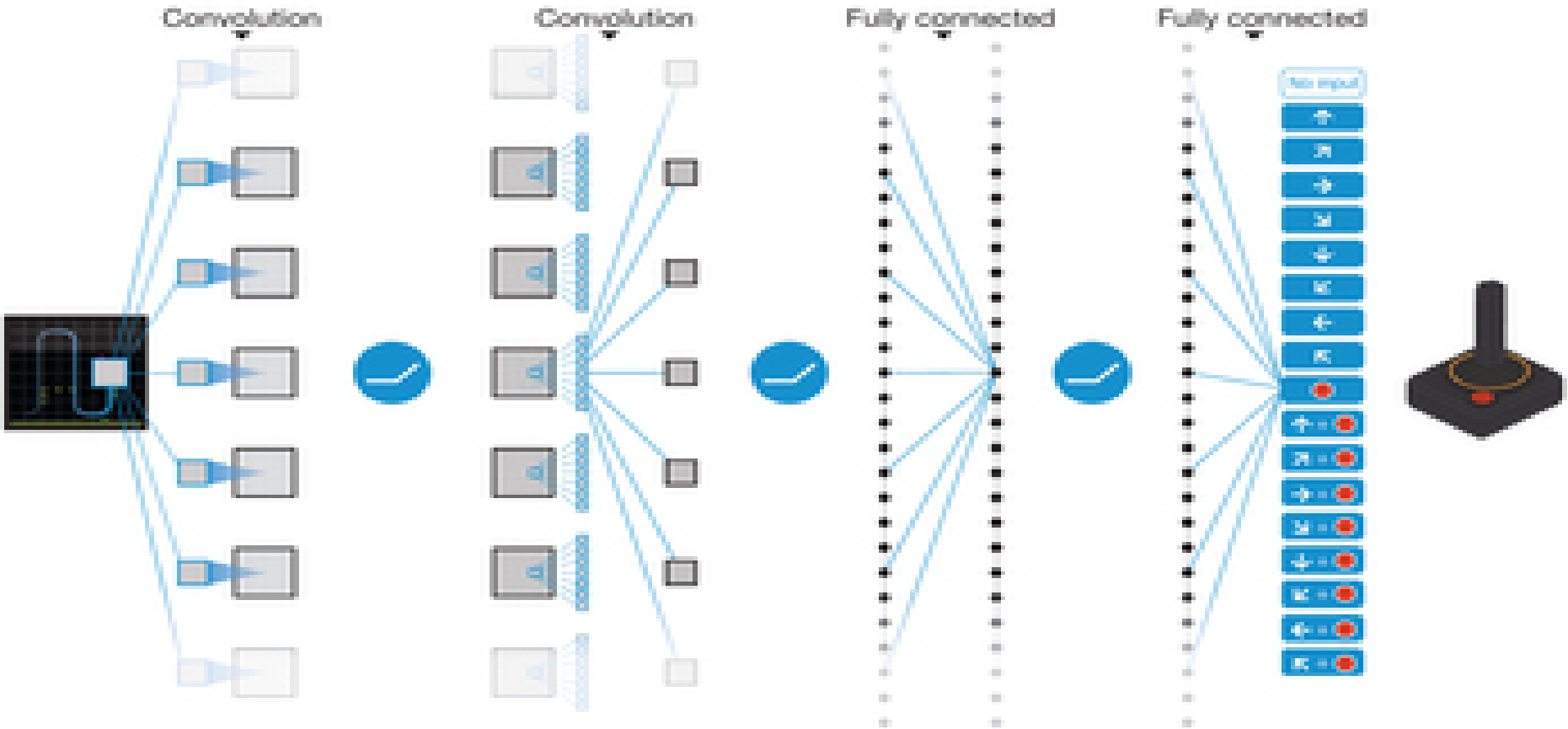


Figure: Convolutional Network used to map states to actions

[1] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

Policy gradients and Actor-Critic

The total "effectiveness" of a policy/objective function can be summarized with the following expression,

$$J(\pi) = \sum_s d^\pi(s) \sum_a \pi(a|s) R_{s,a}$$

$$\textit{where} \quad d^\pi(s) = \lim_{t \to \infty} Pr\{s_t = s | s_0, \pi\}$$

Now we parameterize the policy itself directly - $\pi_\theta(s, a)$ and we get

$$J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) R_{s,a}$$

$$\textit{where} \quad R_{s,a} = E\{R_T | s_t = s, a_t = a\}$$

Now the learning has been converted to an optimization problem where we need to find the parameters that maximizes J.

- We need to get $\nabla_\theta J(\theta)$ for the gradient ascent.

$$J(\theta) = E_{\pi_\theta}[R_{s,a}] = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) R_{s,a}$$

Using $\nabla_\theta \pi_\theta = \pi_\theta \frac{\nabla_\theta \pi_\theta}{\pi_\theta} = \pi_\theta \nabla_\theta log \pi_\theta$, we get

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(a|s) R_{s,a}]$$

Stochastic gradient ascent is applied with the sampled gradients to get the best policy parameters.

- In the Actor-Critic algorithm, $R_{s,a}$ is also paramaterized to a Q-function approximator and updated along with the policy updates.

Asynchronous Actor Critic

The main algorithm involved in [3] is the asynchronous actor critic. Multiple agents/threads are spawned where each agent has a separate value network and a policy network.

Algorithm for each Actor Learner thread

1. Plays for a batch of steps and then accumulates the gradients(actor-critic) asynchronously and parallely.

$$d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$$

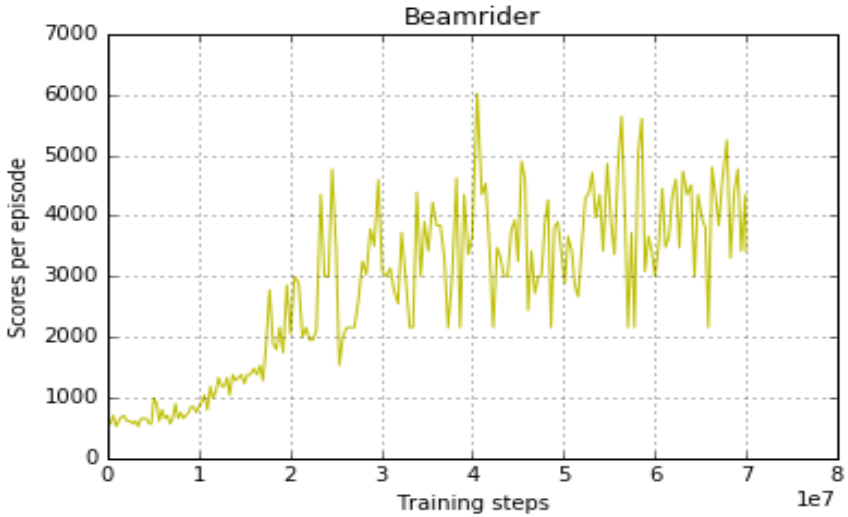
$$d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2/\partial\theta'_v$$

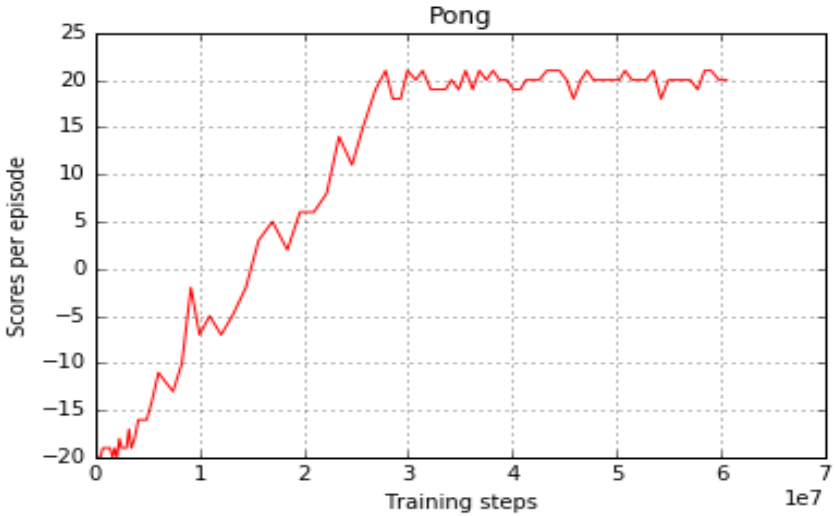
2. Use these gradients to perform actor-critic updates on the shared parameters .
3. Before getting the next batch of experiences the agents synchronize their parameters with the shared parameters.
4. Go to 1

Asynchronous Actor Critic Implementation

- The state/input to the networks are a set of the past 4 consecutive screen images/frames of the game.
- Network architecture : Figure
- Shared RMSprop(Tieleman and Hinton, 2012) was used for the stochastic gradient updates.
- Python does not support multi-threading so multiprocessing was used in python along with Tensorflow.
- Changes in memory/parameter sharing needed to be made.
- Trained each game for almost 2 days on a server with 32 processors.

Results





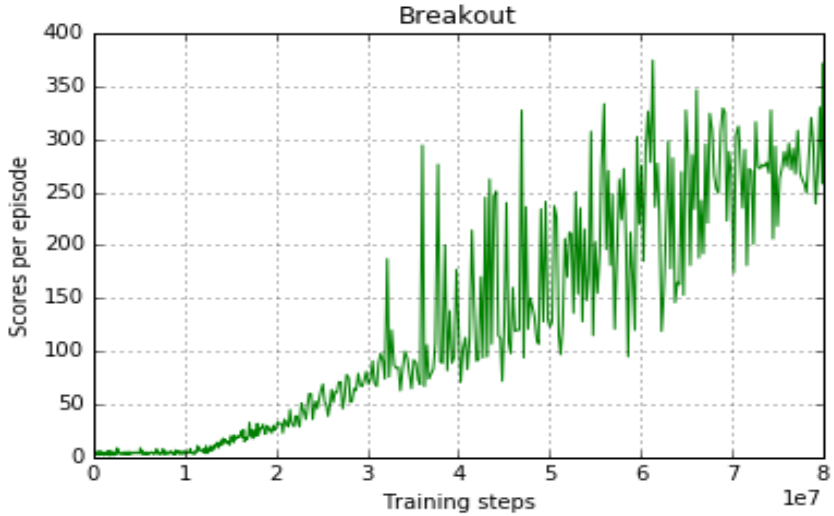


Figure: Training results using Asynchronous Actor Critic for 3 different games

Modifications and Experience Replay

Applied experience replay to asynchronous actor critic algorithm.

Algorithm Steps For each Agent/Process

Repeat

1. Take a few steps with actor policy.
2. Save the experiences in a replay memory buffer and pop out old ones.
3. Sample batch of experiences and accumulate gradients.
4. Send gradients for actor critic update of common parameters.
5. Synchronize parameters with common parameters.

until learnt

Result

- Does not improve results. Slower due to extra memory computations.
- Parallelism of the Asynchronous Actor Critic was enough to stabilize.

Batch Normalization

- [2] Tries to reduce Internal Covariance Shift occurring for each subnetwork. For a layer with d-dimensional input $x = (x^{(1)}, ..., x^{(d)})$,

$$\widetilde{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

- Constraints them to linear regime for sigmoid activations.
- A pair of parameters $\gamma^{(k)}$, $\beta^{(k)}$ for each activation introduced to restore representation power of the network, $y^{(k)} = \gamma^{(k)} x^{(k)} + \beta^{(k)}$

Result

Does not seem to work well with RL scenarios where input data is non-IID. Normalizing with mean and variance of input data creates more correlations.

[2] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

Weight Normalization

- [3] Has shown improvements in training of networks for noisy data. So considered the **RL scenario to be a noisy one.**
- It reparameterizes the weights corresponding to the activation function input as,

$$w = \frac{g}{||v||} v$$

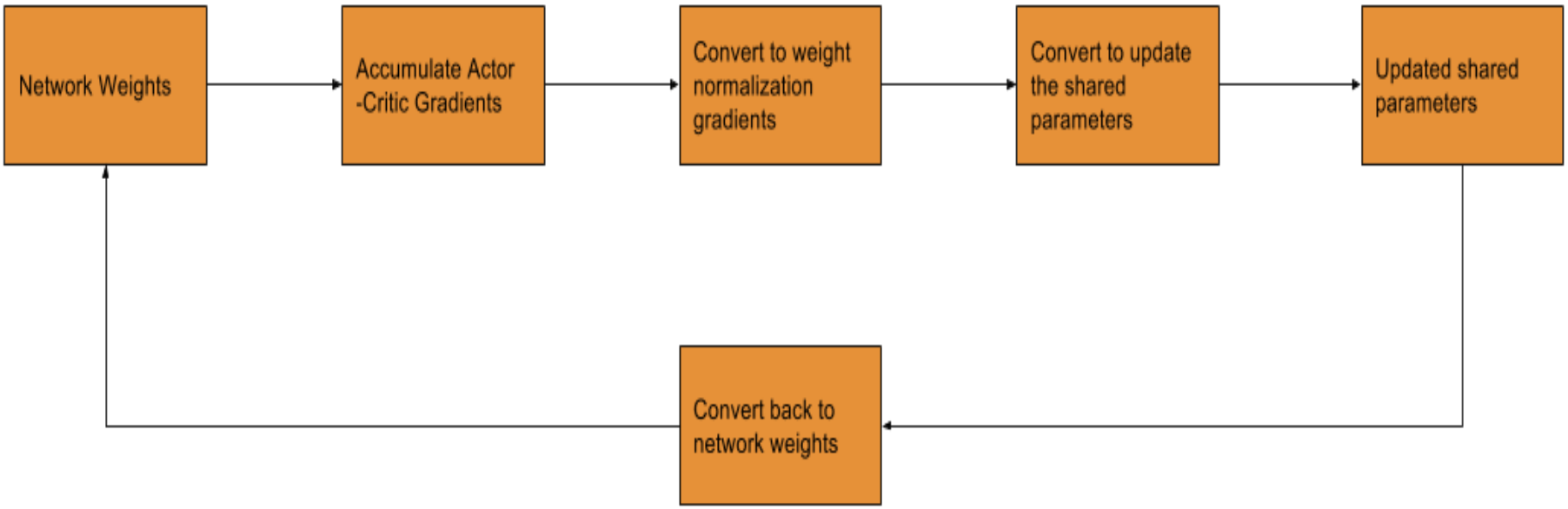
- The extra variables do not increase computation costs much as the gradients can be calculated with the usual w gradients,

$$\nabla_g L = \frac{\nabla_w L . v}{||v||}, \quad \nabla_v L = \frac{g}{||v||} \nabla_w L - \frac{g \nabla_g L}{||v||^2} v$$

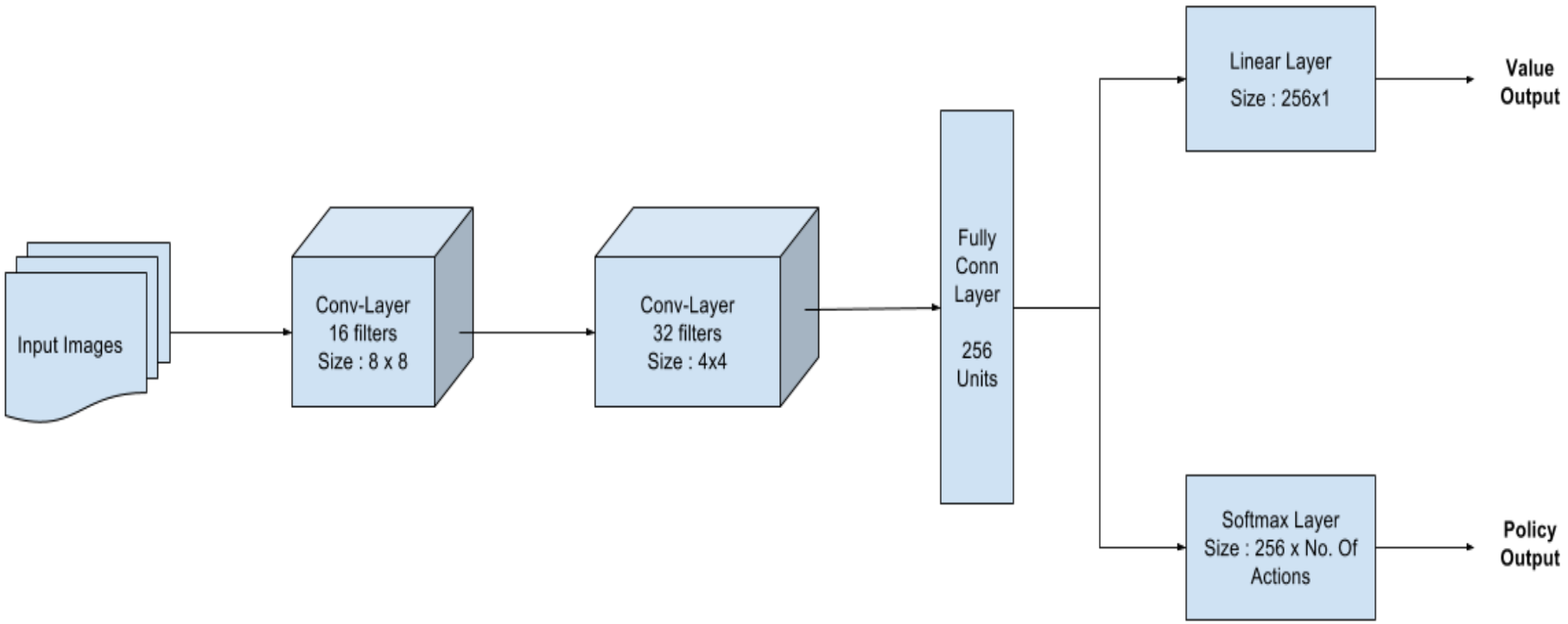
- It scales the gradient and projects the gradient vector away from the current weight vector.

[3] Tim Salimans and Diederik P. Kingma. "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks". In: *CoRR* abs/1602.07868 (2016).

Modified Algorithm



Neural Network Architecture



Experimental Results

- Improved upon the Asynchronous Actor Critic Algorithm.

- More rewards in lesser steps. Training is almost twice as fast as before.

Future Work

Improve algorithms

- Initial Part of Learning is slow, to improve use Transfer Learning or Supervised Learning for this.
- Use better exploration techniques like Thomson's Sampling or UCB.
- Separate background from foreground to reduce the number of parameters.

Application

- Apply to the problem of allotting cloud computing resources dynamically to clients.
- Ideal candidate for application due to availability of simulator(CloudSim) that can train data hungry networks.