CS419 Project Report (Respawn)

Team Members -
Krishna Wadhwani (160010031)
Kushal Yadav(160010011)
Varun Sule (160010013)
Shubham Punjabi (160010035)

Problem Statement: Analyze how well different models perform when enabling an agent to play the game of CartPole, as defined in the gym module provided by Open AI

For this purpose, we have used the above four models

1. Feed Forward Neural Networks (without using Reinforcement Learning)
2. Reinforcement Learning using Policy-Gradient based Algorithms
3. Reinforcement Learning using Deep Q-Learning based Algorithms
4. Reinforcement Learning using Actor-Critic method based algorithms

All the codes and results can be found in our github repository

Reinforcement Learning

Reinforcement Learning is a type of Machine Learning, in which the algorithm allows the software agent (the machine) to automatically determine what is the best path that can be undertaken by the agent in a given context, based on a specific pre-decided rewards, actions and states.

Motivation for deciding Reinforcement learning -

The biggest motivation behind tackling a problem using reinforcement learning is it's capacity to tackle new and challenging problems which don't have an available data set to go with it. Reinforcement Learning allows the machine or software agent to learn its behaviour based on feedback from the environment. This behaviour can be learnt once and for all, or keep on adapting as time goes by.

Applications of Reinforcement learning -

The possible applications of Reinforcement Learning are abundant, due to the genericness of the problem specification. As a matter of fact, a very large number of problems in *Artificial*

Intelligence can be fundamentally mapped to a decision process. This is a distinct advantage, since the same theory can be applied to many different domain specific problem with little effort.
In practice, this ranges from controlling robotic arms to find the most efficient motor combination, to robot navigation where collision avoidance behaviour can be learnt by negative feedback from bumping into obstacles. Logic games are also well-suited to Reinforcement Learning, as they are traditionally defined as a sequence of decisions: games such as poker, backgammon, othello, chess have been tackled more or less successfully.

Gym Environment for the problem - CartPole-v1

As discussed above, the agent tries to learn while performing actions in the environment. For our problem statement open ai has a predefined environment .
A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

The Gym environment provides some predefined functions which do tasks like providing the training attributes for the neural nets, rendering the environment, resetting the environment etc.

Now we will discuss the different implementations of the same problem statement and compare the results obtained from each model.

Feed Forward Neural Networks (without using Reinforcement Learning)

> Goal: To train a Neural Network with 5 fully connected layers to train over a sample dataset of states and actions, sampled from game runs and test the model on 100 episodes of the game to see how effective the model is.

Part-A] Creation of the sample training dataset

The first function in the code runs multiple renditions of the game to obtain sets of observation states and actions, that are stored in an array, while a net reward score is computed alongside. In the next step, those games which have a net score above a certain threshold are selected, and the corresponding observation and action arrays are added to the training data. This step ensures that the training dataset does not contain useless game renditions with very low score values, training on these games will cause the model to train a classifier that gives predictions along the same lines. Note that the output is taken as one-hot, insead as binary, so that generalization would be easier in case of complicated games with more actions.

In the code that we have executed, the number of games that were initially considered was 10000, and the minimum score threshold was kept at 45. We found that these parameters ensured that around 500-600 games were added to the training data, to maintain consistency with the other methods.

Part-B] Defining and Training the Neural Network Model

The Neural Network being used here is a feed-forward neural network with 5 fully connected layers, having 128,256, 512, 256, 128 neurons respectively. Each layer has a dropout value of 0.6. We have used the tflearn library of tensorflow to define the network and for subsequent training. Subsequently, the function tflearn.layers.estimator.regression has been used to apply gradient descent optimization to the model, with the default value of optimizers and loss function (adaptive moment estimation and cross-entropy loss respectively)

For the actual training part, the training dataset obtained from the previous function is split into two separate arrays and the tflearn.model.DNN.fit function is used to execute the training process. The 2 inputs for model.fit are obtained as follows:

I. An array of states taken from the training data obtained earlier, and it corresponds to the input layer to the neural network
II. An array of actions taken from the training data obtained earlier, corresponding to the model targets obtained after applying regression.

To prevent overfitting, the code has been executed for 5 epochs.

Part-C] Testing new games on the model

We begin the final part of the code by following tasks that are similar to what was done while creating the training examples, with the major difference being that each action is predicted from the neural network, by calling the tflearn.model.DNN.predict function. This function takes the previous observation as the input and predicts the next action based on the model. By adding up the reward values for each game, the score can be obtained, and the mean values of the score are compared with those obtained from the other models

Code and Experimentation

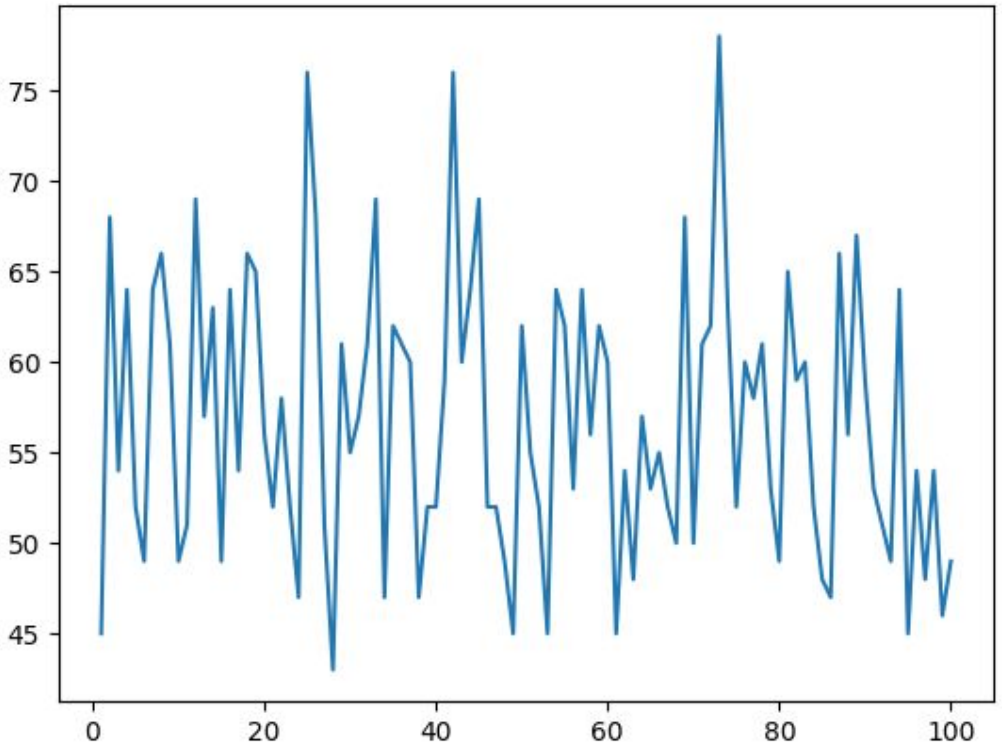
The code has roughly 140 lines, and uses tensorflow and tflearn libraries for ease of handling the neural networks. The code has been trained on 583 episodes, and 100 episodes of the game have been tested. The default time-limit for each game is 500, though in almost all the cases, the game ends much before this limit is reached.

Results:

On carrying out the above experimentation, the following results are obtained:
(Note: Training: ~30000 iterations per epoch, 5 epochs)

Average Score: 56.79
This score is low as compared to the required threshold of 195, as defined in gym.
Runtime: 197.87s on a Ubuntu 16.04 Virtual Machine with 1024 MB RAM

Graphs of Accuracy and Loss parameters from Tensorboard (n_epochs=5)



Graph of the score for each game rendition during testing (n_epoch=5)

Increasing the epoch size above 5, leads to an overfitted model, wherein it can be observed that the losses are not decreasing over time.

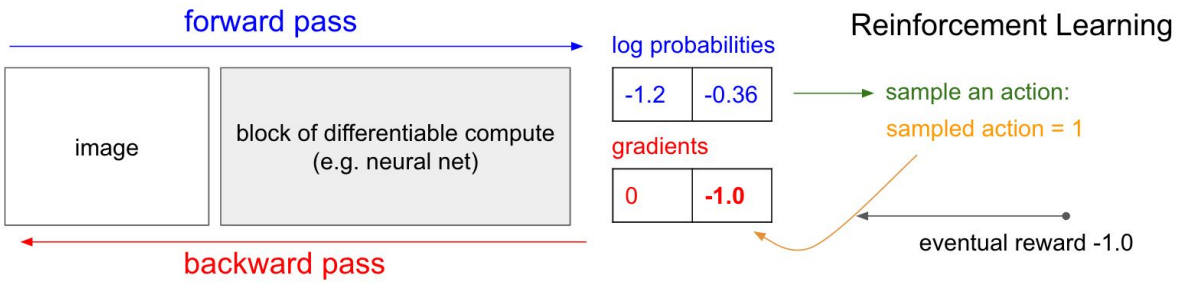
Also, accuracy can be improved by increasing the number of layers, but the runtime for such networks will blow off as expected.

Note: This example used a threshold to obtain the training sample, in a sense, we are forcing the game to obtain better training examples, instead of using all randomly obtained samples. This makes the approach less natural as compared to the RL-based approaches that have been employed by us.

Reinforcement Learning Policy-Gradient based algorithm

Algorithm Description

In policy gradient algorithm, we directly learn a policy function that maps state to the corresponding action. Our neural network outputs a probability distribution over the action space from which we can chose the action with maximum probability for a deterministic case. During training, we treat the environment as a stochastic process and accordingly let the agent explore different policies. We used Monte Carlo learning, that is, we used end of the episode reward in the loss function to update our network's parameters.



Related Literature

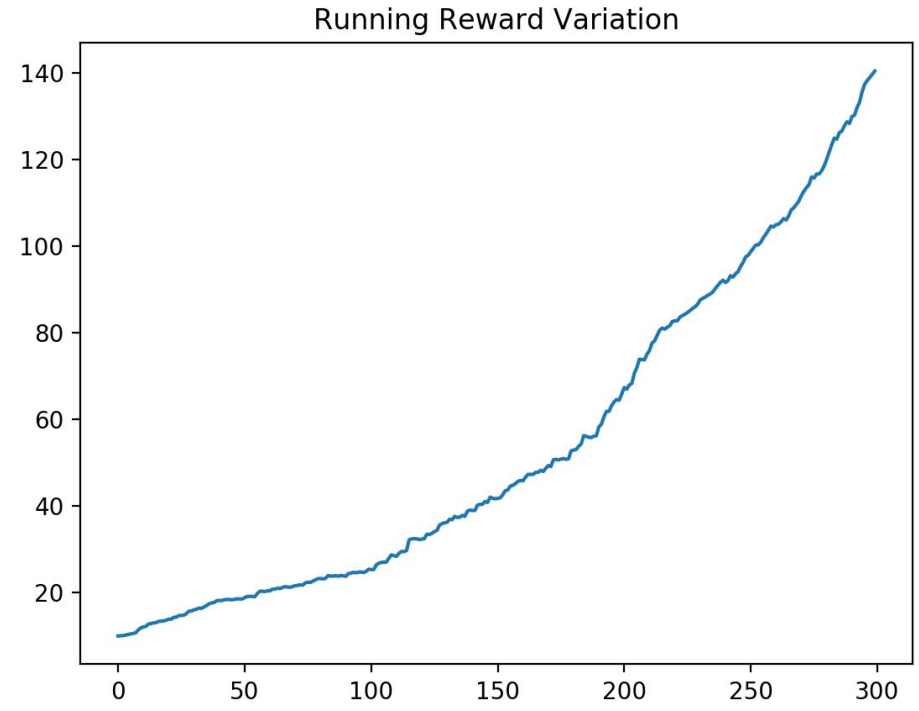
https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf

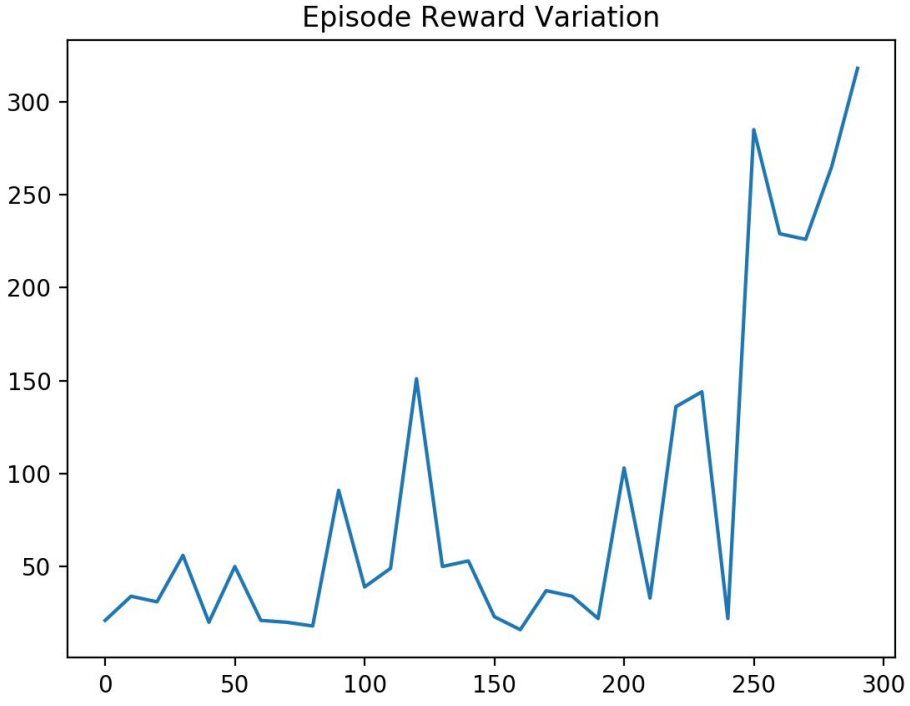
http://karpathy.github.io/2016/05/31/rl/

Experimentation

It is roughly 200 line code written in tensorflow. Other dependencies include gym, matplotlib and numpy. The network comprises 3 fully connected layers and is trained on 500 episodes with timestep of 500. The code takes about a minute to run on an i7 CPU (without GPU). The trained Model's performance was evaluated as an average over 100 episodes.

Results





The average reward over 100 trials is found to be around 150 to 180, implying that this algorithm does a very good job on learning to play the CartPole environment.

Effort

Challenging part of this was in keeping tracks of various terms and rewards involved. THe program has running_reward, episode_reward, maximum_reward, discounted_episode_rewards and the corresponding lists for each of them. Also the code was extremely messy at first but has been cleaned and commented to a considerable extent.

Reinforcement Learning using Deep Q-learning based algorithm

1 - Main goal

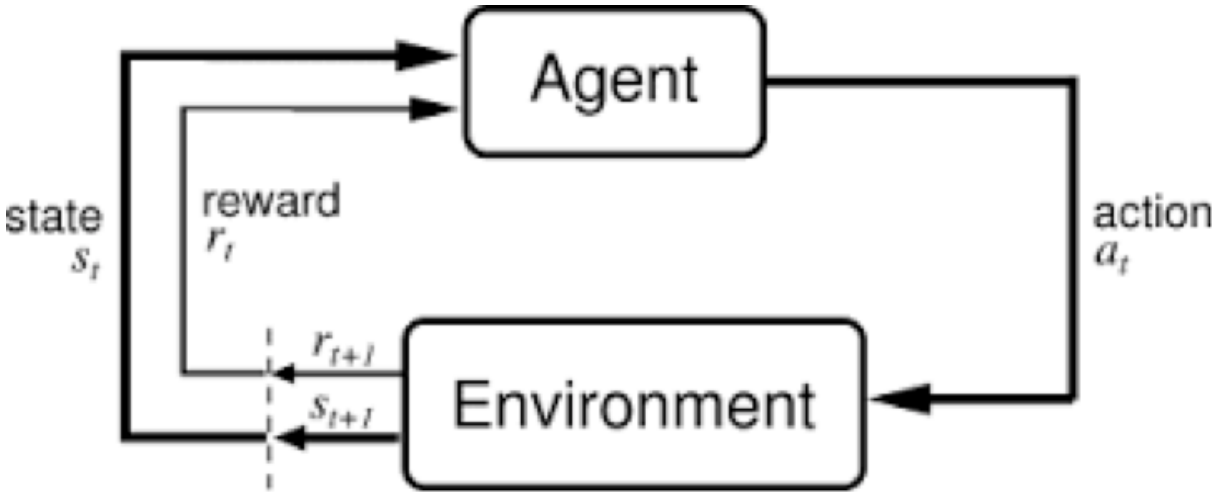
The main goal of the following snippet is to solve the cartpole(inverted pendulum) problem using Deep Q learning

2 - Related Literature

https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8
https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288

3 - Approach and Experimentation

DQN Algorithm -



The main idea behind Q-learning is that if we had a function of the form -

Q*: State x Action -> R

that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards:

$$\pi*(s)=\text{argmax } Q*(s,a) \text{ wrt } a$$

We now need to find Q*.

$$Q^{\pi} = r + \square Q^{\pi}(s', \pi(s'))$$

This is our update function, as every Q function of each policy follows Bellman's equation
And the temporal difference error

$$\delta = Q(s,a) - (r + \square \max Q(s',a))$$

And for this we use the Huber Loss.

$L = 0.5a^2$ if $a \leq \delta$ else $\delta(|a| - 0.5\delta)$

The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large - this makes it more robust to outliers when the estimates of Q are very noisy.

$$Q^{\pi}(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | s_t, a_t]$$

Q value for that state given that action

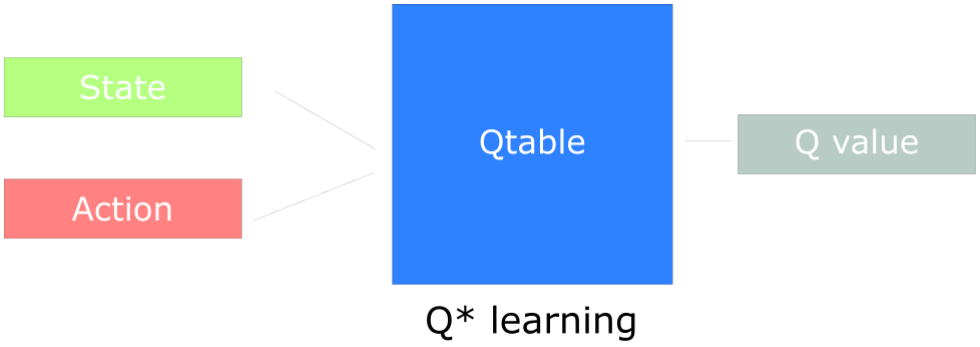
Expected discounted cumulative reward ...

given that state and that action

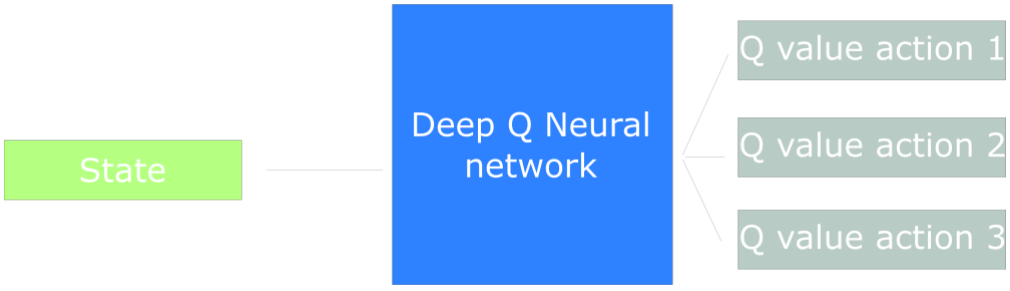
Some terms -

a) Agent: An **agent** takes actions; for example, a drone making a delivery, or Super Mario navigating a video game. The algorithm is the agent.
b) Action (A): A is the set of all possible moves the agent can make. An **action** is almost self-explanatory, but it should be noted that agents choose among a list of possible actions.
c) Discount factor: The **discount factor** is multiplied by future rewards as discovered by the agent in order to dampen these rewards' effect on the agent's choice of action.
d) Environment: The world through which the agent moves. The environment takes the agent's current state and action as input, and returns as output the agent's reward and its next state.
e) Reward (R): A **reward** is the feedback by which we measure the success or failure of an agent's actions. For example, in a video game, when Mario touches a coin, he wins points.
f) Policy (π): The **policy** is the strategy that the agent employs to determine the next action based on the current state. It maps states to actions, the actions that promise the highest reward.

g) Value (V): The expected long-term return with discount, as opposed to the short-term reward R. Vπ(s) is defined as the expected long-term return of the current state under policy π.



Q* learning



Deep Q* learning

Replay Buffer - At each time step, we receive a tuple (state, action, reward, new_state). We learn from it (we feed the tuple in our neural network), and then throw this experience. As a consequence, it can be more efficient to make use of previous experience, by learning with it multiple times. Hence we create a “replay buffer.” This stores experience tuples while interacting with the environment, and then we sample a small batch of tuple to feed our neural network.

Pseudo Code -

1. Initialize Q-values ($Q(s, a)$) arbitrarily for all state-action pairs.
2. For life or until learning is stopped...
3. Choose an action (a) in the current world state (s) based on current Q-value estimates ($Q(s, \cdot)$).
4. Take the action (a) and observe the the outcome state (s') and reward (r).
5. Update $Q(s, a) := Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$

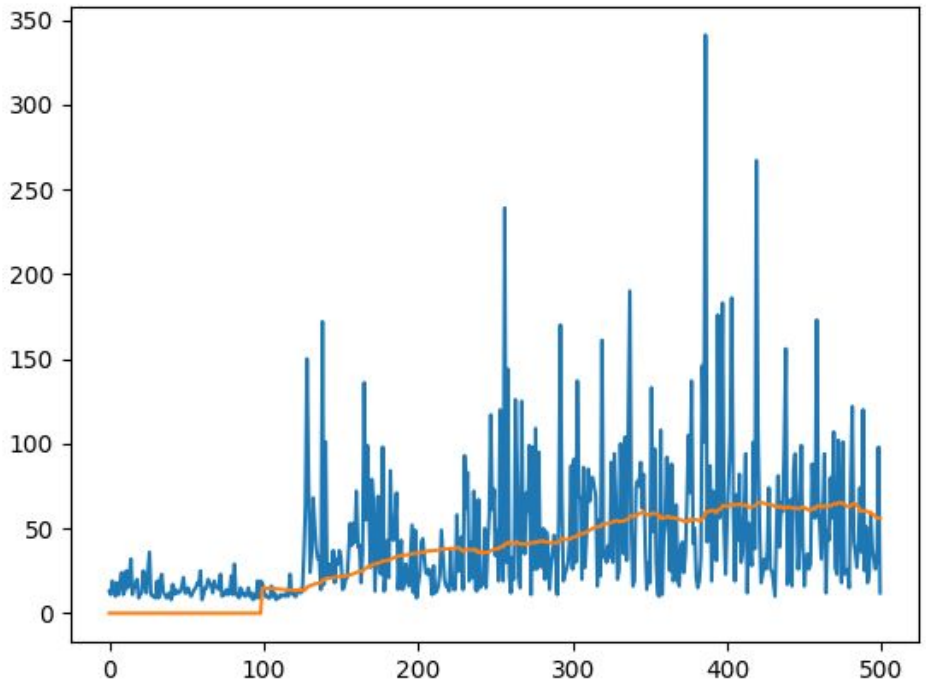
Optimization - First samples a batch, then concentrates all tensors, computes Q(s,a), computes V-Q, and then finds the loss. We also use target network to compute $v(s_{t+1})$ for stability. The target network has its weights kept frozen most of the time, but is updated with the policy network’s weights every so often. This is usually a set number of steps but we shall use episodes for simplicity.

Torchvision -

We have used torchvision.transform. This composes several transforms together, and make it easy to compose image transforms.
We have used these functions -

a) ToPIL - Convert a tensor or an ndarray to PIL Image.
b) Resize - Used for resizing
c) ToTensor - Converts PIL,ndarray into a tensor

Training -



The graph is Time vs Episodes, and the yellow line shows mean for the 100 previous episodes.We see that we hit a mean reward of 50, with a maximum time of 343.

Exact code details -

Run on python 3.5.2 with gym environment. We have 223 lines of code. A 500 episode takes nearly 17 minutes to run

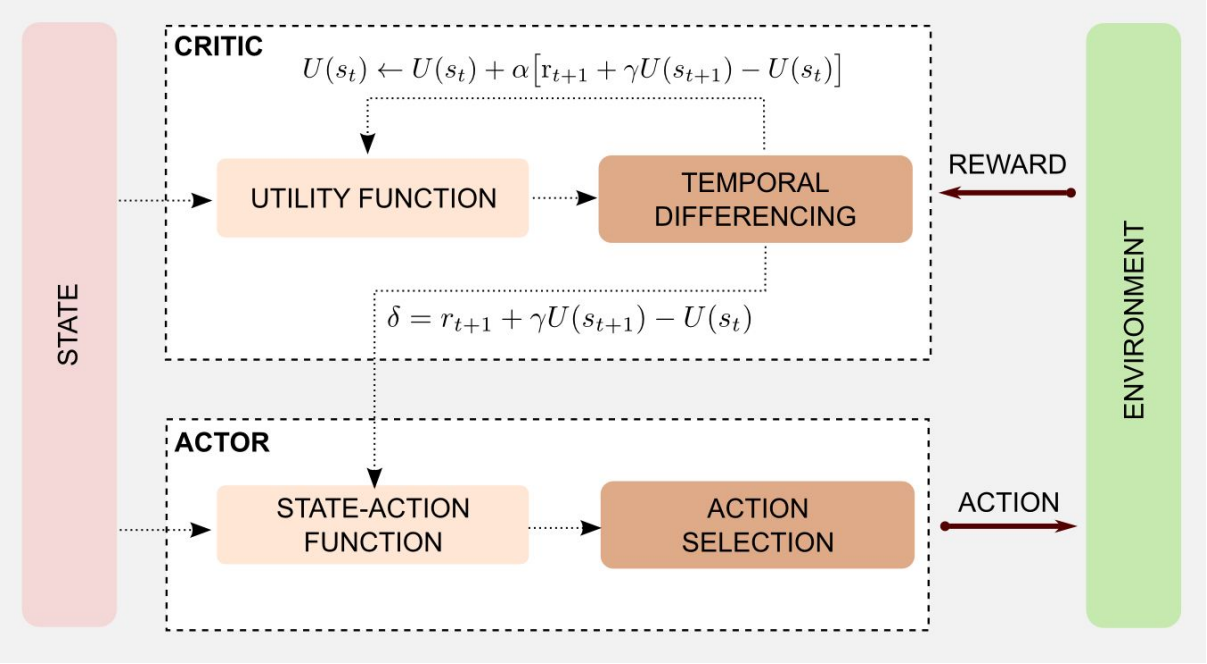
The most difficult part - Optimization, and getting the best convolution parameters as well as starting parameters. Another issue face was installing of all packages on Windows

** The code is very similar to one on the pytorch tutorial, as it was the initial start point, but the code was built from scratch, with the understanding of each concept, and iterations. Only the initial starting values were taken from there

Reinforcement Learning Actor-Critic based algorithm

Algorithm Motivation and Description

While both Q-learning and policy gradients algorithms work well in many cases, they both have their sets of limitations. Q-learning cannot be used for stochastic processes or processes with continuous action space while policy gradients can be used in these cases but they suffer from the problem of credit assignment, that is, the algorithm does not know which of its action in the episode contributed to its reward as in Monte Carlo method, we are using end of episode rewards as a feedback to the agent.



So most of the state of the art architectures use a hybrid network architecture combining Policy gradient and Q-Learning methods. We explored one such hybrid architecture that is the A2C or the Advantage Actor-Critic Algorithm. In this algorithm, we have 2 Neural Networks - Actor and Critic that are simultaneously trained. Actor learns a state to action mapping or the policy function whereas Critic measures how good is the action taken or the Q function.

The way this works is that instead of waiting until the end of the episode ;like we do in Monte Carlo learning, we make an update at each step using TD error. At each timestep, both the

networks are fed the state of the environment at that time. The Actor maps that state to the corresponding action and receives a new state and reward. With the new state and reward, the critic computes the value of taking that action at that stage. The Actor updates its weight using this q-value. With this updated weight, the Actor produces the next action which is used by the critic to update its own weight.

As value based functions have high variability, A2C algorithm defines an advantage function that is as the difference of q-value for an action in a given state and average value of that state.

$$A(s,\ a)\ =\ Q(s,\ a)\ -\ V(s)$$

This function tells us the improvement compared to the average, the action taken at that state is. We can use TD error as an estimator of the advantage function.

Related Literature

https://arxiv.org/pdf/1602.01783v1.pdf

https://medium.freecodecamp.org/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d

Code

The code is around 150 lines and written using pytorch. Other dependencies include gym, numpy, matplotlib and collections. The Actor and critic neural networks comprise 2 fully connected layers as CartPole is a relatively simple environment. The networks have a common 1st layer and this works as cartpole is a relatively simple environment. The agent was trained for 500 episodes and maximum timestep of 500. The trained Model's performance was evaluated as an average over 100 episodes.

As the architecture and environments are simple, the code took roughly 25 seconds on an i7 CPU (without using a GPU)

Results

(The running reward was calculated as running_reward(t+1) = running_reward*0.99 + 0.01*R(t) where R(t) is episodic reward at timestep t and R(0) = 10)


Running Reward Variation
140
120
100

The final average reward varies for different environment conditions. The average reward over 100 games was found to be above 180 and even reaching the maximum 200 in many cases.

Thus this algorithm outperforms our other approaches and has a faster convergence than the other algorithms that we have tried. The running reward is also higher than what we obtained for policy gradient method.