Addition of Rainbow and Soft Actor-Critic to RL codebase

Basic Information

Name: Nishant Kumar
University: Indian Institute of Technology (BHU), Varanasi
Field of Study: Electronics Engineering
Time study was started: July 2018
Expected Graduation date: July 2022
Degree: Bachelor of Technology
Year: Sophomore
Github: nishantkr18
LinkedIn: nishantkr18
IRC nick: nishantkr18
Email Address: nniishantkumar@gmail.com
Phone number: (+91) 7665179120
CV: https://drive.google.com/open?id=1A_6-N_oEfgaRnlC2crXyb5H42p5xzf0k
Timezone: Indian Standard Time (UTC +5:30)

The Project Proposal

Objective

While going through the Reinforcement Learning codebase of MLpack, I noticed that a lot of state-of-the-art algorithms are missing. So after comparing various algorithms and brainstorming about their methods of implementation, I came up with the **Rainbow** (Hessel et al., **2017**) and **Soft Actor-Critic** (Haarnoja et al., **2019**) as the most in-demand and recent algorithms, whose implementation in mlpack would be crucial
So here are the details of what I expect to have accomplished at the end of the summer.

➔ Improving the current QLearning implementation.
➔ Implementing Rainbow as an improvement on DQN
➔ Writing test cases for each of the implementations
➔ Implementing Soft Actor-Critic (SAC) for continuous action space, along with its tests
➔ Creating detailed docs for all the above implementations
➔ Creating necessary environments, for proper testing of algorithms above (after discussion with mentor)

Background Info

This includes descriptions of the algorithms / data structures / ideas I plan to implement. I made sure to detail background information, such that a person who is reasonably familiar with machine learning and mlpack will be able to understand the description without needing to consult other references.

***PART 1**: For the first Part, I would like to add extensions to the Deep Q Learning implementation, and use them together to implement Rainbow.*

Rainbow is a DQN based off-policy deep reinforcement learning algorithm with several improvements.
In fact, Rainbow combines seven algorithms together:
(1) **DQN (Deep Q-Network) :** Applying stochastic gradient descent to minimize the loss given by:

$$(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2 \ ,$$

(2) **DDQN (Double Deep Q-Network) :** Conventional Q-learning is affected by an overestimation bias. The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation.

$$(R_{t+1}+\gamma_{t+1}q_{\overline{\theta}}(S_{t+1}, \underset{a'}{\operatorname{argmax}}\, q_\theta(S_{t+1}, a'))-q_\theta(S_t, A_t))^2$$

(3) **Prioritized Experience Replay (PER) :** DQN samples uniformly from the re-play buffer. Ideally, we want to sample more frequently those transitions from which there is much to learn. Thus, PER samples transitions with probability p t relative to the last encountered absolute TD error:

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t) \right|^\omega$$

(4) **N-Step Q-Learning** : Using Q values with N-step Returns, which are more accurate than 1-step Returns

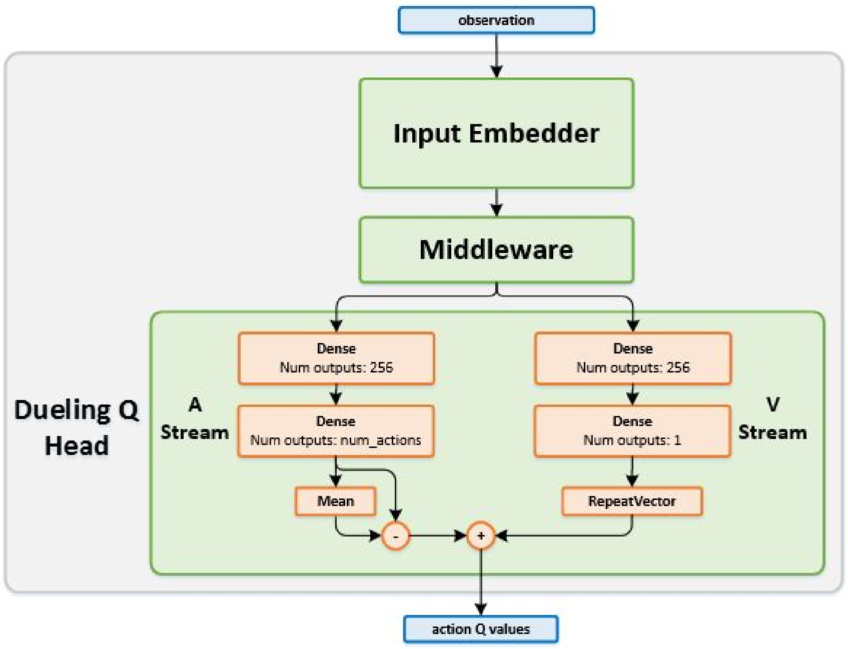
$$(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\overline{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2.$$

Above equation evaluates loss from Q values with N-step Returns.
(5) **Noisy Network**: Combines the final output linear layer of Q-network with a noisy stream

$$\boldsymbol{y} = (\boldsymbol{b} + \mathbf{W}\boldsymbol{x}) + (\boldsymbol{b}_{noisy} \odot \epsilon^b + (\mathbf{W}_{noisy} \odot \epsilon^w)\boldsymbol{x}),$$

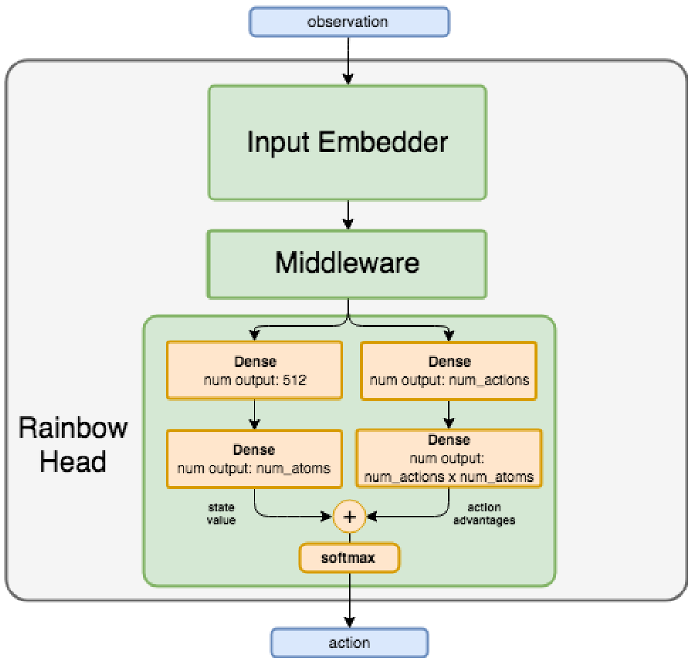
where **b** and **w** are random variables, and the dot symbol denotes the element-wise product. This transformation can then be used in place of the standard linear **y** = **b + Wx**.
(6) **Distributional RL** : Directly learns a distribution of Q values other than average Q values.

(7) **Dueling Q-Network** : DQN's architecture is branched to have two branches. One branch outputs Value, and the other branch outputs Advantages

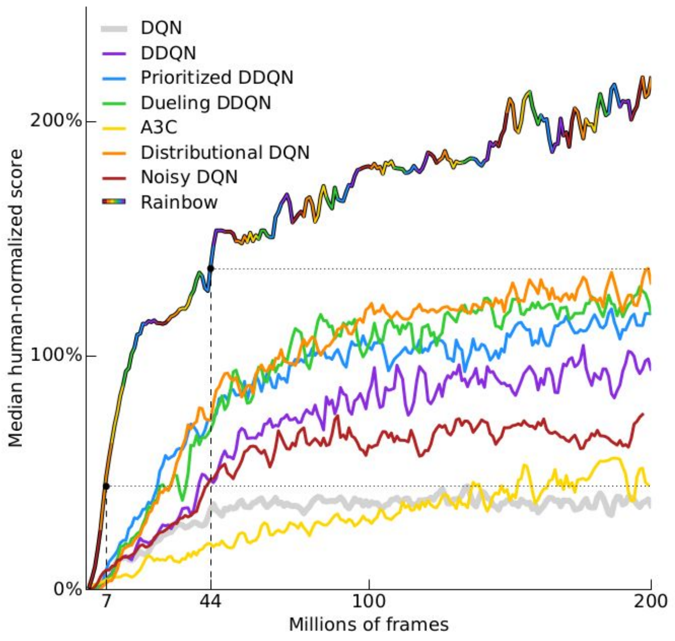


(source)

The (1), (2) and (3) are already implemented while the others are left.
For a complete implementation of Rainbow, the remaining extensions are to be added in such a way that all of them can work together.
It is clearly seen that all the extensions working together can have a very good performance, much better than vanilla DQN.



The Rainbow Network architecture (source)



Improvements of Rainbow over vanilla DQN

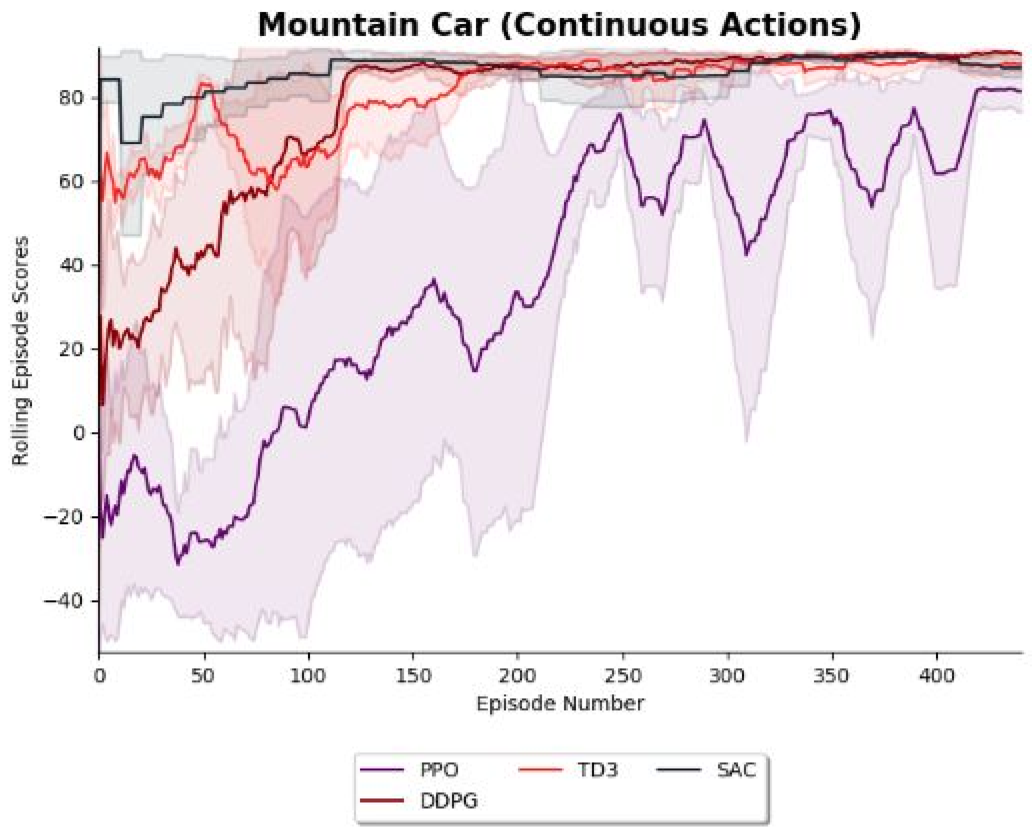
(source)

***PART 2**: For the second part, I would like to implement an off-policy model-free algorithm by the name of SAC (Soft Actor-Critic) for Mlpack*.

Soft Actor-Critic is an off-policy actor-critic deep RL algorithm based on the maximum entropy reinforcement learning frame-work.

The algorithm not only boasts of being more sample efficient than traditional RL algorithms but also promises to be robust to brittleness in convergence. An actual 4 legged bot called Minotaur Robot(Link) has been shown to move and generalize well to unseen environments using SAC.

Some of the most successful RL algorithms in recent years such as Trust Region Policy Optimization (TRPO), Proximal Policy Optimization (PPO) and Asynchronous Actor-Critic Agents (A3C) suffer from sample inefficiency (requires lots of training steps). This is because they learn in an "on-policy" manner, i.e. they need completely new samples after each policy update.
In contrast, Q-learning based "off-policy" methods such as Deep Deterministic Policy Gradient (DDPG) and Twin Delayed Deep Deterministic Policy Gradient (TD3PG) are able to learn efficiently from past samples using experience replay buffers. However, the problem with these methods is that they are very sensitive to hyperparameters and require a lot of tuning to get them to converge.



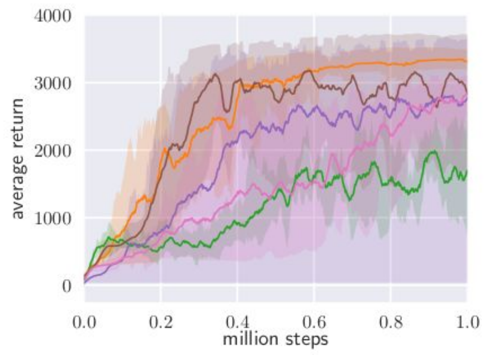
(source)

Soft Actor-Critic follows in the tradition of the latter type of algorithms and adds methods to combat the convergence brittleness.
It uses the following features:

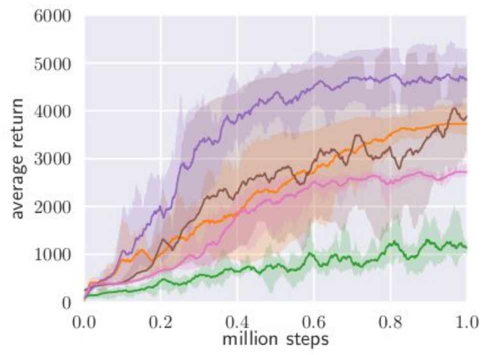
- ➔ The biggest feature of SAC is that it uses a modified RL objective function. Instead of only seeking to maximize the lifetime rewards, SAC seeks to also maximize the entropy of the policy.
- ➔ Maximizing entropy means to succeed at the task while **acting as randomly as possible** in order to ensure better exploration.
- ➔ This makes this approach very stable, achieving very similar performance across different random seeds.

Pros of SAC:

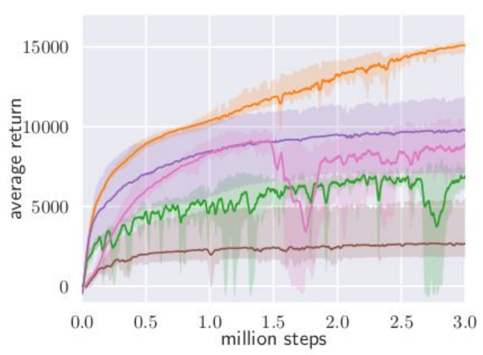
- ➔ SAC is the off-policy approach for RL tasks involving continuous tasks, owing to its superior sample efficiency as compared to TRPO and PPO.
- ➔ SAC overcomes the brittleness problem in many RL algorithms by encouraging the policy network to explore and not assign a very high probability to any one part of the range of actions.



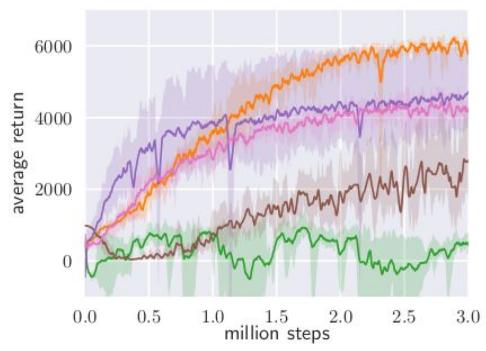
(a) Hopper-v1



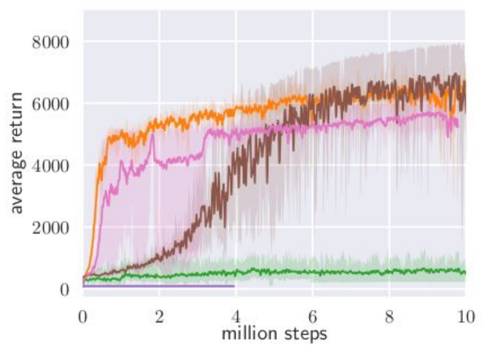
(b) Walker2d-v1



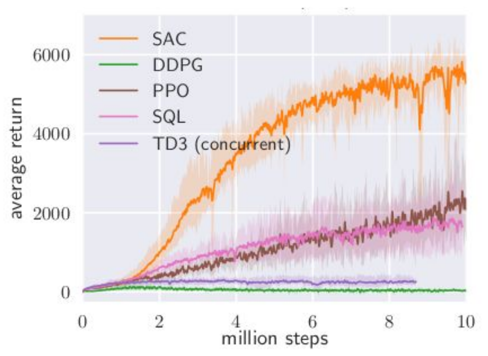
(c) HalfCheetah-v1



(d) Ant-v1



(e) Humanoid-v1



(f) Humanoid (rllab)

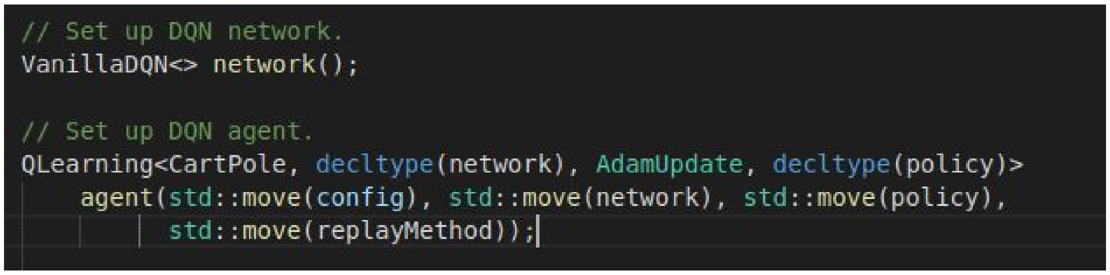
(source)

It is clear from the above graphs, that SAC (yellow) performs consistently across all tasks and outperforms both on-policy and off-policy methods in the most challenging tasks.

Part 1: Full Implementation of Rainbow

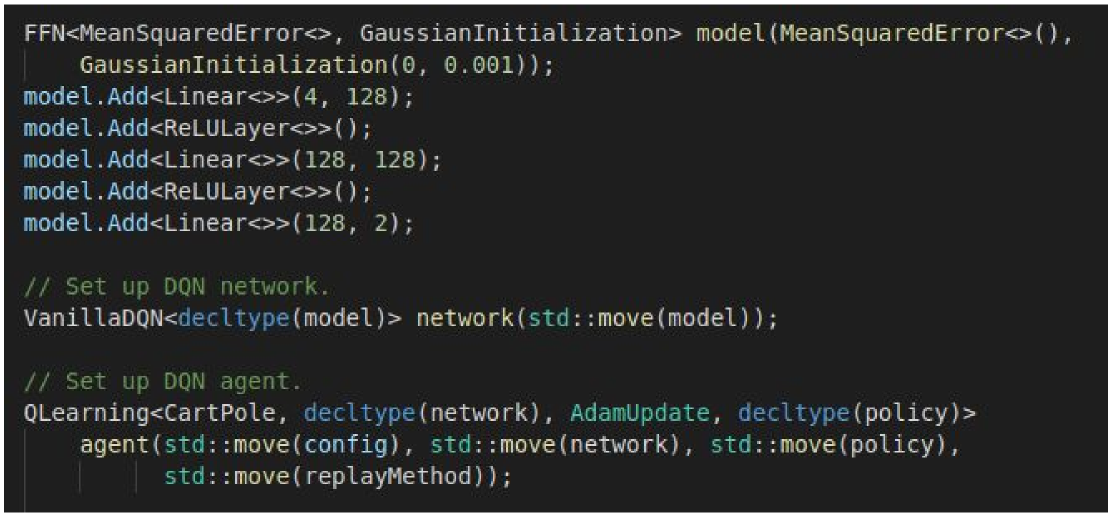
Abstract

Regarding Rainbow: I propose to add separate classes in corresponding files, as in #2317, for Dueling, Noisy, (a separate Categorical/Distributional without including Dueling and Noisy, if necessary) and finally Rainbow. Each of these classes will have their separate Predict(), Forward(), Backward() and ResetParameter() functions. All these files would be created within a subfolder in the RL codebase.
These classes would be accessed as follows:



```
// Set up DQN network.
VanillaDQN<> network();

// Set up DQN agent.
QLearning<CartPole, decltype(network), AdamUpdate, decltype(policy)>
    agent(std::move(config), std::move(network), std::move(policy),
          std::move(replayMethod));
```

The constructor of these classes would contain a default Feed Forward Network. Although a custom model could be made using:



```
FFN<MeanSquaredError<>, GaussianInitialization> model(MeanSquaredError<>(),
    GaussianInitialization(0, 0.001));
model.Add<Linear<>>(4, 128);
model.Add<ReLULayer<>>();
model.Add<Linear<>>(128, 128);
model.Add<ReLULayer<>>();
model.Add<Linear<>>(128, 2);

// Set up DQN network.
VanillaDQN<decltype(model)> network(std::move(model));

// Set up DQN agent.
QLearning<CartPole, decltype(network), AdamUpdate, decltype(policy)>
    agent(std::move(config), std::move(network), std::move(policy),
          std::move(replayMethod));
```

The Q Learning class will remain pretty much the same.

I suggest this change because:

1. With this structure, we will not be needing to write the Episode() and Step() functions for all our other extensions. It would remain intact in the q_learning class and all the networks will be able to use them.
2. We would not need to write separate conditions for DoubleDQN in each class, as it would be reused off of the QLearning's Step() function.

Thus the tree would look like:

```
├── q_networks
    ├── vanilla_dqn.hpp
    ├── dueling_dqn.hpp
    ├── noisy_dqn.hpp
    ├── rainbow.hpp
    └── Categorical/Distributional.hpp (optional)
```

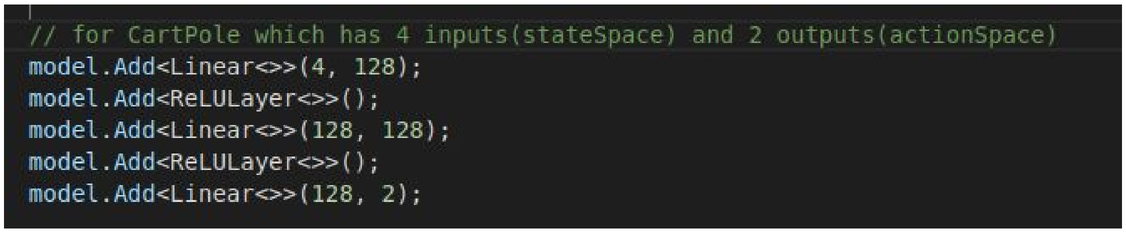
I have already implemented a ***pytorch version*** *of the proposed structure here.*

Plan of Action

1) Changes required by the different extensions for the implementation of Rainbow are:
   a) DuelingDQN:
      i) Change in the network architecture
   b) NoisyDQN:
      i) Requires the implementation of a noisy linear layer
      ii) Changes in the network.
   c) CategoricalDQN:
      i) Change in network architecture
      ii) Would require changes in loss function, so a separate loss function needs to be made
   d) N-Step DQN:
      i) Requires changes in the replay buffer.
      ii) Change in Update and loss calculation

2) Since Dueling, Noisy and Categorical DQNs are just changes to the network structure, separate classes could be created for each of them. **Prioritized Experience Replay** and **DoubleDQN** are already implemented, so no need to do anything for them.

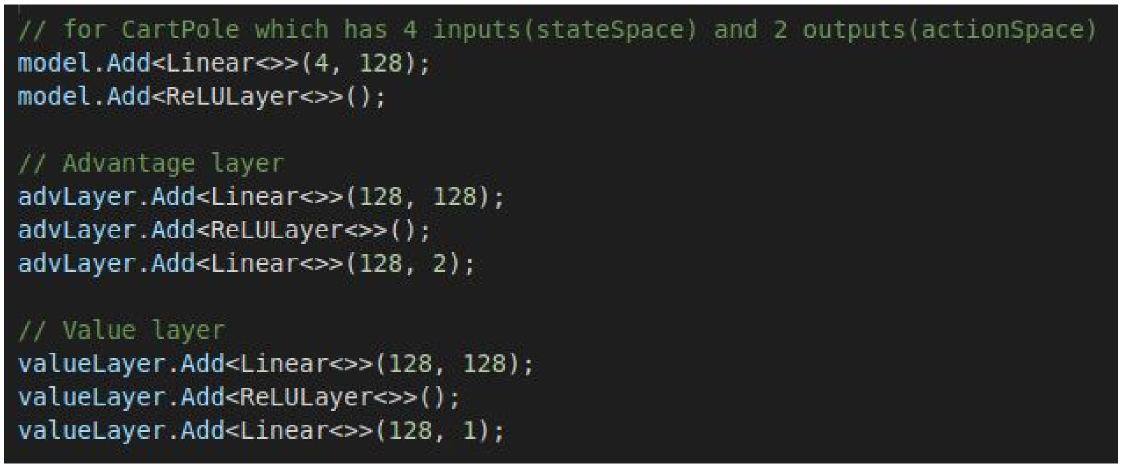
   a) VanillaDQN:
      i) This will be able to create a simple DQN as already we have been using.



```
// for CartPole which has 4 inputs(stateSpace) and 2 outputs(actionSpace)
model.Add<Linear<>>(4, 128);
model.Add<ReLULayer<>>();
model.Add<Linear<>>(128, 128);
model.Add<ReLULayer<>>();
model.Add<Linear<>>(128, 2);
```

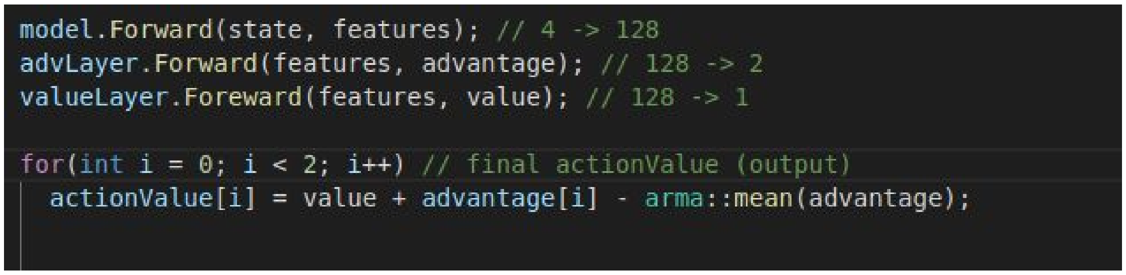
b) DuelingDQN:

i) This will create a network with the advantage and value streams added.



```
// for CartPole which has 4 inputs(stateSpace) and 2 outputs(actionSpace)
model.Add<Linear<>>(4, 128);
model.Add<ReLULayer<>>();

// Advantage layer
advLayer.Add<Linear<>>(128, 128);
advLayer.Add<ReLULayer<>>();
advLayer.Add<Linear<>>(128, 2);

// Value layer
valueLayer.Add<Linear<>>(128, 128);
valueLayer.Add<ReLULayer<>>();
valueLayer.Add<Linear<>>(128, 1);
```

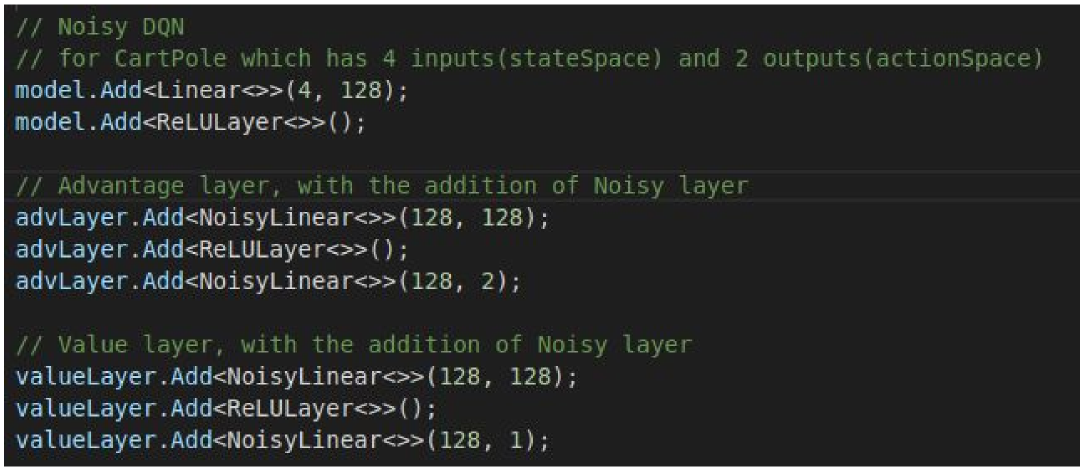
And the forward propagation:



```
model.Forward(state, features); // 4 -> 128
advLayer.Forward(features, advantage); // 128 -> 2
valueLayer.Foreward(features, value); // 128 -> 1

for(int i = 0; i < 2; i++) // final actionValue (output)
  actionValue[i] = value + advantage[i] - arma::mean(advantage);
```

c) NoisyDQN:

i) This will create a network with the noisy layers added. In the paper, NoisyNet is used as a component of the Dueling Network Architecture, which includes Double-DQN and Prioritized Experience Replay. So by default, the network will be constructed with the support for Dueling Network. This means that the Noisy Layers are added after the feature layer, i.e. separate noisy layers for Advantage and Value streams.

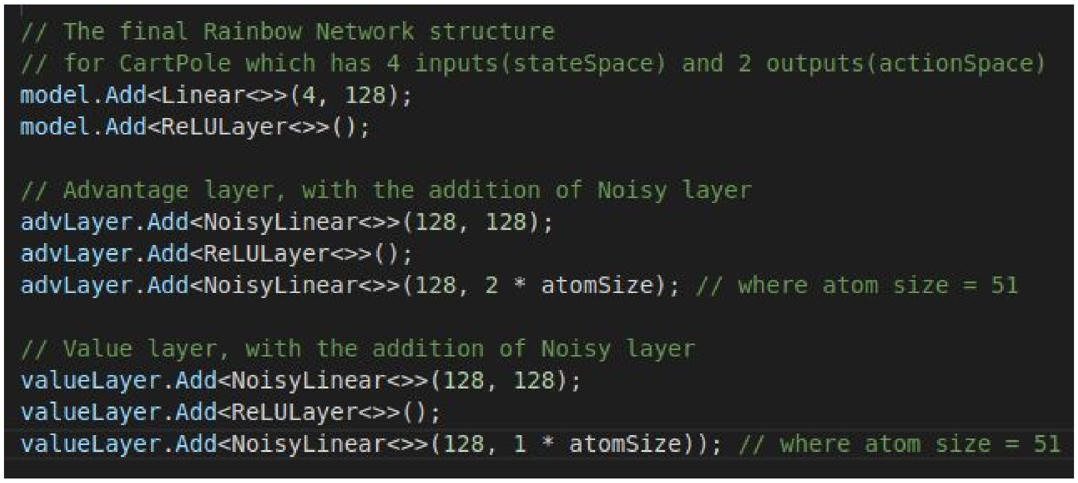


```
// Noisy DQN
// for CartPole which has 4 inputs(stateSpace) and 2 outputs(actionSpace)
model.Add<Linear<>>(4, 128);
model.Add<ReLULayer<>>();

// Advantage layer, with the addition of Noisy layer
advLayer.Add<NoisyLinear<>>(128, 128);
advLayer.Add<ReLULayer<>>();
advLayer.Add<NoisyLinear<>>(128, 2);

// Value layer, with the addition of Noisy layer
valueLayer.Add<NoisyLinear<>>(128, 128);
valueLayer.Add<ReLULayer<>>();
valueLayer.Add<NoisyLinear<>>(128, 1);
```

ii) One thing to note is that NoisyNet is an alternative to the epsilon-greedy exploration policy method. So for Noisy Nets, we need to remove the exploration policy, and directly select the action with max value. This could be achieved by a small if (config.NoisyNet == true) check before the policy sample statement.

d) Categorical/Distributional:

i) This network type would incorporate Categorical with Dueling and Noisy structures, to finally create RAINBOW.



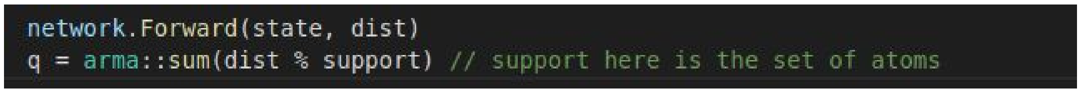
```
// The final Rainbow Network structure
// for CartPole which has 4 inputs(stateSpace) and 2 outputs(actionSpace)
model.Add<Linear<>>(4, 128);
model.Add<ReLULayer<>>();

// Advantage layer, with the addition of Noisy layer
advLayer.Add<NoisyLinear<>>(128, 128);
advLayer.Add<ReLULayer<>>();
advLayer.Add<NoisyLinear<>>(128, 2 * atomSize); // where atom size = 51

// Value layer, with the addition of Noisy layer
valueLayer.Add<NoisyLinear<>>(128, 128);
valueLayer.Add<ReLULayer<>>();
valueLayer.Add<NoisyLinear<>>(128, 1 * atomSize)); // where atom size = 51
```

ii) To estimate q-values, we use inner product of each action's softmax distribution and support which is the set of atoms:

$$\{z_i = V_{min} + i\Delta z : 0 \le i < N\},\ \Delta z = \frac{V_{max} - V_{min}}{N-1}.$$

$$Q(s_t, a_t) = \sum_i z_i p_i(s_t, a_t),\ \text{where } p_i \text{ is the probability of } z_i \text{(the output of softmax).}$$

This could be provided with:



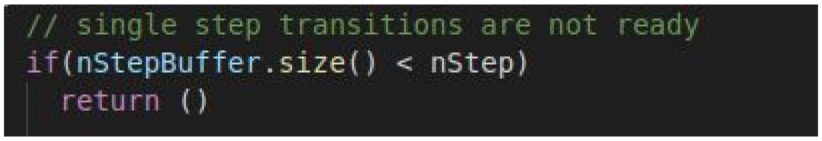
```
network.Forward(state, dist)
q = arma::sum(dist % support) // support here is the set of atoms
```

iii) A second type of class (only Categorical without Dueling and Noisy) could also be created along with the full rainbow implementation. We could discuss later whether to implement it.

3) **N-Step learning:** following are the exhaustive list of changes required:

a) In the Replay classes, we would need to create an nStepBuffer, and store all transitions in that buffer.
Then, we would need to prevent storing any transition in the original buffer, unless nStepBuffer is full.

Here, the nStep denotes the N-step learning that we require to perform, i.e. the number of steps that we look ahead.



```
// single step transitions are not ready
if(nStepBuffer.size() < nStep)
  return ()
```

b) We would also need to use two buffers here: memory and memory_n for 1-step transitions and n-step transitions respectively. It would guarantee that any paired 1-step and n-step transitions have the same indices.
   Note that these statements would be implemented in the loss functions and update functions of the respective classes(vanilla, dueling, noisy and rainbow). So we can choose to implement Nstep Learning for only RAINBOW class, or all the individual classes.
c) Also, we would need to combine 1-step loss and n-step loss so as to control high-variance / high-bias trade-off.

This would complete the implementation of Rainbow. This kind of a structure, therefore, essentially allows for the user to have all the variations of Rainbow individually as well as a complete Rainbow implementation; with **maximum code reuse**. The option to **individually select** which exact extension to include, could prove helpful in benchmarking variations of RAINBOW with different environments. Again, we could discuss any improvements to the structure.

Proposed changes in the existing codebase

Along with the changes discussed above, the following additional changes need to be made:

- **Changes in the file *methods/reinforcement_learning/replay/prioritized_replay.hpp***
  - To incorporate N-step learning for Prioritized Experience Replay. Addition of a Store() method, which would return a boolean in order to inform if a N-step transition has been generated.
- **Addition of NoisyLinear**:
  - A Noisy Linear Layer implementation with its separate Forward(), ResetNoise() and Backward() methods.
- **Addition of tests in *tests/q_learning_tests.cpp:***
  - For testing the newly implemented classes.

Testing

Describes how I will test my project.

In the paper, it is mentioned that Rainbow (the combination of six extensions of DQN) have state-of-the-art performance on the Atari 2600 benchmark. So, maybe we could find some way to train and test it using a gym TCP-API (link) project, which will allow us to communicate with the gym environment via an IP address. But this would only allow us to train and test the agent on our local systems. For the test suitcase, we need a faster and easier approach.

So we could reuse already implemented environments in mlpack for testing. Currently for the testing of DQN, we set a threshold of 1000 episodes, and train it on different environments with average Reward threshold as follows:

Environment Name	**Reward Threshold[R] (test pass if avg_reward > R)**
DQN for CartPole	35
DQN with PER for CartPole	35
DDQN for CartPole	40
DQN for Acrobot	-380.0
DQN for Mountain Car	-370.0
DQN for DoublePoleCart	> 280 for any one episode

Since, all DQN extensions deal with Discrete Action space, the following environments can be reused for testing:

- ➔ CartPole
- ➔ DoublePoleCart
- ➔ MountainCar
- ➔ Acrobot

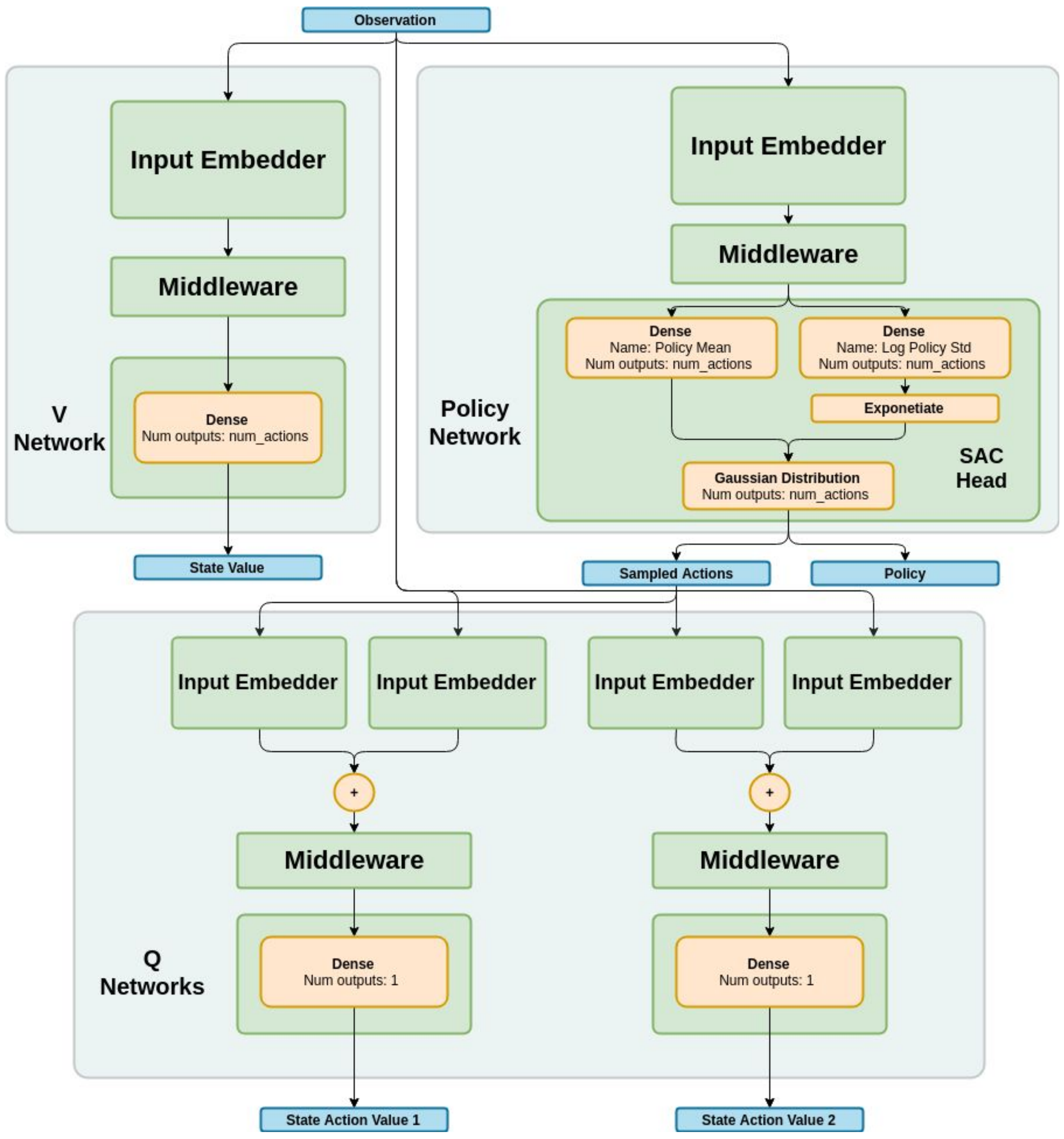
We would just need to add higher thresholds for Dueling, Noisy, Distributional and N-Step. And as for the exact Reward threshold, I would individually check each of the different extensions, on this implementation, and accordingly set reward thresholds.
I could also add LunarLander with discrete action space, as a new environment for testing, after discussing with the mentors

Part 2: Full Implementation of Soft Actor-Critic

Abstract

The architecture of the Soft Actor-Critic model consists of the following networks:

1. Two identical Q learning networks -> Q1Learning and Q2Learning
2. Two identical Q target networks -> Q1Target and Q2Target
3. A policy network (may be Deterministic or Gaussian)



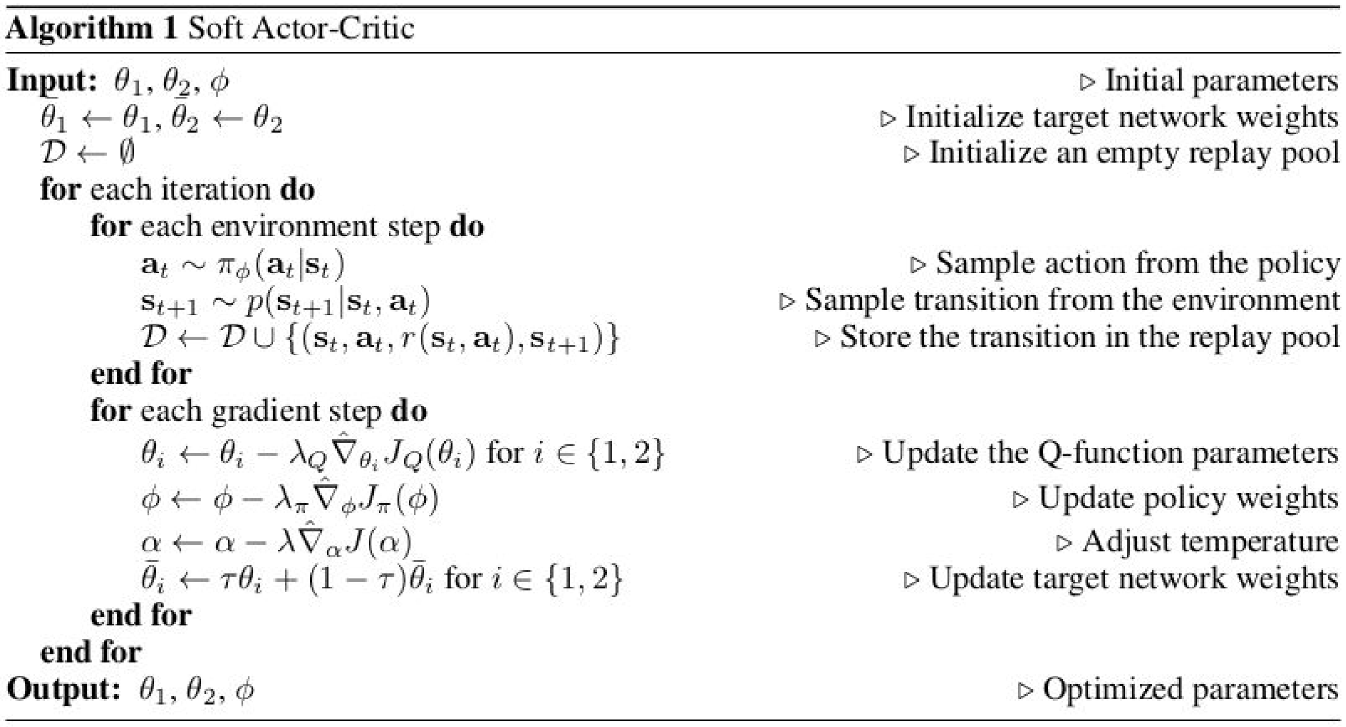
The V network which represents Value network, was present in the previous version of Soft-Actor-Critic Paper, and has now been deprecated.

The API of the algorithm will remain the same as that for QLearning.

```cpp
// define agent, pass the environment, network,
// training configurations, replayMethod(default: random)
SoftActorCritic<decltype(network)>
             agent(ContinuousMountainCar,
                  std::move(q1Network),
                  std::move(q2Network),
                  std::move(policyNetwork),
                  std::move(config),
                  std::move(replayMethod));
for(int episodes = 0; episodes < 1000; episodes++)
{
    double episodeReturn = agent.Episode();
    averageReturn(episodeReturn);
    std::cout << episodes << ' ' << episodeReturn << '\n';
}
```

There would be a default architecture of q1, q2 and policy networks present in the SoftActorCritic class, so passing them as parameters would be optional.

Plan of Action



Algorithm 1 Soft Actor-Critic

```
Input: $\theta_1, \theta_2, \phi$                                        ▷ Initial parameters
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$                 ▷ Initialize target network weights
  $\mathcal{D} \leftarrow \emptyset$                                       ▷ Initialize an empty replay pool
  for each iteration do
     for each environment step do
        $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$                        ▷ Sample action from the policy
        $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$             ▷ Sample transition from the environment
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$    ▷ Store the transition in the replay pool
     end for
     for each gradient step do
        $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$   ▷ Update the Q-function parameters
        $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$                 ▷ Update policy weights
        $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$                    ▷ Adjust temperature
        $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$     ▷ Update target network weights
     end for
  end for
Output: $\theta_1, \theta_2, \phi$                                       ▷ Optimized parameters
```

The constructor of the class would initialize/store the following values.

```
//! Locally-stored hyper-parameters.
TrainingConfig config;

//! Locally-stored learning Q1 and Q2 network.
QNetworkType learningQ1Network;
QNetworkType learningQ2Network;

//! Locally-stored target Q1 and Q2 network.
QNetworkType targetQ1Network;
QNetworkType targetQ2Network;

//! Locally-stored policy network.
PolicyNetworkType policyNetwork;

//! Locally-stored updater.
UpdaterType updater;

//! Locally-stored experience method.
ReplayType replayMethod;

//! Locally-stored reinforcement learning task.
EnvironmentType environment;

//! Locally-stored flag indicating training mode or test mode.
bool deterministic;
```

The usual **Step()** and **Episode()** functions need to be made:

```
/**
  * Execute a step in an episode.
  * @return Reward for the step.
  */
double Step();

/**
  * Execute an episode.
  * @return Return of the episode.
  */
double Episode();
```

In the **Episode()** function, the following need to be added:

- ➔ Get an initial state from the environment
- ➔ Create variable for the counting of total steps and return
- ➔ Till the environment ends, do:
    - ◆ totalReturn += **Steps()**
    - ◆ If Deterministic, end the loop
    - ◆ Execute **Update()**

In the **Step()** function, the following need to be added:

- ➔ Forward pass the state into the policy network to get the action
- ➔ Sample reward and next state from environment by passing in the state and action
- ➔ Store the transition in buffer

Apart from these, following functions should be present:

- ➔ ***Update()***
  - ◆ This function will contain code for the update of all networks.
  - ◆ In terms of pseudo code:
    - Sample a batch from the replay buffer.
    - Compute targets for Q_learning functions (Q1 and Q2). Alpha here denotes the entropy term. r stands for reward and (1-d) deals with the terminal state.

$$y(r, s', d) = r + \gamma(1-d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

This can be achieved with the following lines:

```
// takes input nextStateBatch and gives out nextStateAction and nextStateLogPi
policyNetwork.Predict(nextStateBatch, nextStateAction, nextStateLogPi);

// takes input nextStateBatch and nextStateAction, gives out Q1 Q2 targets
targetQ1Network.Predict(nextStateBatch, nextStateAction, q1NextTarget,
q2NextTarget);

// takes the minimum of two Q targets, and
minQNextTarget = arma::min(q1NextTarget, q2NextTarget) -
config.entropy()*nextStateLogPi;

// calculates the target Q values
nextQValue = rewardBatch + maskBatch * config.gamma() * (minQNextTarget);
```

    - Update Q_learning network by one step of gradient descent

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2$$

Could be done by:

```
// Update the Q Learning Networks

arma::mat Q1, Q2;
learningQ1Network.Forward(stateBatch, actionBatch, Q1);
learningQ2Network.Forward(stateBatch, actionBatch, Q2);
// backward Propagate the loss:
arma::mat gradients1, gradients2;
learningQ1Network.Backward(stateBatch, nextQValue, gradients1);
learningQ2Network.Backward(stateBatch, nextQValue, gradients2);
// Update network:
updatePolicy->Update(learningQ1Network.Parameters(), config.StepSize(), gradients1);
```

```
updatePolicy->Update(learningQ2Network.Parameters(), config.StepSize(), gradients2);
```

- Update policy network by one step of Gradient Ascent, here, min denotes the minimum of the two Q_learning networks (Q1 and Q2)

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta \left(\tilde{a}_\theta(s) | s \right) \right)$$

```
arma::mat pi, logPi;
policyNetwork.Forward(state_batch, pi, logPi);
minQPi = arma::min(Q1, Q2);
// policy loss calculated from minQPi
policyLoss = arma::mean((config.entropy() * logPi) - minQPi);
```

The policy loss would then be backward propagated.

- Tune the entropy value by calling **TuneEntropy()**

- Finally, SoftUpdate the **learning Q_network** parameters into their **target Q_network** parameters. (for both Q1 and Q2)

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho) \phi_i$$

```
SoftUpdate(targetQ1Network, learningQ1Network, config.rho());
SoftUpdate(targetQ2Network, learningQ2Network, config.rho());
```

◆ **Parameters:** None
◆ **Return:** void

➔ ***SoftUpdate***(target, learning, rho):
◆ This function would be used for "softly" copying the learning Q_network parameters to the target Q_network parameters. Here the value of rho is kept 0.005 according to paper.

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho) \phi_i$$

◆ **Input Parameters:** The target, learning networks; rho is the Interpolation factor in polyak averaging
◆ **Return:** void

➔ ***TuneEntropy()***
◆ Would update the entropy term, if config.autoUpdateEntropy() is enabled
◆ **Parameters:** None
◆ **Return:** void

➔ Other functions like Forward and Backward passes for the Q and policy networks would be added as per necessary.
➔ **Note**: We **might** require to divide the code into classes, eg: adding separate classes for Policy and Q network(which would contain the forward and backward pass functions). This would depend on how long the sac_impl.hpp goes, and its complexity. I'll keep in mind to discuss this with the mentor, after implementing.

Proposed changes in the existing codebase

➔ **Addition of the file *methods/reinforcement_learning/sac.hpp***
  - This would contain the definition for the SoftActorCritic class

➔ **Addition of the file *methods/reinforcement_learning/sac_impl.hpp***
  - This would contain the implementation for the SoftActorCritic class

➔ **Addition of tests in *tests/sac_tests.cpp:***
  - For testing the SAC environment.

➔ **Addition of new environments in *methods/reinforcement_learning/environment:***
  - New continuous action space environments could be added for the testing of Soft Actor-Critic

Testing

Describes how I will test my project.

Currently, Mlpack has the following environments with continuous action space:

➔ MountainCar Continuous
➔ DoublePoleCart Continuous
➔ Pendulum
➔ LunarLanderContinuous (needs to be merged #1912)

These could be used for testing out Soft Actor-Critic. I will get the appropriate thresholds for rewards for each environment, after running the same environment in the original paper's implementation(link), and comparing results.

I could also implement a more challenging continuous action-space environment. In my opinion, BipedalWalker and BipedalWalkerHardcore would be good options. But since these environments are computationally more intensive and challenging, we would require them to run for less number of episodes and keep very low test-passing reward barriers, if we decide to use them. After discussing with the mentor, I would like to add these to the environment list, if required.

Timeline with Proposed Deliverables

Community Bonding Period — May 4 - June 1

- ➔ The Project will be discussed in further detail with the mentor.
- ➔ I will get more acquainted with the codebase, especially FFN, along with that I plan on getting to know the armadillo library as it will be used extensively.
- ➔ I'll try to fix some issues (if any) related to reinforcement learning, while continuing with #1912 (PR for PPO).
- ➔ I will also try and experiment more with the currently proposed idea, with inputs from the other developers. The optional parts, like Distributional DQN, would especially be discussed with the mentor, along with any flaws in the overall architecture.

Week 1 — June 1 - 7

- ➔ This period would be spent on writing a skeleton layout of the Dueling and Noisy extensions to implement.
- ➔ I would also spend more time on reading related research papers to get a more robust view of the problem, along with articles from Spinning Up (OpenAI).

Weeks 2 & 3 — June 8 - 21

- ➔ Once the skeleton is ready, I'll finish the **Dueling** and **Noisy** extensions, this would include writing Forward and Backward functions for the networks.
- ➔ Noisy Layer would be added as a new layer type to methods/ann/layer

Week 4 & 5 — June 22 - July 5

- ➔ I'll get the Dueling and Noisy extensions merged as a separate PR by writing tests for both of them, and getting issues fixed.
- ➔ I'll then proceed with adding support for **N-Step Learning**, by adding on to the Replay classes, and creating its separate loss functions.
- ➔ I'll also try to complete the tests for N-step in this time period.
- ➔ Detailed documentation for Dueling and Noisy would be completed.

Week 6 & 7 — July 6 - 19

- ➔ These two weeks will be spent on the **Distributional/Categorical** part of the extension, thus completing the Rainbow implementation.
- ➔ By now, I'll make sure to complete documentation for Distributional and N-Step as well, describing each function in detail.
- ➔ Then, I'll write tests, while getting the basic layout of SAC ready. I'll try to get the PR for the rainbow merged by the end.

Week 8 & 9 — July 20 - August 2

- ➔ These two weeks will be spent extensively on **SAC**. I'll try to complete the full implementation including the main update function.
- ➔ I'll also test the code along the way to check if it converges for relatively easy environments like Pendulum

Week 10 & 11 — August 3 - 16

- ➔ I'll complete **writing tests for the SAC** implementation.
- ➔ If required after discussing with the mentor, I would create and test SAC on new environments.
- ➔ I'll provide detailed documentation for Soft Actor-Critic and also a tutorial, if time remains.

Ultimate Week — August 17 - 24

- ➔ I would wrap up my implementations by completing all **pending work** (if any) from before, including implementation, tests and documentation.
- ➔ I would try to get all **PRs merged**, after thorough review from mentors and fellow developers.
- ➔ Finally, I will discuss with the mentor regarding future improvements.

Personal Details

I am Nishant Kumar, pursuing graduation in electronics at Indian Institute of Technology (BHU), Varanasi.

I started primarily with coding 2 years back and have grown to like open source, because of its vibrant and engaging community. I actively use open source softwares and like to contribute back when I can. I believe in solving problems by using the latest technology available, rather than reinventing the wheel.

I have extensively used deep learning frameworks like pytorch and keras with tf for my projects, and have always wondered what goes into making such easy to use APIs. Working with Mlpack, I would like to find the answer to that!

I have been using **Ubuntu 16.04**, with **zsh** as my favoured shell. I use **Sublime Text** for small codes and **Visual Studio Code** for working on development projects.

Technical Proficiency and coding skills

1) **What languages do you know? Rate your experience level (1-5: rookie-guru) for each.**

➢ Python	: 4
➢ C++	: 4
➢ JavaScript	: 3
➢ Dart	: 2
➢ Java	: 1

2) **How long have you been coding in those languages?**

I started using Java when I had Computer Applications in my Class 10 Board examinations. Then the next year, I shifted to using C++, and have been using it since then for competitive programming, due to its incredible speed. As for python, I have been using it since the past 2 years now, and it provides a very fast prototyping platform. That is the reason why I use it for most of my Machine learning projects. As for JavaScript, I used it for the implementation of a project of mine an year ago; where I made a flappy bird game, which could run on any browser, and used neuroevolution to train it. While making the project, I made use of all OOPs concepts that I learnt earlier. And finally, I have been using Dart for about 5 months now, for making a flutter app for my Institute.

3) **Are you a contributor to other open-source projects?**

As of now, I have been developing the Institute App with a team of developers from my Institute. Apart from that, I have not had the privilege of working with an organization as such.

4) **Do you have a link to any of your work (i.e. github profile)?**

➢ Pytorch Implementation of RainbowDQN: Code
➢ JerBot - a biomimetic bipedal bot : Code
➢ A Robust Hand gesture recognition system: Link
➢ Built an agent to play the Flappy Bird using Evolutionary Strategies: Demo, Code
➢ Visual Servoing for making a LUDO solving Robot: Code
➢ Institute App : Code
➢ Other Projects can be found here: https://github.com/nishantkr18

5) **What areas of machine learning are you familiar with?**

I was intrigued by the idea of machine learning when I first saw a video demonstration of an agent, trying to find its path through a maze, without any specific set of rules programmed. The curiosity of its working led me to the world of machine learning, where I explored all my interests:

➢ I constructed a neural network from scratch to play the flappy bird game. I used the (**Evolutionary Strategy** i.e. selection of the fittest) technique to obtain the best performing network for the game.

- ➢ To dive deeper, I started working with **CNNs**, and building projects while also participating and winning some supervised learning competitions mentioned in my resume.
- ➢ When I came across **Variational Autoencoders** and **GANs**, I found it amazing and hence learnt about them, by hacking through their codes and tweaking parameters.
- ➢ Apart from that, I have used **XGBoost** for making a Music Recommendation System, for yet another hackathon.
- ➢ After having explored the domain of supervised learning, I made my mind to dive deep into the topic I began my journey with, i.e. **Reinforcement Learning**.
- ➢ I have in depth knowledge about the implementation of some model-free off-policy algorithms like **DQNs**, **Soft A-C, A2C**, **REINFORCE**, along with **PPO** and **TRPO**(on-policy algos).

6) Have you taken any coursework relevant to machine learning?

- ➢ Having made a hand gesture recognition system from scratch, I then went for a thorough revision of my understanding by enrolling in 'Stanford's **CS231N** CNNs for Computer Vision' course.
- ➢ To better understand the concepts of RL, I started following the "**UCL**'s course on RL by **David Silver**."
- ➢ I have also taken Stanford's Lectures on Deep Learning **CS230**.
- ➢ As for other parts of my work on machine learning, I rely on **Medium** blogs, **reddit**, **StackOverflow** discussions and **Research Papers**. I learnt a lot by interacting with the community out there.

Other open-ended questions

1) What are your long-term plans, if you have figured those out yet? Where do you hope to see yourself in 10 years?

- ➢ I plan on working at the junction of research-oriented machine learning and development. Because I believe that machine learning problems can only be properly tested when applied to real-world problems. So, I hope to see myself in 10 years, working with an eminent research facility or organization, implementing frontier research ideas for real-life use-cases. I find the field of Artificial Intelligence quite fascinating, so I plan on working for its application on cyber-physical systems. If I am lucky enough, I would love to work in places like UC Berkeley, CMU, ETH Zurich if I get a chance.
- ➢ I would also like to mention that I had long been planning on working on open-source projects, and now that I am getting familiar with the workflow, I have no plans on quitting. So, I would like to contribute to open-source projects whenever I get time from my work, for fun, learning, and to have a sense of belonging, as I have been experiencing for quite some days now! And the experience I have had using open source projects, I think OpenSource development will occupy a long portion of my life.

2) **Describe the most interesting application of machine learning you can think of, and then describe how you might implement it.**
   - ➢ There are countless applications of machine learning almost everywhere around us, which have made our lives easier. But all these are uses of supervised and clustering algorithms, along with a few applications of Reinforcement Learning here and there.
   - ➢ But I believe that the most revolutionary invention(or discovery) is yet to come. It can be referred to as General Intelligence, which could be thought of as an agent capable of making complex decisions on its own and having the ability to reason. The benefit of such an agent will mark the end of barriers in communication between humans and machines. Instead of just using machines for computation, we could use them to help us; let's say intuitively solve an equation. Maybe time travel, or nuclear fusion in a controlled manner, are achievable; yet waiting to be discovered. With the help of General Intelligence, we could fasten the process.
   - ➢ These are dreams which are not that far away. We already have image segmentation and visual question answering methods, which could bring out representations from image data. The field of natural language processing has shown how to make meaning out of text. Generative models(like GANs) can be considered as dreaming in some form. So, we have achieved most human level tasks separately, we need a mechanism to bring it all together.
   - ➢ I believe this is achievable with the combination of reinforcement learning and evolutionary strategies. We could compare different agents using various RL techniques and evolve them through neuro-evolution. Doing this would make them learn from their mistakes and would be able to bring out the best of them from their generation.
   - ➢ With innovations at this pace, we can expect some form of General Intelligence in less than 30 years! And as an enthusiast, I feel obliged to keep myself updated with these technologies and contribute where-ever I can.

3) **Both algorithm implementation and API design are important parts of mlpack. Which is more difficult? Which is more important? Why?**
   - ➢ Although both are important parts of mlpack, I personally find API design to be more challenging and important at the same time. This is because implementation of an algorithm could be done straight forward, but for a proper API design, the developer has to consider a lot of things, including the ease of usage and understandability both for fellow/future developers and for end users. Also, the codebase needs to be structured in such a way that it is easy for adding new features in the future.

Communication

I’m flexible with my schedule and have inculcated the habit of working at night, so time zone difference shouldn’t be an issue. I’m comfortable with any of the communication mediums I mentioned above.

I can work full-time on weekdays and am usually available between **11 AM IST to 2 AM IST**. On weekends, I would love to spend time communicating with the team to learn from them, while working on whatever issues occur at that time.

I’ll responsibly keep my mentor updated in case of any emergency that occurs with suitable details.

Post GSoC

If there are things left unimplemented, I’ll try to complete them post GSoC and will keep contributing to Mlpack, by adding other algorithms , which are yet to be implemented like: **ACKTR**, **A2C**, **TD3**, **DDPG, TRPO**. I would also like to contribute more to documentation of the code.

Contributions

Merged Pull Requests:

- DoubleDQN doesn't utilize DoubleQLearning - BugFix relating to testing of DoubleDQN implementation.

Opened Pull Requests:

- Addition of q_network - Adds the proposed implementation of QNetwork as a separate class.
- Proximal Policy Optimization - Working on getting last year’s GSoC project, merged with the master branch.