Udacity Project Navigation Report

Contents

Introduction

In this document I will cover the explanation and description of my solution to The Challenge project Navigation for the Deep Reinforcement Learning Nanodegree of Udacity. My solution covers 8 different algorithms as I wanted to explore all possible improvements to the Vanilla deep RL DQN algorithm. The skeleton of this solution is based on the coding Exercise Deep Q-Networks (lesson 2) of this program, while I also use other resources as books, or public information available that I will detail on the references.

The application solves the environment with the following 7 implementations

Mode 1 → Plain Deep DQN vanilla. (Greedy algo for action selection)

Mode 2 → Duelling DQN with priority buffer replay (Greedy algo for action selection)

Mode 3 → Duelling DQN without priority buffer replay (only replay buffer) (Greedy algo for action selection)

Mode 4 → categorical DQN, without priority buffer replay (only replay buffer) (Greedy algo for action selection)

Mode 5→ Duelling DQN, with priority buffer replay and Noisy Layer for exploration. (NO greedy algo for action selection)

Mode 6 → DQN n-steps (only replay buffer) (Greedy algo for action selection)

Mode 7 → Rainbow DQN (Duelling DQN + n-Steps + Categorical + Noisy Layer for Exploration + priority Buffer Replay)

Mode 8 → Duelling DQN, without priority buffer replay and Noisy Layer for exploration. (NO greedy algo for action selection)

Installation

My solution works as an application which run in a windows command line window (I did not try in Linux, but I suspect that with minimum changes it will work). To setup the environment, I simply setup the DRLND GitHub repository in an Conda environment as is demanded in the project instructions and then a windows(64-bit) unity environment. I use Pycharm Professional for code Development:

Just copy and paste from Udacity

Step 1: Clone the DRLND Repository

If you haven't already, please follow the **instructions in the DRLND GitHub repository** to set up your Python environment. These instructions can be found in `README.md` at the root of the repository. By following these instructions, you will

install PyTorch, the ML-Agents toolkit, and a few more Python packages required to complete the project.
(*For Windows users*) The ML-Agents toolkit supports Windows 10. While it might be possible to run the ML-Agents toolkit using other versions of Windows, it has not been tested on other versions. Furthermore, the ML-Agents toolkit has not been tested on a Windows VM such as Bootcamp or Parallels.

Step 2: Download the Unity Environment

For this project, you will **not** need to install Unity - this is because we have already built the environment for you, and you can download it from one of the links below. You need only select the environment that matches your operating system:

- Linux: **click here**
- Mac OSX: **click here**
- Windows (32-bit): **click here**
- Windows (64-bit): **click here**

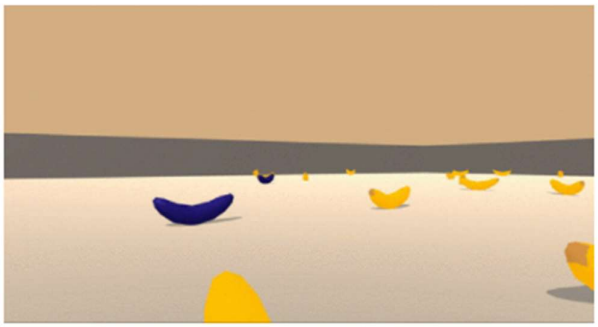
Then, place the file in the `p1_navigation/` folder in the DRLND GitHub repository, and unzip (or decompress) the file.
(*For Windows users*) Check out **this link** if you need help with determining if your computer is running a 32-bit version or 64-bit version of the Windows operating system.
(*For AWS*) If you'd like to train the agent on AWS (and have not **enabled a virtual screen**), then please use **this link** to obtain the "headless" version of the environment. You will **not** be able to watch the agent without enabling a virtual screen, but you will be able to train the agent. (*To watch the agent, you should follow the instructions to **enable a virtual screen**, and then download the environment for the **Linux** operating system above.*)

Problem Description

Just copy and paste from Udacity

The Environment

For this project, you will train an agent to navigate (and collect bananas!) in a large, square world.



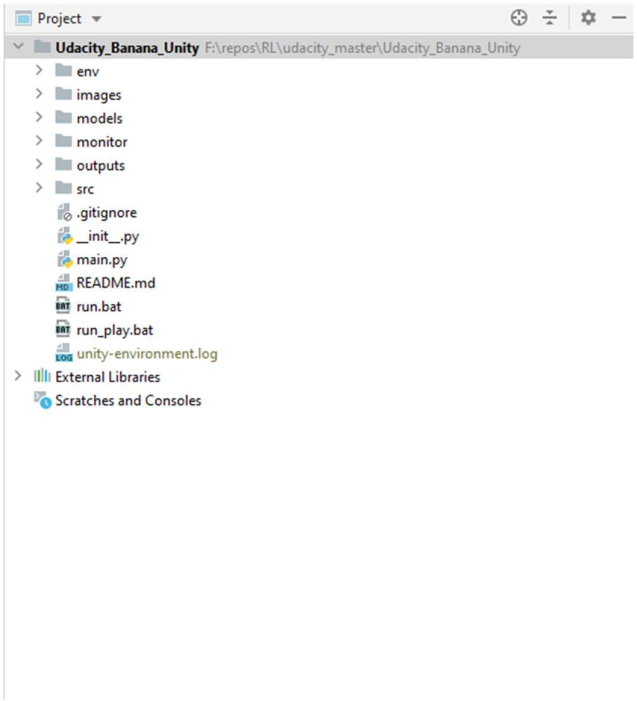
A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent must learn how to best select actions. Four discrete actions are available, corresponding to:

- `0` - move forward.
- `1` - move backward.
- `2` - turn left.
- `3` - turn right.

The task is episodic, and to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.

Directory Structure



env: the unity environment

Images: Folder where I save plots during training and final plots

Models: Folder where I save the operational models

Monitor: Folder where I save a csv file where I am collecting information of experiments

Outputs: Folder where I save a pickle file containing a dictionary which contains all data to build the final plots and this report

src: contains python scripts with classes to support the application

In the root I have the python scripts, and some cmd scripts to help to run in a loop the environment using different algorithms either during training phase or during play phase

Main.py: Contains the logic which govern the 5 main operations modes

In src folder

Agents.py: contains classes which wrap the operation of the Banana env working with different algorithms and buffers. Additionally, some functions to operate the env in training or play mode

Networks: contains different implementation of Neural Network architectures use by the agents to solve the environment

Buffers.py: contains different buffer implementations. Senment_tree.py and sumtree.py contains classes only use by the different buffers.

Hyper.py: contains functions and wrappers for hyper parameter tuning

Utils.py: contains helpers to monitor, plot and instantiate the agents

Banana.exe and UnityPlayer.dll are the Unity Banana environment

Run.bat and run_play.bat are two cmd scripts to help me to run all solvers in a loop in training and play mode

Operations mode:

--mode training|play|compare|compare_play|plot → Mandatory

training → Train and agent. Save a model policy if the agent get more or equals than 13

play → play an agent with a save policy and report the score

compare → train all solvers and collect information for reporting

compare_play → play all solvers and collect information for reporting

plot → generate the plot from information collected in compare modes

hp_tuning → hyper parameter tuning example

--type → Mandatory

type 1-->Vanilla DQN

type 2--> Duelling DQN PBR

type 3--> Duelling DQN no PBR

type 4--> categorical DQN

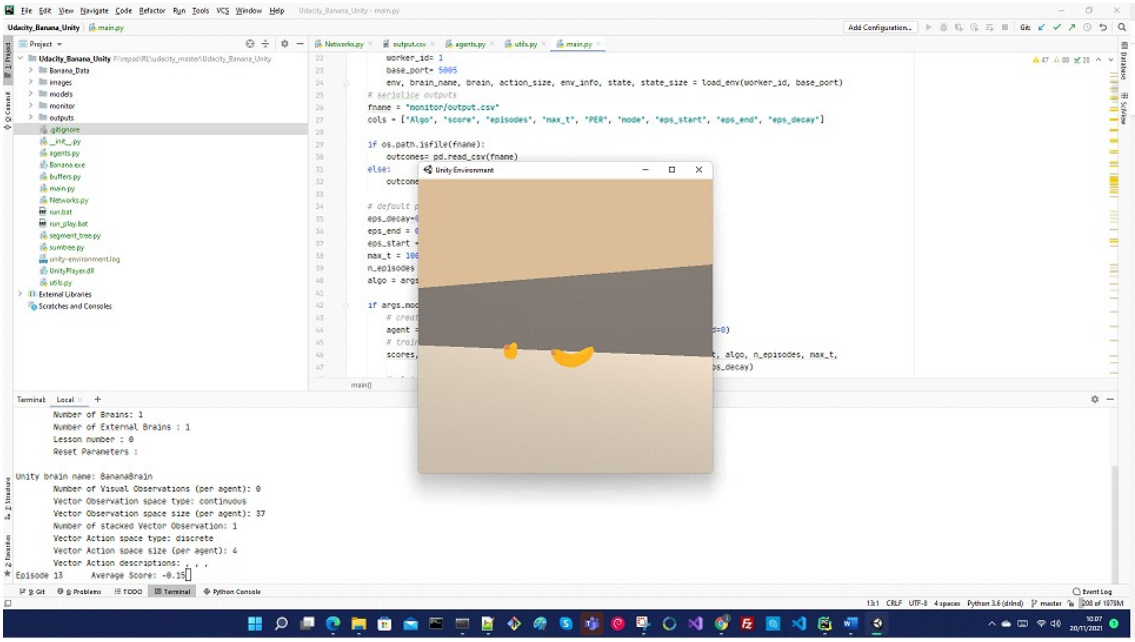
type 5--> Duelling DQN with Noisy layer and PBR

Type 6--> DQN n-steps

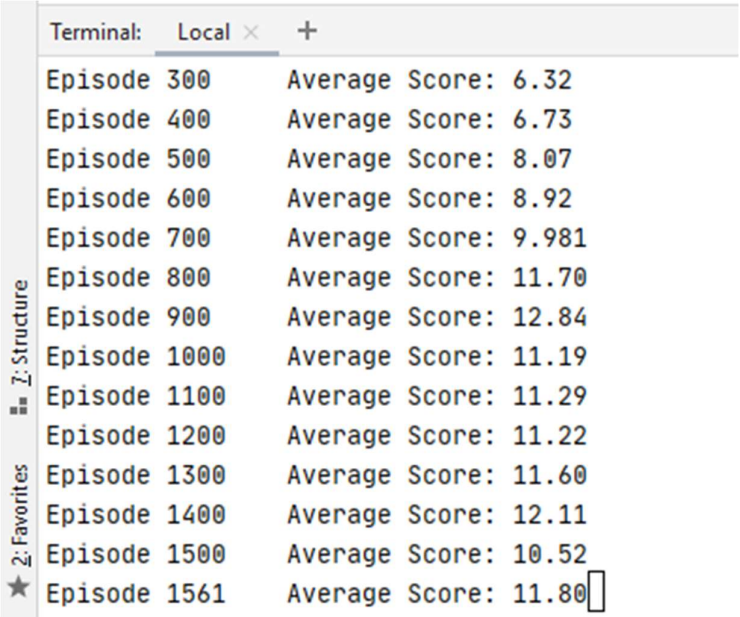
Type 7 --> Rainbow DQN

Type 8--> Duelling DQN with Noisy layer No PBR

Ex. python main.py --mode training --type 1



When we are training, I print the mean score for the 1000 steps of each episode as requested



I only save the model if the agent scores 16 during training. On the mode compare_all I save the model when the agent scores 13. Metrics of the score and conf selected for training is saved for each train or play on the output.csv (see methods dqn_runner and all_dqn_runner in src/agents.py and src/utils.py)

Output.csv Ex.

```
Algo,score,episodes,max_t,PER,mode,eps_start,eps_end,eps_decay
4,14.0,0,1000,0,play,1.0,0.01,0.995
4,16.03,1093,1000,0,training,1.0,0.01,0.995
4,15.0,0,1000,0,play,1.0,0.01,0.995
4,16.01,1565,1000,0,training,1.0,0.01,0.995
3,14.72,2000,1000,0,training,1.0,0.01,0.995
1,16.02,726,1000,0,training,1.0,0.01,0.995
1,6.0,0,1000,0,play,1.0,0.01,0.995
```

In main.py

default parameters

eps_decay=0.995 # decay multiplicator for epsilon greedy algo

eps_end = 0.01 # epsilon greedy min threshold. Always leave the possibility to do exploration

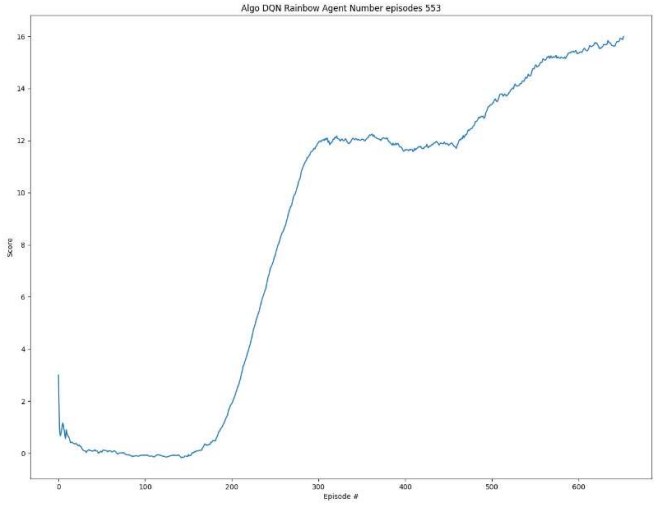
eps_start = 1.0 # epsilon greedy parameter start always in 1, first action always random

max_t = 1000 # max number of steps per episode

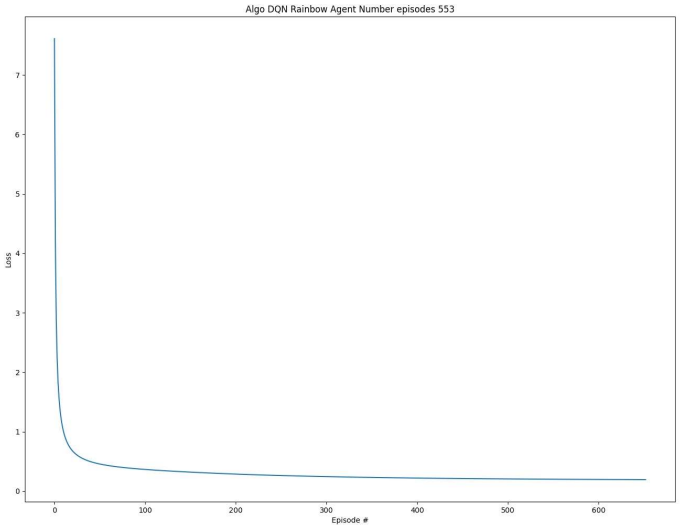
n_episodes = 2000 # max number of episodes on training phase

On training I am saving an image of loss and reward evolution up to the agent hit the score 16

algo 7 scores 16 after 663 episodes



Loss for that experiment



Mode Compare

I run all solvers 2000 episodes and collect statistics. Use run.bat

Mode Compare_plot

Run all solvers in mode play and collect statistics. Use run_play.bat

Description Algorithms

Default values

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 32        # minibatch size replay from Target Networks
GAMMA = 0.99           # discount factor
TAU = 1e-3             # for soft update of target parameters (when we apply soft update)
LR = 5e-4              # learning rate for Optimizer. Usually, Adam
UPDATE_EVERY = 4       # how often to update the network (only for soft update) for hard update fix
                        # Value of 200 and we don't apply TAU
```

Type 1 Vanilla DQN

https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

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 to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values

$$r + \gamma \max_{a'} Q(s',a').$$

The authors suggest two key ideas to address these instabilities with a novel variant of Q-learning: Replay buffer and Fixed Q-target.

Uniformly random sampling from Experience Replay Memory

Reinforcement learning agent stores the experiences consecutively in the buffer, so adjacent (s,a,r,s') transitions stored are highly likely to have correlation. To remove this, the agent samples experiences uniformly at random from the pool of stored samples ((s,a,r,s')∼U(D)). See sample method of ReplayBuffer class for more details.

Random replay buffer Code Snippet

```
state, action, reward, next_state, done = zip(*random.sample(self.buffer, batch_size))
```

Fixed Q-target

DQN uses an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target; if not, it is easily diverges because the target continuously moves. The Q-learning update at iteration i uses the following loss function:

$$Li(\theta i)=E(s,a,r,s')\sim U(D)[(r +\gamma \max_{a'} Q(s',a';\theta_i^-)-Q(s,a;\theta_i))^2]$$

in which γ is the discount factor determining the agent's horizon, θi are the parameters of the Q-network at iteration i and θ−i are the network parameters used to compute the target at iteration i. The target network parameters θ−i are only updated with the Q-network parameters (θi) every C step and are held fixed between individual updates. (Here in the application, we use a default value of 200, but is a candidate value to be optimized in a HP tuning phase)

DQN Code Snippet

```
# Predict values local DQN and target DQN
q_values = self.qnetwork_local(state)
next_q_values = self.qnetwork_target(next_state)

# transform the dimension of both to single values
q_value = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
# as target value is a matrix we use max to get the max value of first row
next_q_value = next_q_values.max(1)[0]

# calculate expected rewards. If not done consider next_q_value from target
network, if done just return reward for that final state
expected_q_value = reward + gamma * next_q_value * (1 - done)

#  loss calculation
loss = F.smooth_l1_loss(q_value, autograd.Variable(expected_q_value.data))


# record loss for plotting
self.losses.append(loss.item())

# We first apply zero_grad to the optimizer to zero out any gradients as a
reset. We then push the loss backward, and finally perform one step on the
optimizer
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

```
# Decide if we update target network. Here target_update has a default value of 200

self.update_cnt += 1
# if hard update is needed
if self.update_cnt % self.target_update == 0:
    self._target_hard_update()
```

For more stability: Gradient clipping

The authors also found it helpful to clip the error term from the update

$$r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a,;\theta_i)$$

to be between -1 and 1. Because the absolute value loss function |x| has a derivative of -1 for all negative values of x and a derivative of 1 for all positive values of x, clipping the squared error to be between -1 and 1 corresponds to using an absolute value loss function for errors outside of the (-1,1) interval. This form of error clipping further improved the stability of the algorithm.

To implement gradient clipping we use the torch.nn.SmoothL1Loss

Note: From pytorch documentation SmoothL1Loss

Creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise. It is less sensitive to outliers than torch.nn.MSELoss and in some cases prevents exploding gradients

For a batch of size N, the unreduced loss can be described as:

$$\ell(x, y) = L = \{l_1, ..., l_N\}^T$$

with

$$l_n = \begin{cases} 0.5(x_n - y_n)^2/beta, & \text{if } |x_n - y_n| < beta \\ |x_n - y_n| - 0.5 * beta, & \text{otherwise} \end{cases}$$

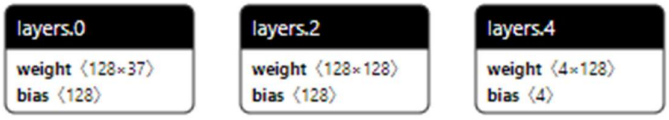
If *reduction* is not *none*, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

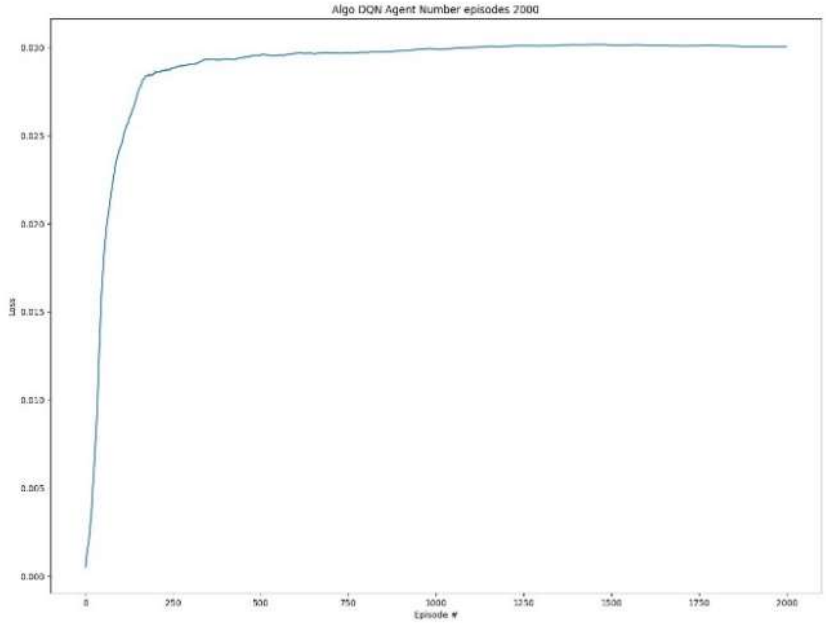
DQN Network Code Snippet

```
class DQN(nn.Module):
    def __init__(self, num_inputs, num_actions):
        super(DQN, self).__init__()

      # First layer input defined by a vector size observations space and
      # 128 nodes. Hidden layer 128 nodes and output layer size equal to
      # number of actions of the env
      # Candidate parameters to be optimized are the number of nodes on
      # the hidden layer
      # Activation function ReLU(x)=(x) + =max(0,x)

        self.layers = nn.Sequential(
            nn.Linear(num_inputs, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, num_actions)
        )

      # forward propagation
    def forward(self, x):
        return self.layers(x)
```

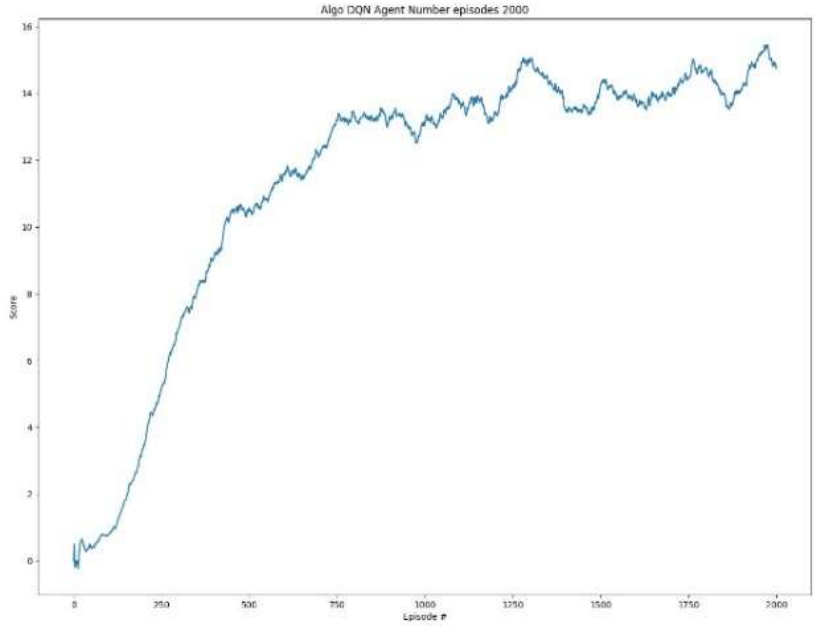
Network Architecture



loss 2000 Episodes



Reward 2000 Episodes



Type 2 Dueling Network (Double DQN)

van Hasselt et al., "Deep Reinforcement Learning with Double Q-learning." arXiv preprint arXiv:1509.06461, 2015.

https://arxiv.org/pdf/1511.06581.pdf

https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751

Double DQN

Let's take a close look at the difference between DQN and Double-DQN. The max operator in standard Q-learning and DQN uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates.

$$\theta_{t+1} = \theta_t + \alpha(Y_t^Q - Q(St,At;\theta t)) \cdot \nabla_{\theta t} Q(St,At;\theta t),$$

where α is a scalar step size and the target Y_t^Q is defined as

$$Y_t^Q = R_{t+1} + \gamma \max_a Q(S_{t+1},a;\theta t).$$

In Double Q-learning (van Hasselt 2010), two value functions are learned by assigning experiences randomly to update one of the two value functions, resulting in two sets of weights, θ and θ'. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. For a clear comparison, we can untangle the selection and evaluation in Q-learning and rewrite DQN's target as

$$Y_t^Q = R_{t+1} + \gamma\, Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1},a;\theta_t);\theta_t).$$

The Double Q-learning error can then be written as

$$Y_t^{DoubleQ} = R_{t+1} + \gamma\, Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1},a;\theta_t);\theta'_t).$$

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. In conclusion, the weights of the second network θ'_t are replaced with the weights of the target network for the evaluation of the current greedy policy. This makes just a small change in calculating the target value of DQN loss.

Double DQN Loss Calculation Code Snippet

```
# no prioritary buffer
# Double DQN take the max in between the prediction calculated by the target network and the prediction of the local network using the next_state
        Q_targets_next = self.qnetwork_target(next_states).gather(
            1, self.qnetwork_local(next_states).argmax(dim=1, keepdim=True)).detach()

        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        # loss = F.mse_loss(Q_expected, Q_targets) replaced mse loss with smooth for gradient
        # clipping
        loss = F.smooth_l1_loss(Q_expected, Q_targets)
        # record loss
        self.losses.append(loss.item())

        # Minimize the loss and backpropagate them
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

Dueling DQN.

Essentially, the proposed network architecture, which is named Dueling architecture, explicitly separates the representation of state values and (state-dependent) action advantages.

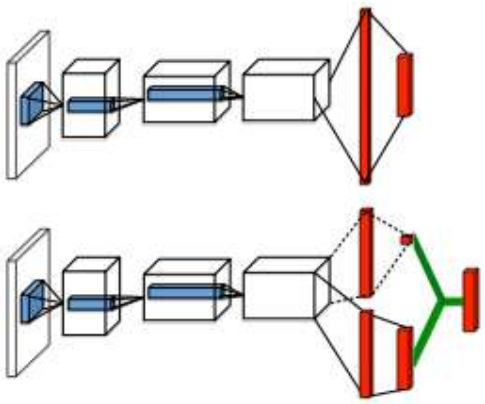


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

The Dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision. Intuitively, the Dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.

The Dueling architecture represents both the value $V(s)$ and advantage $A(s,a)$ functions with a single deep model whose output combines the two to produce a state-action value $Q(s,a)$. Unlike in advantage updating, the representation and algorithm are decoupled by construction.

$$A^{\pi}(s,a)=Q^{\pi}(s,a)-V^{\pi}(s).$$

The value function V measures the how good it is to be in a particular state s . The Q function, however, measures the value of choosing a particular action when in this state. Now, using the definition of advantage, we might be tempted to construct the aggregating module as follows:

$$Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)+A(s,a;\theta,\alpha),$$

where θ denotes the parameters of the convolutional layers, while α and β are the parameters of the two streams of fully-connected layers.

Unfortunately, the above equation is unidentifiable in the sense that given Q we cannot recover V and A uniquely; for example, there are uncountable pairs of V and A that make Q values to zero. To address this issue of identifiability, we can force the advantage function estimator to have zero advantage at the chosen action. That is, we let the last module of the network implement the forward mapping.

$$Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)+(A(s,a;\theta,\alpha)-max_{a'\in|A|}A(s,a';\theta,\alpha)).$$

This formula guarantees that we can recover the unique V and A , but the optimization is not so stable because the advantages have to compensate any change to the optimal action's advantage. Due to the reason, an alternative module that replaces the max operator with an average is proposed:

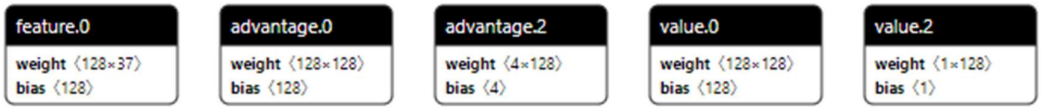
$$Q(s,a;\theta,\alpha,\beta)=V(s;\theta,\beta)+(A(s,a;\theta,\alpha)-(1/|A|)\sum_{a'}A(s,a';\theta,\alpha)).$$

Unlike the max advantage form, in this formula, the advantages only need to change as fast as the mean, so it increases the stability of optimization.

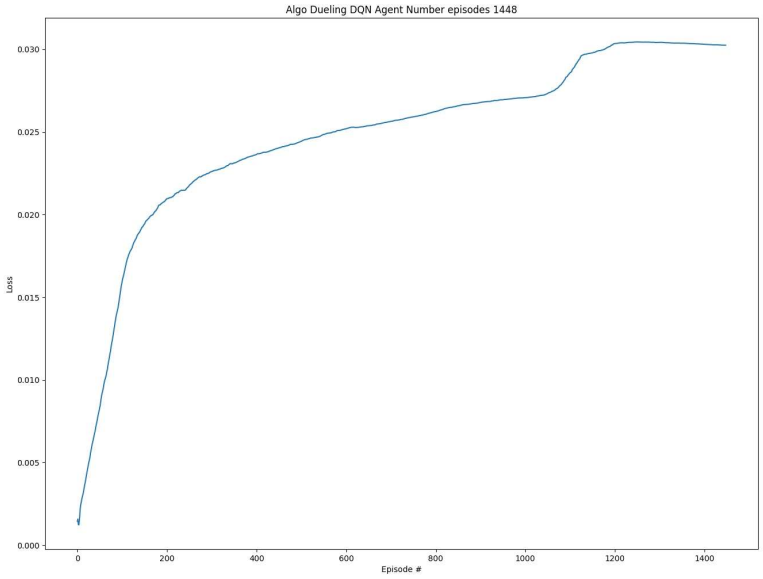
Dueling DQN Network Code Snippet

```
class DDQN(nn.Module):
    """
    Dueling DQ Network . State + advantage value functions
    """
    def __init__(self, num_inputs, num_outputs):
        super(DDQN, self).__init__()
        # input layer = size observation spaces
        self.feature = nn.Sequential(
            nn.Linear(num_inputs, 128),
            nn.ReLU()
        )
    # hidden layer which calculate the state-value function
    # output as in DQN number of possible actions
        self.advantage = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, num_outputs)
        )
     # hidden layer for advantage function
     # option continuous value for advantage function
        self.value = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )
    # forward propagation of calculations and combination advantage and value function
    def forward(self, x):
        x = self.feature(x)
        advantage = self.advantage(x)
        value = self.value(x)
        return value + advantage - advantage.mean()
```

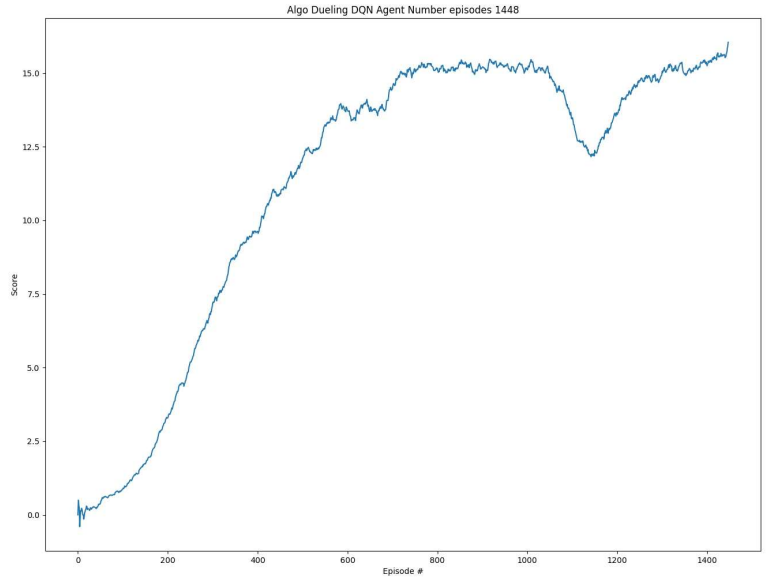
Network Architecture



Loss 2000 Episodes (solved environment 16 reward at 1448 episodes)



Scores mean 2000 episodes (solved environment 16 reward at 1448 episodes)



Type 3 Dueling Network (Double DQN) with prioritized experience replay

https://adventuresinmachinelearning.com/sumtree-introduction-python/

https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/

Implementation of prioritized experience replay. Adapted from:

https://github.com/rlcode/per/blob/master/prioritized_memory.py

A binary sum-tree. See Appendix B.2.1. in

https://arxiv.org/pdf/1511.05952.pdf

Using a replay memory leads to design choices at two levels: which experiences to store, and which experiences to replay (and how to do so). This paper addresses only the latter: making the most effective use of the replay memory for learning, assuming that its contents are outside of our control.

The central component of prioritized replay is the criterion by which the importance of each transition is measured. A reasonable approach is to use the magnitude of a transition's TD error δ, which indicates how 'surprising' or unexpected the transition is. This algorithm stores the last encountered TD error along with each transition in the replay memory. The transition with the largest absolute TD error is replayed from the memory. A Q-learning update is applied to this transition, which updates the weights in proportion to the TD error. One thing to note that new transitions arrive without a known TD-error, so it puts them at maximal priority to guarantee that all experience is seen at least once. (see store method)

We might use 2 ideas to deal with TD-error: 1. greedy TD-error prioritization, 2. stochastic prioritization. However, greedy TD-error prioritization has a severe drawback. Greedy prioritization focuses on a small subset of the experience: errors shrink slowly, especially when using function approximation, meaning that the initially high error transitions get replayed frequently. This lack of diversity that makes the system prone to over-fitting. To overcome this issue, we will use a stochastic sampling method that interpolates between pure greedy prioritization and uniform random sampling.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. In practice, we use additional term ϵ in order to guarantee all transactions can be possibly sampled: $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant.

One more. Let's recall one of the main ideas of DQN. To remove correlation of observations, it uses uniformly random sampling from the replay buffer. Prioritized replay introduces bias because it doesn't sample experiences uniformly at random due to the sampling proportion corresponding to TD-error. We can correct this bias by using importance-sampling (IS) weights

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^{\beta}$$

that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$. These weights can be folded into the Q-learning update by using $w_i\delta_i$ instead of δ_i. In typical reinforcement learning scenarios, the unbiased nature of the updates is most important near convergence at the end of training, we therefore exploit the flexibility of annealing the amount of importance-sampling correction over time, by defining a schedule on the exponent β that reaches 1 only at the end of learning.

I implement the Prioritized Experience replay buffer based on an implementation of a sumTree, (see sumtree.py in src) which keeps the weights of each step. The error is measured as Temporal Differential error calculated to keep more attention on that transaction whit bigger error

```
with torch.no_grad():

# Get old Q value. Note that if continuous we need to account for batch
# dimension
    old_q = self.local_prediction(state)[action]
    # Get the new Q value.
    new_q = reward
    if not done:
        new_q += GAMMA * torch.max(
            self.qnetwork_target(
                Variable(torch.FloatTensor(next_state)).to(device)
            ).data
        )
    # Temporal Difference error in absolute to setup weight. If the error
    # is bigger get more weight
    td_error = abs(old_q - new_q)
self.qnetwork_local.train()
self.qnetwork_target.train()
# Save experience in replay memory buffer
self.memory2.add(td_error.item(), (state, action, reward, next_state,
done))
```

we use additional term ϵ to guarantee all transactions can be possibly sampled: pi=|δi|+ϵ, where ϵ is a small positive constant. value in e. The exponent α determines how much prioritization is used, with α=0 corresponding to the uniform case.

To remove correlation of observations, it uses uniformly random sampling from the replay buffer.

Prioritized replay introduces bias because it doesn't sample experiences uniformly at random due to the sampling proportion corresponding to TD-error. We can correct
this bias by using importance-sampling (IS) weights ➔

that fully compensates for the non-uniform probabilities P(i) if β=1 . $w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^{\beta}$

These weights can be folded into the Q-learning update by using wiδi
instead of δi. In typical reinforcement learning scenarios, the unbiased nature of the updates is most important near convergence at the end of training, we therefore exploit the flexibility of annealing

the amount of importance-sampling correction over time, by defining a schedule on the exponent β that reaches 1 only at the end of learning. Here instead to use and schedule we define a beta equal to a constant and an increment per sampling also constant. (Look at attributes beta and beta_increment_per_sampling)

From PrioritizedReplayBuffer class

```
def __init__(self, capacity):
    """
    :param capacity:
    """
    self.e = 0.01 # constant to add to TD error
    self.a = 0.6  # determines how much prioritization is  used, with  α=0  corresponding to the
                  # uniform case.
    self.beta = 0.6 # to correct bias by using importance-sampling
    self.beta_increment_per_sampling = 0.01# increment of beta per each step
    self.tree = SumTree(capacity)
    self.capacity = capacity

def _get_priority(self, error):
    """
    Get priority based on error
    Arguments:
        error {float} -- TD error
    Returns:
        [float] -- priority
    """
        # error is equal to error + epsilon constant , elevated to the power of alpha
    return (error + self.e) ** self.a

def add(self, error, sample):
    """Add sample to memory
    Arguments:
        error {float} -- TD error
        sample {tuple} -- tuple of (state, action, reward, next_state, done)
    """
    p = self._get_priority(error) # get priority of this sample
    self.tree.add(p, sample) # add the sample to the sumTree
```

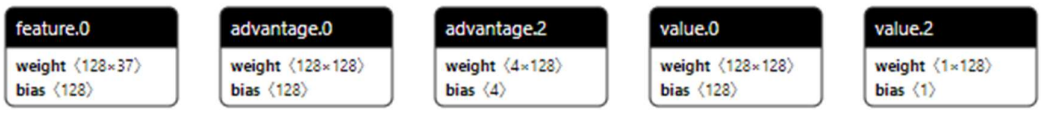
When we sample from PER buffer, we now obtain the index of this errors on the sumTree and its weights

```
def sample(self, n):
    """
    Sample from prioritized replay memory
    Arguments:
        n {int} -- sample size
    Returns:
        [tuple] -- tuple of ((state, action, reward, next_state, done), indexes, weights)
    """
    batch = []
    indexes = []
    # total return Value of root node
    segment = self.tree.total() / n
    priorities = []
    # calculate beta. max value will be 1 , but we take the min between 1 and current calculation
    self.beta = np.min([1., self.beta + self.beta_increment_per_sampling])
    for i in range(n):
        a = segment * i
        b = segment * (i + 1)
        # random value to decide which side of the tree explore
        s = random.uniform(a, b)
        (idx, p, data) = self.tree.get(s)
        if p > 0: # add only if weight is hight than 0
            priorities.append(p)
            batch.append(data)
            indexes.append(idx)

    # Calculate importance scaling for weight updates
    sampling_probabilities = priorities / self.tree.total()
    weights = np.power(self.tree.n_entries * sampling_probabilities, -self.beta)
    # Paper states that for stability always scale by 1/max w_i so that we only scale downwards
    weights /= weights.max()

    # Extract (s, a, r, s', done)
    batch = np.array(batch).transpose()
    states = np.vstack(batch[0])
    actions = list(batch[1])
    rewards = list(batch[2])
    next_states = np.vstack(batch[3])
    dones = batch[4].astype(int)

    return (states, actions, rewards, next_states, dones), indexes, weights
```

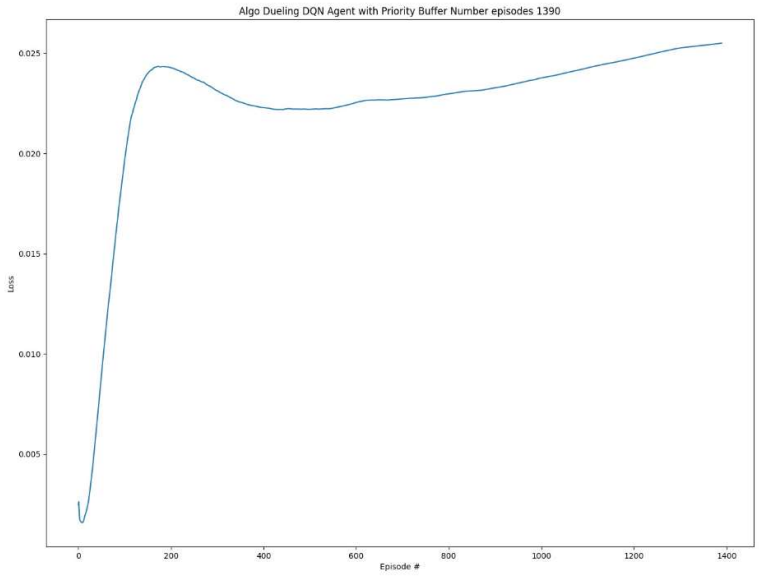
Last, when we are training, we update the error weights on the SumTree

```
# Dueling Network with Priority buffer
q_local_argmax = self.qnetwork_local(next_states).detach().argmax(dim=1).unsqueeze(1)
q_targets_next = self.qnetwork_target(next_states).gather(1, q_local_argmax).detach()
# Get Q values for chosen action
predictions = self.qnetwork_local(states).gather(1, actions)
# Calculate TD targets
targets = (rewards + (GAMMA * q_targets_next * (1 - dones)))
# Update priorities. Calculate current TD errors based in new predictions
errors = torch.abs(predictions - targets).data.cpu().numpy()
for i in range(len(errors)):
   self.memory2.update(idxs[i], errors[i])
```

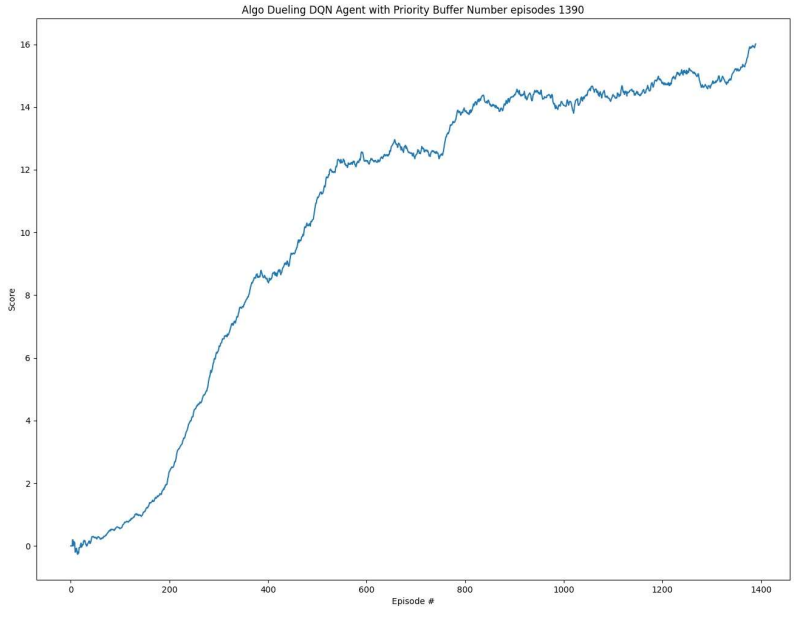
Network Architecture



Loss Dueling DQN with PER 2000 episodes (16 reward at 1390)



Reward Dueling DQN with PER 2000 episodes (16 reward at 1390)



Type 4 Categorical DQN or Distributional RL (also called C51)

Dabney, W., Rowland, M., Bellemare, M. G., and Munos, R. (2017). Distributional Reinforcement Learning with Quantile Regression. arXiv:1710.10044 [cs, stat] - > https://arxiv.org/pdf/1707.06887.pdf

Bellemare et al. (2017) proposed to learn the value distribution (the probability distribution of the returns) through a modification of the Bellman equation. They show that learning the complete distribution of rewards instead of their mean leads to performance improvements on Atari games over modern variants of DQN.

Their proposed categorical DQN (also called C51) has an architecture based on DQN, but where the output layer predicts the distribution of the returns for each action **a** in state **s**, instead of its mean $Q^{\pi}(s,a)$. In practice, each action a is represented by N output neurons, who encode the support of the distribution of returns. If the returns take values between V_{min} and V_{max}, one can represent their distribution Z by taking N discrete "bins" (called atoms in the paper) in that range

https://julien-vitay.net/deeprl/DistributionalRL.html

The name distributional RL can be a bit misleading and may conjure up images of multilayer distributed networks of DQN all working together. Well, that indeed may be a description of distributed RL, but distribution RL is where we try and find the value distribution that DQN is predicting, that is, not just find the maximum or mean value but understanding the data distribution that generated it. This is quite like both intuition and purpose for PG methods. We do this by projecting our known or previously predicted distribution into a future or future predicted distribution.

The authors argued the importance of learning the distribution of returns instead of the expected return, and they proposed to model such distributions with probability masses placed on a discrete support z, where z is a vector with $N_{atoms} \in N^{+}$ atoms, defined by $z_i = V_{min} + ((i-1)V_{max} - V_{min)} / N - 1$ for $i \in \{1,...,N_{atoms}\}$.

The key insight is that return distributions satisfy a variant of Bellman's equation. For a given state S_t and action A_t, the distribution of the returns under the optimal policy $\pi*$ should match a target distribution defined by taking the distribution for the next state S_{t+1} and action $a^{*}_{t+1} = \pi^{*}(S_{t+1})$, contracting it towards zero according to the discount, and shifting it by the reward (or distribution of rewards, in the stochastic case). A distributional variant of Q-learning is then derived by first constructing a new support for the target distribution, and then minimizing the Kullbeck-Leibler divergence between the distribution dt and the target distribution

$$d'_t = (R_{t+1} + \gamma_{t+1} z, p_{\theta}\hat{}(S_{t+1}, a\hat{}^{*}_{t+1})), D_{KL}(\varphi_z d'_t \| d_t).$$

Here φ_z is a L2-projection of the target distribution onto the fixed support z, and $a\hat{}^{*}_{t+1} = \text{argmax}_a q_{\theta}\hat{}(S_{t+1}, a)$ is the greedy action with respect to the mean action values $q_{\theta}\hat{}(S_{t+1}, a) = z^{T} p_{\theta}(S_{t+1}, a)$ in state S_{t+1}.

Implementation on Pytorch

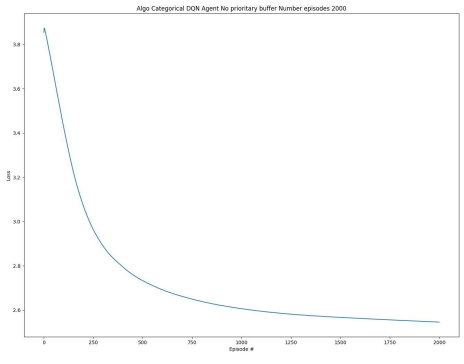
```
class Categorical_DQN(nn.Module):
   def __init__(
         self,
         in_dim: int,
         out_dim: int,
         atom_size: int, # number of bins
         support: torch.Tensor
   ):
      """
           Initialization. The Network is the same than a plain DQN
      """
      super(Categorical_DQN, self).__init__()

      self.support = support
      self.out_dim = out_dim
      self.atom_size = atom_size

      self.layers = nn.Sequential(
         nn.Linear(in_dim, 128),
         nn.ReLU(),
         nn.Linear(128, 128),
         nn.ReLU(),
         nn.Linear(128, out_dim * atom_size)
      )
   def forward(self, x: torch.Tensor) -> torch.Tensor:
      """
           Forward method implementation. The output of this Network is a matrix size number of
           actions (rows) where each action have atom_size slices / bins of a distribution
       """
      dist = self.dist(x)
      q = torch.sum(dist * self.support, dim=2)

      return q
   def dist(self, x: torch.Tensor) -> torch.Tensor:
      """
      Get distribution per each atom. Here we get a distribution per each action represented by a N
(atom_size) discrete bins. We get these distributions applying SoftMax to each bin
      """
      q_atoms = self.layers(x).view(-1, self.out_dim, self.atom_size)
      dist = F.softmax(q_atoms, dim=-1)
      dist = dist.clamp(min=1e-3)  # clapping the lower band for avoiding nans

      return dist
```

Loss calculations. There is a very good and detailed explanation of how is use quantile regression to

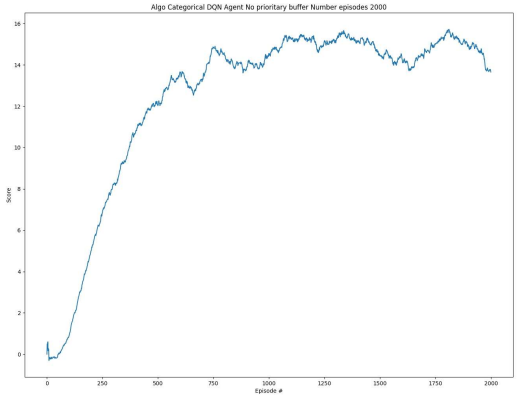
calculate the loss in https://julien-vitay.net/deeprl/DistributionalRL.html

```
# Categorical DQN algorithm
# factor to escalate the rewards according to our min , max and number of bins
    delta_z = float(self.v_max - self.v_min) / (self.atom_size - 1)

    with torch.no_grad():
      # 1) Computation of the Bellman update TZθ(s,a). They simply compute translated values for
      # each zi according to:              Tzi=r+γzi
      # and clip the obtained value to [Vmin,Vmax]. The reward r translates the distribution of
      # atoms, while  the discount rate γ scales it. prediction of next action using next_state on
      # target
      next_action = self.qnetwork_target(next_state).argmax(1)
      # Expected future rewards distribution based in atom_size bins of each possible action
      next_dist = self.qnetwork_target.dist(next_state)
      next_dist = next_dist[range(BATCH_SIZE), next_action]
      # Calculate expected reward
      t_z = reward + (1 - done) * gamma * self.support
      # clip reward to be in between min and max
      t_z = t_z.clamp(min=self.v_min, max=self.v_max)
      # escalate the reward
      b = (t_z - self.v_min) / delta_z
      # calculate the floor and ceiling of this reward Ex. reward = 2.4 --> floor 2, ceil 3
      l = b.floor().long()
      u = b.ceil().long()
      # Placeholders for alignment target distribution to support of local network distributions
      offset = (  torch.linspace( 0, (BATCH_SIZE - 1) * self.atom_size, BATCH_SIZE ).long()
            .unsqueeze(1).expand(BATCH_SIZE, self.atom_size).to(device) )

      proj_dist = torch.zeros(next_dist.size(), device=device)
      # 2) calculate the bins. Distribution of the probabilities of TZθ(s,a) on the support of Zθ(s,a).
      # The projected atom Tzi lie between two “real” atoms zl and zu, with a non-integer index b
      # (for example, b=3.4, l=3 and u=4). The corresponding probability pb(s’,a’;θ) of the next
      # greedy action (s’,a’) is “spread” to its neighbours through a local interpolation depending on
      # the distances between b, l and u:
      proj_dist.view(-1).index_add_(
        0, (l + offset).view(-1), (next_dist * (u.float() - b)).view(-1)
      )
      proj_dist.view(-1).index_add_(
        0, (u + offset).view(-1), (next_dist * (b - l.float())).view(-1)
      )
    # expected immediate rewards distribution for this(state , action) using local Network
    # 3) Minimizing the statistical distance between ΦTZθ(s,a) and Zθ(s,a). Now that the Bellman
    # update has the same support as the value distribution, we can minimize the KL divergence
    # between the two for a single transition:
    #                   L(θ)=DKL(ΦTZθ’(s,a)|Zθ(s,a))
    # using a target network θ’ for the target. It is to be noted that minimizing the KL divergence is
    # the same as minimizing the cross-entropy between the two, as in classification tasks:
    #                 L(θ)=−∑i(ΦTZθ’(s,a))ilogpi(s,a;θ)
    dist = self.qnetwork_local.dist(state)
    log_p = torch.log(dist[range(BATCH_SIZE), action])
    loss = -(proj_dist * log_p).sum(1).mean()
```

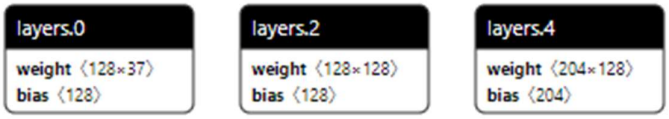
Loss Categorical DQN 2000 episodes max 51 atoms min 0 max 40



Reward Categorical DQN 2000 episodes max 51 atoms min 0 max 40



Network Architecture



Notice: that output is 204 =

51 atoms * 4 actions

Type 5 Dueling Network (Double DQN) with Noisy Net and PER

M. Fortunato et al., "Noisy Networks for Exploration." arXiv preprint arXiv:1706.10295, 2017.

NoisyNet is an exploration method that learns perturbations of the network weights to drive exploration. The key insight is that a single change to the weight vector can induce a consistent, and potentially very complex, state-dependent change in policy over multiple time steps.

Firstly, let's take a look into a linear layer of a neural network with p inputs and q outputs, represented by

$$y=wx+b,$$

where $x \in \mathbb{R}^p$ is the layer input, $w \in \mathbb{R}^{q \times p}$, and $b \in \mathbb{R}$ the bias.

The corresponding noisy linear layer is defined as:

$$y=(\mu^w+\sigma^w \odot \epsilon^w)x+\mu^b+\sigma^b \odot \epsilon^b,$$

where $\mu^w + \sigma^w \odot \epsilon^w$ and $\mu^b + \sigma^b \odot \epsilon^b$ replace w and b in the first linear layer equation. The parameters $\mu^w \in \mathbb{R}^{q \times p}, \mu^b \in \mathbb{R}^q, \sigma^w \in \mathbb{R}^{q \times p}$ and $\sigma^b \in \mathbb{R}^q$ are learnable, whereas $\epsilon^w \in \mathbb{R}^{q \times p}$ and $\epsilon^b \in \mathbb{R}^q$ are noise random variables which can be generated by one of the following two ways:

Independent Gaussian noise: the noise applied to each weight and bias is independent, where each random noise entry is drawn from a unit Gaussian distribution. This means that for each noisy linear layer, there are $pq + q$ noise variables (for p inputs to the layer and q outputs).

Factorised Gaussian noise: This is a more computationally efficient way. It produces 2 random Gaussian noise vectors (p, q) and makes $pq + q$ noise entries by outer product as follows:

$$\epsilon i,jw=f(\epsilon i)f(\epsilon j),$$

$$\epsilon jb=f(\epsilon i),$$

were $f(x) = sgn(x)\sqrt{|x|}.$

Implementation References

https://github.com/higgsfield/RL-Adventure/blob/master/5.noisy%20dqn.ipynb

https://github.com/Kaixhin/Rainbow/blob/master/model.py

Our implementation is a Factorised Gaussian Noise, where we are producing noise for the w and b or this equation

$$y=wx+b,$$

NoisyNet, a deep reinforcement learning agent with parametric noise added to its weights and show that the induced stochasticity of the agent's policy can be used to aid efficient exploration. The parameters of the noise are learned with gradient descent along with the remaining network weights. Let's look to the implementation in Pytorch. In our implementation we are introducing Noisy layers in a Dueling DQN architecture. We define new parameters which define the gaussian distribution of weights and bias parameters

```
class NoisyLinear(nn.Module):
    """
    Noisy linear module for NoisyNet.
    References:

    https://github.com/higgsfield/RL-Adventure/blob/master/5.noisy%20dqn.ipynb
    https://github.com/Kaixhin/Rainbow/blob/master/model.py

    Attributes:
        in_features (int): input size of linear module
        out_features (int): output size of linear module
        std_init (float): initial std value
        weight_mu (nn.Parameter): mean value weight parameter
        weight_sigma (nn.Parameter): std value weight parameter
        bias_mu (nn.Parameter): mean value bias parameter
        bias_sigma (nn.Parameter): std value bias parameter

    """

    def __init__(self, in_features: int, out_features: int, std_init: float = 0.5):
        """Initialization."""
        super(NoisyLinear, self).__init__()

        self.in_features = in_features
        self.out_features = out_features
        self.std_init = std_init

        self.weight_mu = nn.Parameter(torch.Tensor(out_features, in_features))
        self.weight_sigma = nn.Parameter(
            torch.Tensor(out_features, in_features)
        )
        self.register_buffer(
            "weight_epsilon", torch.Tensor(out_features, in_features)
        )

        self.bias_mu = nn.Parameter(torch.Tensor(out_features))
        self.bias_sigma = nn.Parameter(torch.Tensor(out_features))
        self.register_buffer("bias_epsilon", torch.Tensor(out_features))

        self.reset_parameters()
        self.reset_noise()

    def reset_parameters(self):
        """Reset trainable network parameters (factorized gaussian noise)."""
        mu_range = 1 / math.sqrt(self.in_features)
        self.weight_mu.data.uniform_(-mu_range, mu_range)
        self.weight_sigma.data.fill_(
            self.std_init / math.sqrt(self.in_features)
        )
        self.bias_mu.data.uniform_(-mu_range, mu_range)
        self.bias_sigma.data.fill_(
            self.std_init / math.sqrt(self.out_features)
        )
```

And in the forward method we are just introducing in our calculation 2 new parameters to optimize

Apart of x, now we have weight and bias defined by theirs mean and their respective standard deviation

```
def reset_noise(self):
        """Make new noise."""
        epsilon_in = self.scale_noise(self.in_features)
        epsilon_out = self.scale_noise(self.out_features)

        # outer product
        self.weight_epsilon.copy_(epsilon_out.ger(epsilon_in))
        self.bias_epsilon.copy_(epsilon_out)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward method implementation.

        We don't use separate statements on train / eval mode.
        It doesn't show remarkable difference of performance.
        """
        return F.linear(
            x,
            self.weight_mu + self.weight_sigma * self.weight_epsilon,
            self.bias_mu + self.bias_sigma * self.bias_epsilon,
        )

    @staticmethod
    def scale_noise(size: int) -> torch.Tensor:
        """Set scale to make noise (factorized gaussian noise)."""
        x = torch.randn(size)

        return x.sign().mul(x.abs().sqrt())
```

As we are introducing noise on the weights of our Linear layers, our Dueling DQN class now change slightly

```
# set common feature layer. it doesn't change from original implementation
    self.feature_layer = nn.Sequential(
      nn.Linear(in_dim, 128),
      nn.ReLU(),
    )

    # set advantage layer, which now is a noisy layer
    self.advantage_hidden_layer = NoisyLinear(128, 128)
    self.advantage_layer = NoisyLinear(128, out_dim )

    # set value layer , which now is a noisy layer
    self.value_hidden_layer = NoisyLinear(128, 128)
    self.value_layer = NoisyLinear(128, 1)
```

Forward function does not change in comparison with the original implementation of Dueling DQN

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    """
    Forward method implementation
    """
    feature = self.feature_layer(x)
    value = self.value_layer(feature)
    advantage = self.advantage_layer(feature)
    # forward is defined as the value of this action together with the advantage to take this action
    # -minus the mean of the expected advantage function values from this state to done
    q = value + advantage - advantage.mean(dim=-1, keepdim=True)

    return q
```

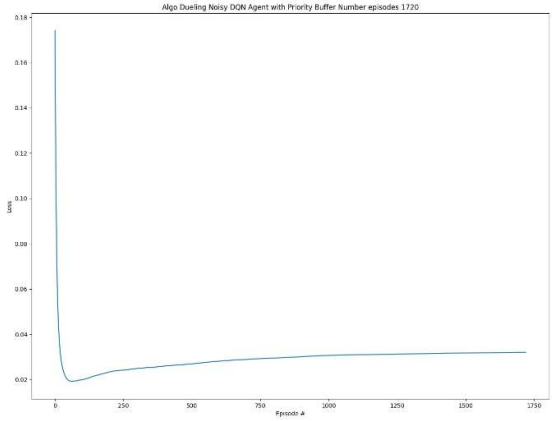
The loss calculation does not change

```
# Double DQN take the max in between the prediction calculated by the target network and the
# prediction of the local network using the next_state
        Q_targets_next = self.qnetwork_target(next_states).gather(
            1, self.qnetwork_local(next_states).argmax(dim=1, keepdim=True)).detach()

        # Compute Q targets for current states. same as DQN
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        # loss = F.mse_loss(Q_expected, Q_targets) replaced mse loss with smooth for gradient
        # clipping
        loss = F.smooth_l1_loss(Q_expected, Q_targets)
        # record loss
        self.losses.append(loss.item())

        # Minimize the loss and backpropagate it
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

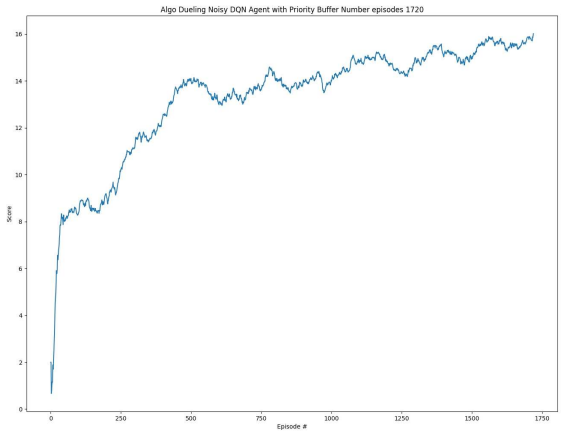
The only that change in our Agent is that now we don't use Epsilon Greedy algo for exploitation/exploration Phases, as we are introducing the entropy for exploration on the noisy layers

```
    if self.noisy: # If NOISY LAYER. we don't use epsilon-greedy algo
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        # Choose action values according to local model
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()
        return np.argmax(action_values.cpu().data.numpy())
```

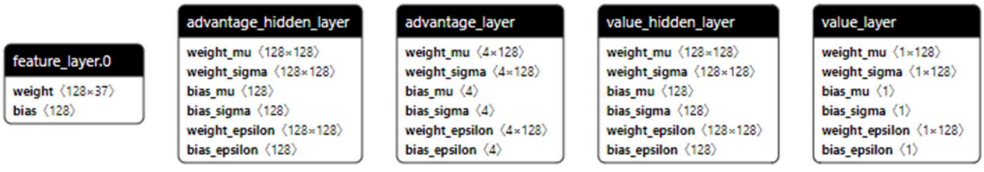
Loss Duelling DQN with Noisy Layer and PER (Reward 16 at 1720)



Rewards Duelling DQN with Noisy Layer and PER (Reward 16 at 1720)



Network Architecture



Type 6 N-Steps Learning DQN Agent

http://incompleteideas.net/papers/sutton-88-with-erratum.pdf

N-Step TD also call n-Step SARSA will perform an update based on the next n rewards, and the estimated value of the corresponding state (n steps ahead).

Q-learning accumulates a single reward and then uses the greedy action at the next step to bootstrap. Alternatively, forward-view multi-step targets can be used (Sutton 1988). We call it Truncated N-Step Return from a given state S_t. It is defined as,

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}.$$

A multi-step variant of DQN is then defined by minimizing the alternative loss,

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

Multi-step targets with suitably tuned n often lead to faster learning (Sutton and Barto 1998).

We use two buffers: memory and memory_n for 1-step transitions and n-step transitions respectively. It guarantees that any paired 1-step and n-step transitions have the same indices (See step method for more details). Due to the reason, we can sample pairs of transitions from the two buffers once we have indices for samples.

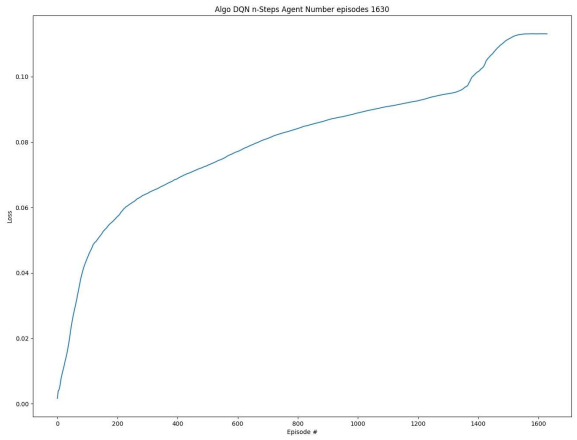
One thing to note that we are going to combine 1-step loss and n-step loss so as to control high-variance / high-bias trade-off.

```
def step(self, state, action, reward, next_state, done):
    """
    Step implementation
    :param state: state agent
    :param action: action agent
    :param reward: reward r after taken action a in state s
    :param next_state: state after taken action a in state s
    :param done:
    :return:
    """
    # Save experience in replay memory
    if self.train:
        # PER: increase beta
        fraction = min(self.update_cnt / self.num_frames, 1.0)
        self.beta = self.beta + fraction * (1.0 - self.beta)

        self.transition += [reward, next_state, done]
        # N-step transition add to n_step buffer
        if self.use_n_step:
            one_step_transition = self.memory_n.store(*self.transition)
        # 1-step transition
        else:

            one_step_transition = self.transition

        # add a single step transition
        if one_step_transition:
            self.memory.store(*one_step_transition)
        # learn
        self.learn(GAMMA)
```

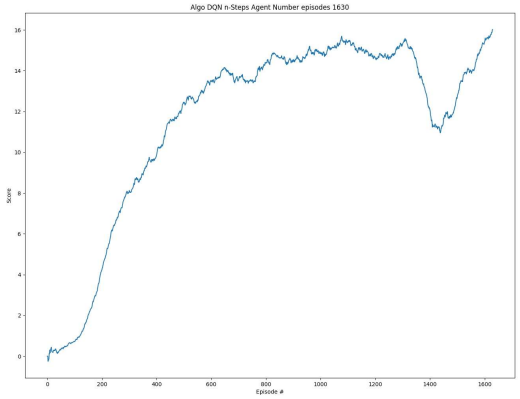
When calculating loss, it is combined 1 step with n-step loss

```
    if len(self.memory) > BATCH_SIZE: # sample from simple buffer.
        experiences = self.memory.sample_batch()
        indices = experiences["indices"]
        loss = self._compute_dqn_loss(experiences, gamma)
        # N-step Learning loss we are gonna combine 1-step loss and n-step loss so as to
        # prevent high-variance.
        if self.use_n_step:
       # if we use n_step we sample from n_step buffer using the indexes we got before
            samples = self.memory_n.sample_batch_from_idxs(indices)
            gamma = gamma ** self.n_step
            n_loss = self._compute_dqn_loss(samples, gamma)
            loss += n_loss
        # record loss
        self.losses.append(loss.item())
        # backward gradients
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

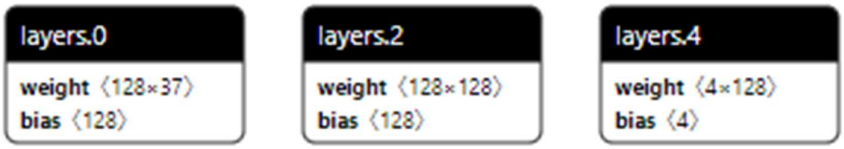
Loss Agent DQN n-steps (3 steps) 2000 episodes (solved env → 16 scores in 1630)



Scores Agent DQN n-steps (3 steps) 2000 episodes (solved env → 16 scores in 1630)



Network Architecture



Type 7 Rainbow DQN

https://arxiv.org/pdf/1710.02298.pdf

We will integrate all the following seven components into a single integrated agent, which is called Rainbow

- DQN
- Double DQN
- Prioritized Experience Replay
- Dueling Network
- Noisy Network
- Categorical DQN
- N-step Learning

Our Network architecture change as now we have a dueling architecture together with Noisy layers and Categorical distributions

```
class Rainbow_DQN(nn.Module):
    def __init__(
            self,
            in_dim: int,
            out_dim: int,
            atom_size: int,
            support: torch.Tensor
    ):
        """Initialization."""
        super(Rainbow_DQN, self).__init__()

        self.support = support
        self.out_dim = out_dim
        self.atom_size = atom_size # number of bin Categorical distribution

        # set common feature layer
        self.feature_layer = nn.Sequential(
            nn.Linear(in_dim, 128),
            nn.ReLU(),
        )
        # set advantage layer based on Noisy layers
        self.advantage_hidden_layer = NoisyLinear(128, 128)
        self.advantage_layer = NoisyLinear(128, out_dim * atom_size)

        # set value layer based on Noisy layers
        self.value_hidden_layer = NoisyLinear(128, 128)
        self.value_layer = NoisyLinear(128, atom_size)
```

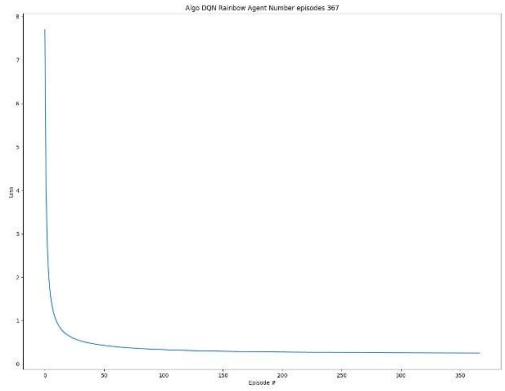
```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    """
        Forward method implementation.
    """

    dist = self.dist(x)
    q = torch.sum(dist * self.support, dim=2)

    return q

def dist(self, x: torch.Tensor) -> torch.Tensor:
    """
        Get distribution for atoms/ Bins
    """
    feature = self.feature_layer(x)
    adv_hid = F.relu(self.advantage_hidden_layer(feature))
    val_hid = F.relu(self.value_hidden_layer(feature))
    # advantage layer shape number of output * number of atoms/bins
    advantage = self.advantage_layer(adv_hid).view(
        -1, self.out_dim, self.atom_size)
    # value layer shape 1 * number of atoms/bins
    value = self.value_layer(val_hid).view(-1, 1, self.atom_size)
    q_atoms = value + advantage - advantage.mean(dim=1, keepdim=True)
    # we apply the softmax function to the combined layer
    dist = F.softmax(q_atoms, dim=-1)
    dist = dist.clamp(min=1e-3)  # clapping for avoiding nans

    return dist

def reset_noise(self):
    """
        Reset all noisy layers.
    """
    self.advantage_hidden_layer.reset_noise()
    self.advantage_layer.reset_noise()
    self.value_hidden_layer.reset_noise()
    self.value_layer.reset_noise()
```

As we are using Noisy layers, we don't use in this implementation Epsilon Greedy algo to select new action.
As we are combining Categorical DQN with Double DQN, therefor Here, we use qnetwork_local instead of qnetwork_target to obtain the target actions.

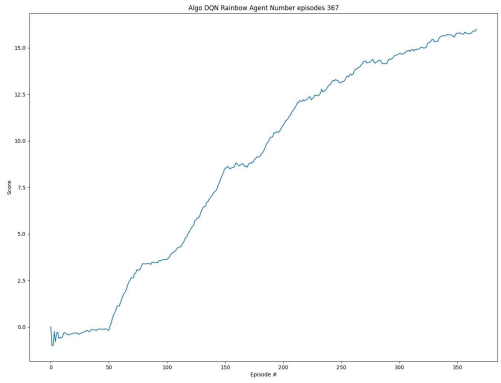
```
def _compute_dqn_loss(self, samples: Dict[str, np.ndarray], gamma: float) ->
torch.Tensor:
    """Return categorical dqn loss."""
    state = torch.FloatTensor(samples["obs"]).to(device)
    next_state = torch.FloatTensor(samples["next_obs"]).to(device)
    action = torch.LongTensor(samples["acts"]).to(device)
    reward = torch.FloatTensor(samples["rews"].reshape(-1, 1)).to(device)
    done = torch.FloatTensor(samples["done"].reshape(-1, 1)).to(device)
    # Categorical DQN algorithm
    # factor to escalate the rewards according to our min , max and number
    # of bins
    delta_z = float(self.v_max - self.v_min) / (self.atom_size - 1)
```

```
    with torch.no_grad():
        # 1) Computation of the Bellman update TZθ(s,a). They simply
        # compute translated values for each zi according to: Tzi=r+γzi
        # and clip the obtained value to [Vmin,Vmax]. The reward r
        #translates the distribution of atoms, while  the discount rate γ
        # scales it. prediction of next action using next_state on target
        # The idea of Double Q-learning is to reduce overestimations by
        # decomposing the max operation in the target into action selection
        # and action evaluation. Here, we use self.dqn instead of
        #       self.dqn_target to obtain the target actions.
        next_action = self.qnetwork_local(next_state).argmax(1)
        # Expected future rewards distribution based in atom_size bins of
        # each possible action
        next_dist = self.qnetwork_target.dist(next_state)
        next_dist = next_dist[range(BATCH_SIZE), next_action]
        # Calculate expected reward
        t_z = reward + (1 - done) * gamma * self.support
        # clip reward to be in between min and max
        t_z = t_z.clamp(min=self.v_min, max=self.v_max)
        # escalate the reward
        b = (t_z - self.v_min) / delta_z
        # calculate the floor and ceiling of this reward Ex. reward = 2.4 -
        # -> floor 2, ceil 3
        l = b.floor().long()
        u = b.ceil().long()
        # Placeholders for alignment target distribution to support of
        # local network distributions
        offset = (torch.linspace(0, (BATCH_SIZE - 1) * self.atom_size,
BATCH_SIZE).long()
                  .unsqueeze(1).expand(BATCH_SIZE, self.atom_size).to(device))

        proj_dist = torch.zeros(next_dist.size(), device=device)
        # 2) calculate the bins. Distribution of the probabilities of
        # TZθ(s,a) on the support of Zθ(s,a). # The projected atom Tzi lie
        # between two "real" atoms zl and zu, with a non-integer index b
        # (for example, b=3.4, l=3 and u=4). The corresponding probability
        # pb(s',a';θ) of the next greedy action (s',a') is "spread" to its
        # neighbours through a local interpolation depending on
        # the distances between b, l and u:
        proj_dist.view(-1).index_add_(
            0, (l + offset).view(-1), (next_dist * (u.float() - b)).view(-1)
        )
        proj_dist.view(-1).index_add_(
            0, (u + offset).view(-1), (next_dist * (b - l.float())).view(-1)
        )
    # expected immediate rewards distribution for this(state , action)
    # using local Network
    # 3) Minimizing the statistical distance between ΦTZθ(s,a) and Zθ(s,a).
    # Now that the Bellman update has the same support as the value
    # distribution, we can minimize the KL divergence between the two for a
    # single transition: L(θ)=DKL(ΦTZθ'(s,a)|Zθ(s,a))
    # using a target network θ' for the target. It is to be noted that
    # minimizing the KL divergence is the same as minimizing the cross-
    # entropy between the two, as in classification tasks:
    #                       L(θ)=-∑i(ΦTZθ'(s,a))ilogpi(s,a;θ)
    dist = self.qnetwork_local.dist(state)
    log_p = torch.log(dist[range(BATCH_SIZE), action])
    elementwise_loss = -(proj_dist * log_p).sum(1)

    return elementwise_loss
```

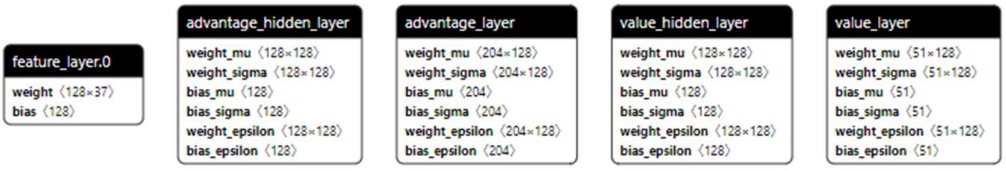
Loss DQN Rainbow (solved env → 16 scores in 367)



Reward DQN rainbow (solved env → 16 scores in 367)



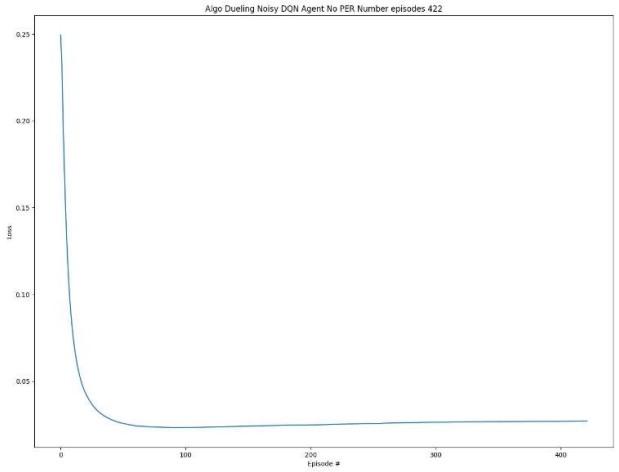
Network Architecture



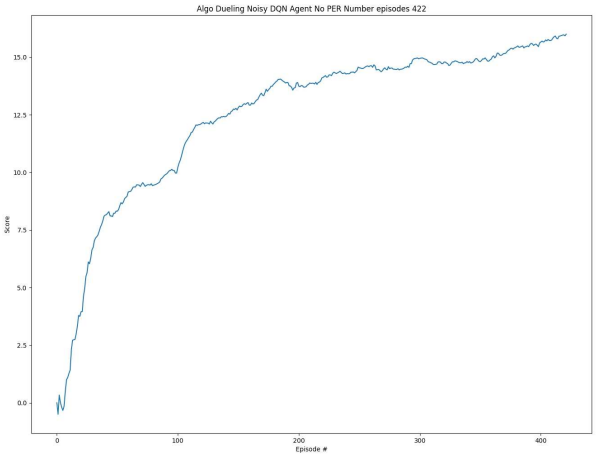
Type 8 Dueling Network (Double DQN) with Noisy Net without PER

We just made and experiment like 5 to understand the effect of PER

Loss Duelling DQN with Noisy Layer NO PER (Solved at 422 Episodes)



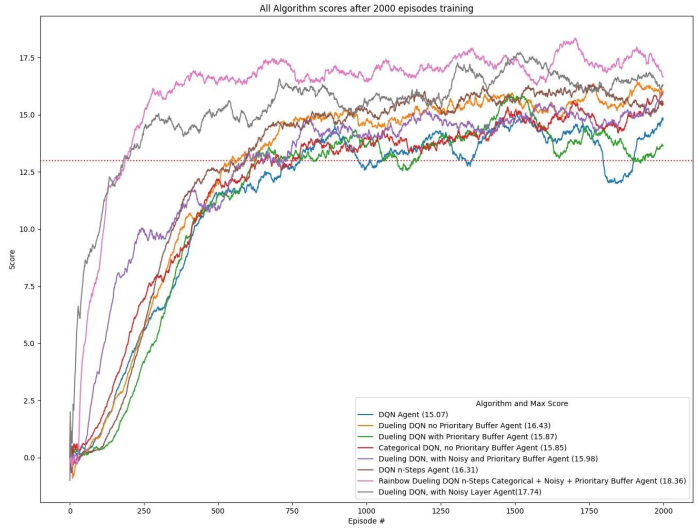
Reward Duelling DQN with Noisy Layer NO PER (Solved at 422 Episodes)



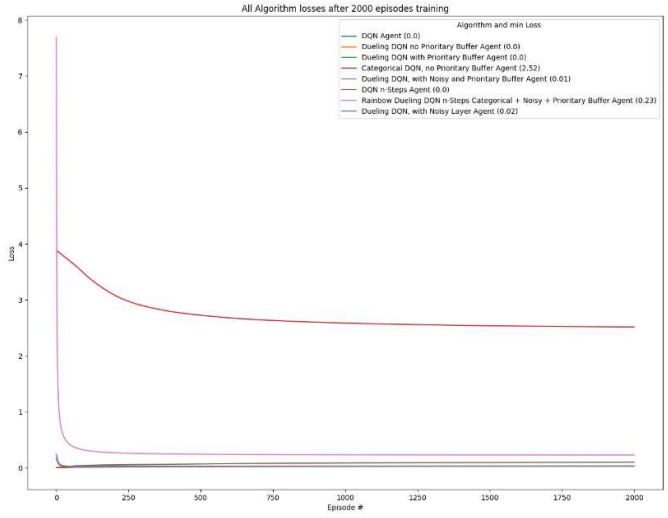
NOTE: The main take-off of using or not using PER, at least with this DQN architecture and this env is the time need to solve the environment. With PER we need 1720 episodes to get 16 as reward. Same network without PER need only 422 episodes. On play mode with PER, we get a score of 16, without a score of 22. We need to evaluate and test where is suitable use PER as some cases not only that more time and effort to solve the problem, when even learning poor policies

Comparison different algorithms

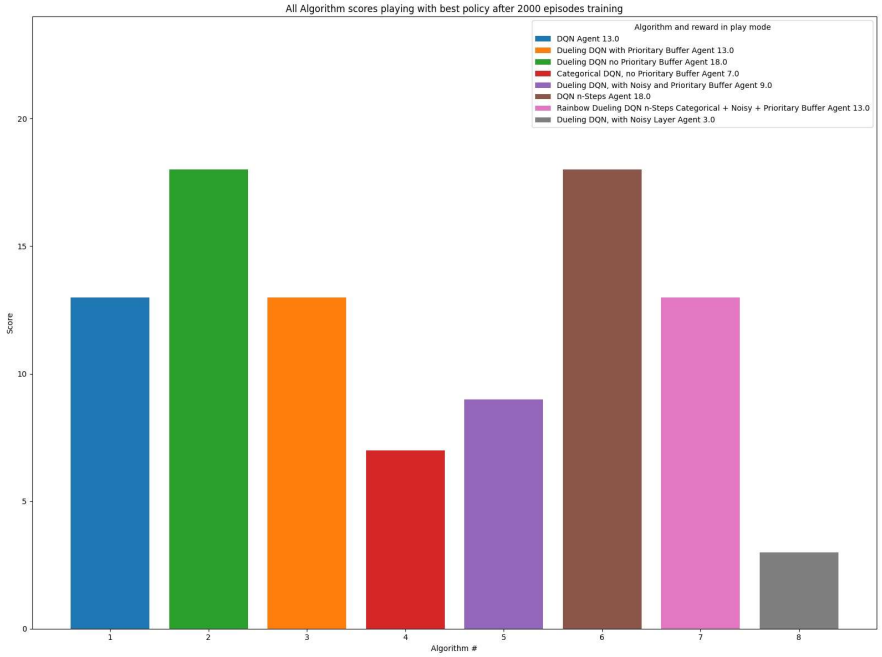
Reward 2000 Episodes



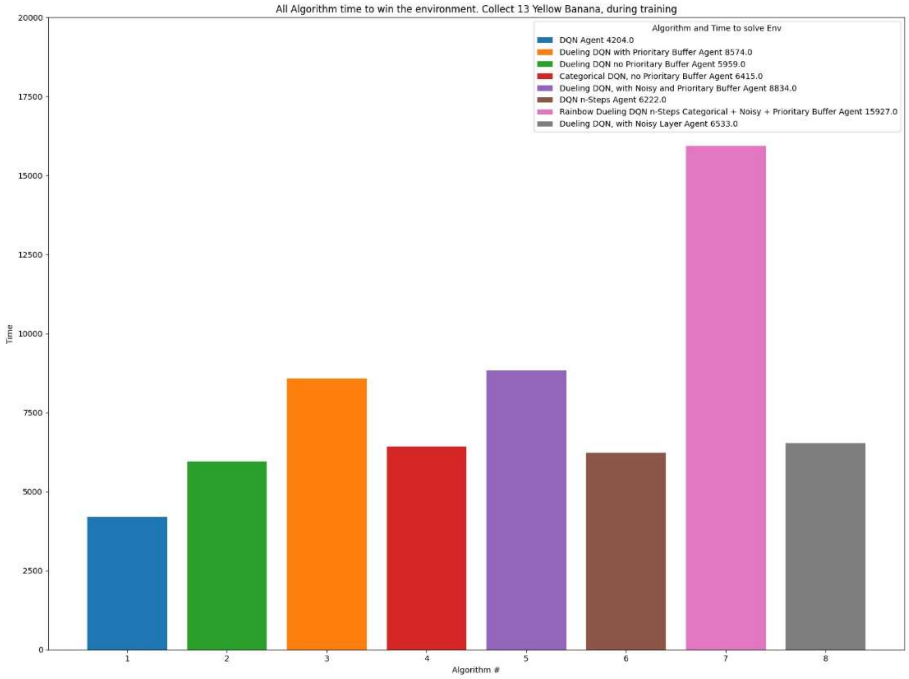
Loss 2000 Episodes



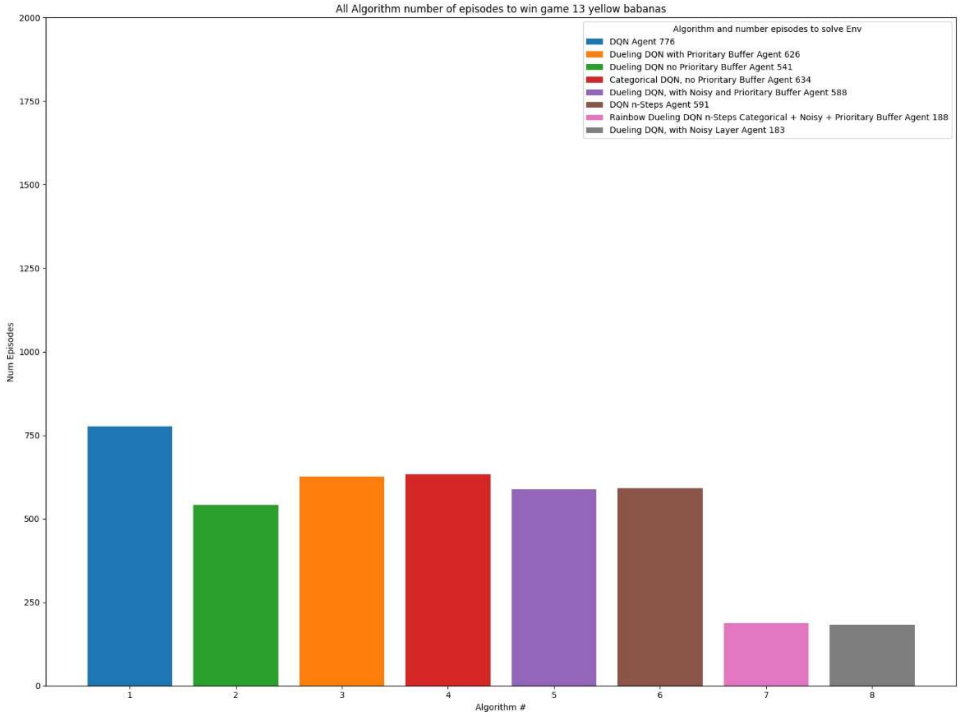
Reward of playing with best saved policy (policy which hit higher reward during training)



Training Time (Time to solve the environment. Collect 13 yellow Bananas)



Number of Episodes (to solve the environment. Collect 13 yellow Bananas)



Hyper parameter tuning

http://hyperopt.github.io/hyperopt/

In mode hp_tuning and using library Hyperopt library, I setup an example of how to optimize parameters of an agent using Bayesian Optimization. It it's just a simple example but give you a grasp of how we can optimize the parameters. There are other frameworks to optimize parameters like RL Baselines3 Zoo if we use Stable baselines library or Ray for unity RL agents, but here as this is a tailored environment, I decided to use a general optimization framework and learn how to use it in Deep RL. Here in this simple configuration, I am optimizing 3 parameters of the Vanilla DQN agent model, and I limit the trials to 30 for this experiment

I use Bayesian Optimization and my observation Space looks like this

```
# define search Space
search_space = { 'gamma': hp.loguniform('gamma' ,np.log(0.9),
np.log(0.99)),
                'batch_size' : hp.choice('batch_size', [32,64, 128]),
                 'lr': hp.loguniform('lr',np.log(1e-4), np.log(15e-3)),
                 'brain_name' : brain_name,
                 'state_size' : state_size,
                 'action_size' : action_size,
                           }
```

I am just optimizing 3 parameters

Gamma: range 0.9 to 0.99

Batch size: Choice 32,64,128

Learning Rate Range 15e-3 to 1e-4

The metric to optimize is the mean rewards 100 episodes with a maximum of 500 episodes per trial. I just limited the experiment for the purpose of the example and to limit the time of hyper parameter tuning.

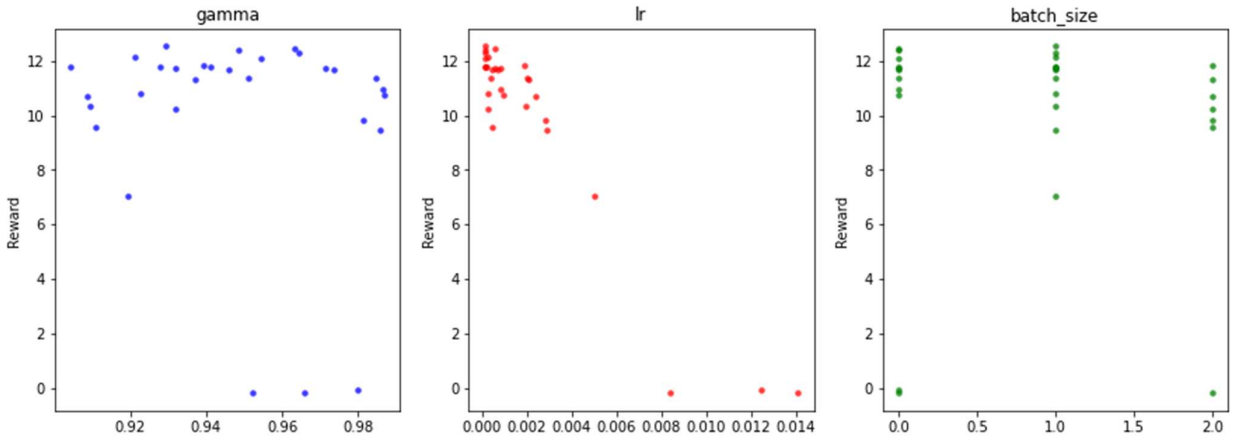
```
fmin_objective = partial(objective, env=env)
trials = Trials()
argmin = fmin(
    fn=fmin_objective,
    space=search_space,
    algo=tpe.suggest,  # algorithm controlling how hyperopt navigates the
search space
    max_evals=30,
    trials=trials,
    verbose=True
    )#
# return the best parameters
best_parms = space_eval(search_space, argmin)
```

This example best parameters

{'action_size': 4, 'batch_size': 64, 'brain_name': 'BananaBrain', 'gamma': 0.9294268596722944, 'lr': 0.000146338513448115, 'state_size': 37}

With a reward of 12.54 after 500 experiments

```
parameters = ['gamma', 'lr', 'batch_size']
colors =  ['blue', 'red', 'green']
cols = len(parameters)
f, axes = plt.subplots(nrows=1, ncols=cols, figsize=(15,5))
cmap = plt.cm.jet

for i, val in enumerate(parameters):
    xs = np.array([t['misc']['vals'][val] for t in trials.trials]).ravel()
    ys = [-t['result']['loss'] for t in trials.trials]
    xs, ys = zip(*sorted(zip(xs, ys)))
    ys = np.array(ys)
    axes[i].scatter(xs, ys, s=20, linewidth=0.01, alpha=0.75, c=colors[i])
    axes[i].set_title(val)
    axes[i].set_ylabel('Reward')
```



Conclusions

- All solvers win the environment, getting more than 13 as reward for a period of 100 episodes, but different speeds and different policy quality as we observe later in mode play
- Rainbow agents get the best performance in average but taking the longest time to train the agent and it does not produce, at least in my case the best policy. Potentially a hyper parameter tuning session will fine tune the algo. Looking at the time of training and the reward obtained in mode play the winners are in my opinion 3 and 6, duelling DQN and DQN n-steps (3) which spend a reasonable time in training and get both 18 as rewards in mode play.
- We observe that as much complex the agent is, at least with this environment, it does not that means, getting better outcomes, which demand a hyper parameter session of the algos to get the best of them in a specific environment.
- Some of the improvements like Noisy and Categorical layers, are very complex to implement and at least in my case does not show very good outcomes, but again, it would need HP tuning to get a conclusion.
- Hyper parameter tuning helps to understand how the algorithms behaves, so before to introduce an application in production, it would be mandatory to run extensive tuning and validations

Ideas for Future Works

- Introduce Multiprocessing for hyperparameter tuning and potentially for training
- Fine tune the implementations
- Explore other Architectures like Deep Recurrent Q Networks DRQN and DARQN
- Try other Unity environments and compare outcomes with what we get in this project
- Explore use of libraries like Ray Rllib of Stable Baselines 3
- Find out how to record videos

References

1. V. Mnih et al., "Human-level control through deep reinforcement learning." Nature, 518 (7540):529–533, 2015.
2. van Hasselt et al., "Deep Reinforcement Learning with Double Q-learning." arXiv preprint arXiv:1509.06461, 2015.
3. T. Schaul et al., "Prioritized Experience Replay." arXiv preprint arXiv:1511.05952, 2015.
4. Z. Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning." arXiv preprint arXiv:1511.06581, 2015.
5. M. Fortunato et al., "Noisy Networks for Exploration." arXiv preprint arXiv:1706.10295, 2017.
6. M. G. Bellemare et al., "A Distributional Perspective on Reinforcement Learning." arXiv preprint arXiv:1707.06887, 2017.
7. R. S. Sutton, "Learning to predict by the methods of temporal differences." Machine learning, 3(1):9–44, 1988.
8. M. Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning." arXiv preprint arXiv:1710.02298, 2017.
9. https://github.com/Curt-Park/rainbow-is-all-you-need
10. Hands-on Reinforcement Learning for Games (Book) Michael Lanham
11. Grokking Deep Reinforcement Learning (Book) Miguel Morales
12. Hands-on Reinforcement Learning with Python (book) by Sudharsan Ravichandiran
13. binary sum-tree. See Appendix B.2.1. in https://arxiv.org/pdf/1511.05952.pdf. Adapted implementation from https://github.com/jaromiru/AI-blog/blob/master/SumTree.py
14. SegmentTree from OpenAi repository. https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py
15. PER implementation. https://github.com/rlcode/per/blob/master/prioritized_memory.py
16. Noisy Layers . https://github.com/higgsfield/RL-Adventure/blob/master/5.noisy%20dqn.ipynb https://github.com/Kaixhin/Rainbow/blob/master/model.py