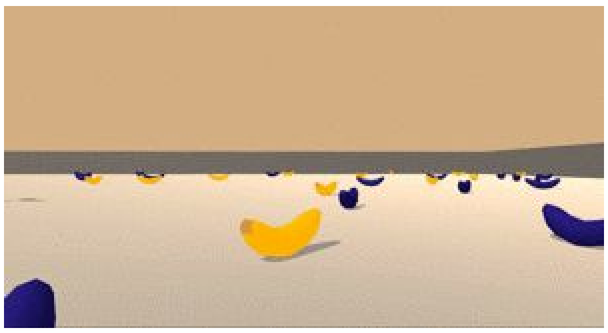
Deep RL course - Navigation project report

Author: Guillaume Boniface-Chang

Project setup

The project consists of solving a reinforcement learning environment made of a space stochastically populated with blue and yellow bananas. At each step the agent has 4 actions (move forward, move backward, turn right and turn left). Walking over a blue banana results in a -1 reward, walking over a yellow banana results in a +1 reward. The environment is considered 'won' if the agent succeeds in getting an average reward of 13 over 100 consecutive episodes. Each episode lasts 300 steps. While represented through a 3D graphical interface, the environment state is passed as a 1-D array of length 37.



The problem can be modelled as a classic Markov Decision Process, with a continuous state space made of 37 variables and a discrete action space with 4 possible values.

Proposed approach

We implement a Deep Q-Network characterized by a replay memory buffer stochastically sampled and a deep learning value function estimator. To further refine our approach, we implement different improvements over the classic DQN architecture and compare their performance on the task. The studied improvements are specifically: double DQN, dueling DQN and prioritized replay.

DQN

Our DQN implementation relies on a simple multi-layer perceptron to model the value function, with a 'relu' activation function applied at each hidden layer and enabling the model to learn non linear functions. For the training step, we use a mean squared error loss and an Adam optimizer.

*loss = mean((action_values_target - action_values_estimations) ** 2)*

The replay buffer stores state transitions with a sliding window (i.e with a maximum storage capacity and a first-in first-out policy). The training step involves sampling from the replay buffer a set of state transitions, actions and rewards. The sampling is done with uniform probability over the whole of the replay buffer.

The target value for the training steps are obtained through the Bellman equation, combining the actual reward of the sampled transition with a discounted estimation of the next state value (obtained by taking the maximum of the value estimations over all actions). We use a second model (the 'target' model) to compute the training targets, which helps stabilize the learning process by providing 'fixed' targets.

next_action = argmax(target_model.predict_action_values(next_state))
*action_value_target = reward + discount_rate * target_model.predict_value(next_state, next_action)*

The weights of the target model are updated regularly with the trained model weights. Instead of doing this updates every N steps as in the original DQN implementation, we do so at every training step but proportionally and according to a rate 'tau'.

DQN parameters	
gamma (discount rate)	0.99
epsilon	0
tau	0.001
replay buffer size	10000
batch size	64
model update frequency	4
multi layer perceptron layers	200, 150

learning rate	0.0005

Double DQN

Based on this paper.

The double DQN involves a slight variation in the training step. To calculate the target action values, we use one model (the trained model) to select the on-policy actions that would be performed in the next state and another (the target model) to evaluate the value of the action-state.

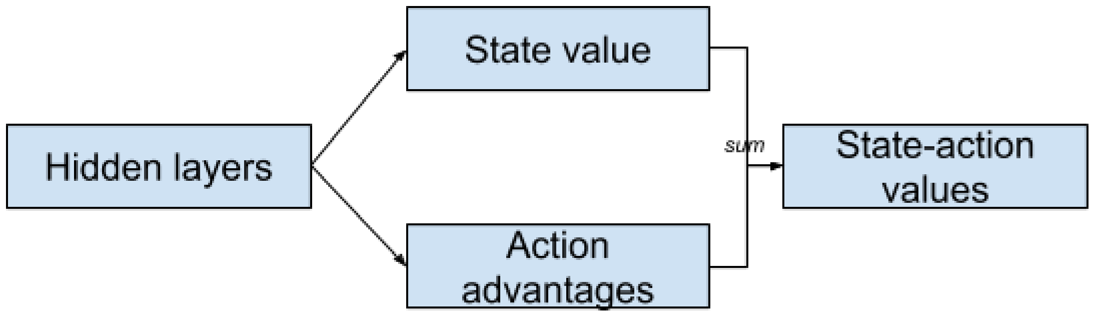
next_action = argmax(trained_model.predict_action_values(next_state))
*action_value_target = reward + discount_rate * target_model.predict_value(next_state, next_action)*

This addresses the overestimation bias created in the DQN architecture by systematically selecting the maximum value out of all action estimates when the estimations are themselves noisy and inaccurate.

Dueling DQN

Based on this paper.

The dueling DQN relies on a different model architecture to estimate the value function, effectively forcing it to distinguish between a state value and an action advantage value which are added to get the state action estimations.



A subtlety of this architecture is the need to subtract from the action advantages value their own mean. By centering them on zero, the network is forced to estimate the delta of each action over the state value.

action_advantages = f(hidden_layers)
action_advantages = action_advantages - mean(action_advantages)

The dueling DQN forces more resources of the network to be dedicated to estimating the state value function which in many cases will matter more than the specific of each action.

Prioritized replay

Based on this paper.

Prioritized replay departs from the standard uniform sampling policy by introducing sampling weights based on the temporal difference at the last training step. At each training step, sampling weights are updated in the replay buffer. To ensure that new samples get prioritized, they are added to the buffer with the maximum priority value.

In order to tune the sampling behavior, a prioritization exponent is applied to the temporal difference to get the weights. An exponent of 0 is equivalent to no prioritization while 1 gives the full prioritization behavior. A minimum prioritization is added to the temporal differences to prevent samples with a zero temporal differences from never being selected again.

*sampling_weights = (temporal_differences + min_prioritization) ** prioritization_exponent*
sampling_probabilities = sampling_weights / sum(sampling_weights)

Prioritization sampling introduces a bias as the sampling doesn't reproduce anymore the expectation distribution. To correct for this bias, an importance sampling correcting weight is introduced to compute the loss.

*importance_sampling_weights = 1 / (buffer_size * sampling_probabilities) ** importance_sampling_exponent*

Prioritized replay better leverages the replay buffer by focusing the training on the samples that have the biggest potential impact on the policy.

Following the algorithm in the original paper, we implement prioritized replay in conjunction with double DQN.

Prioritized replay parameters	
prioritization exponent	0.5
prioritization importance sampling exponent	0.4 -> 1 (linearly annealed)
minimum prioritization	0.01

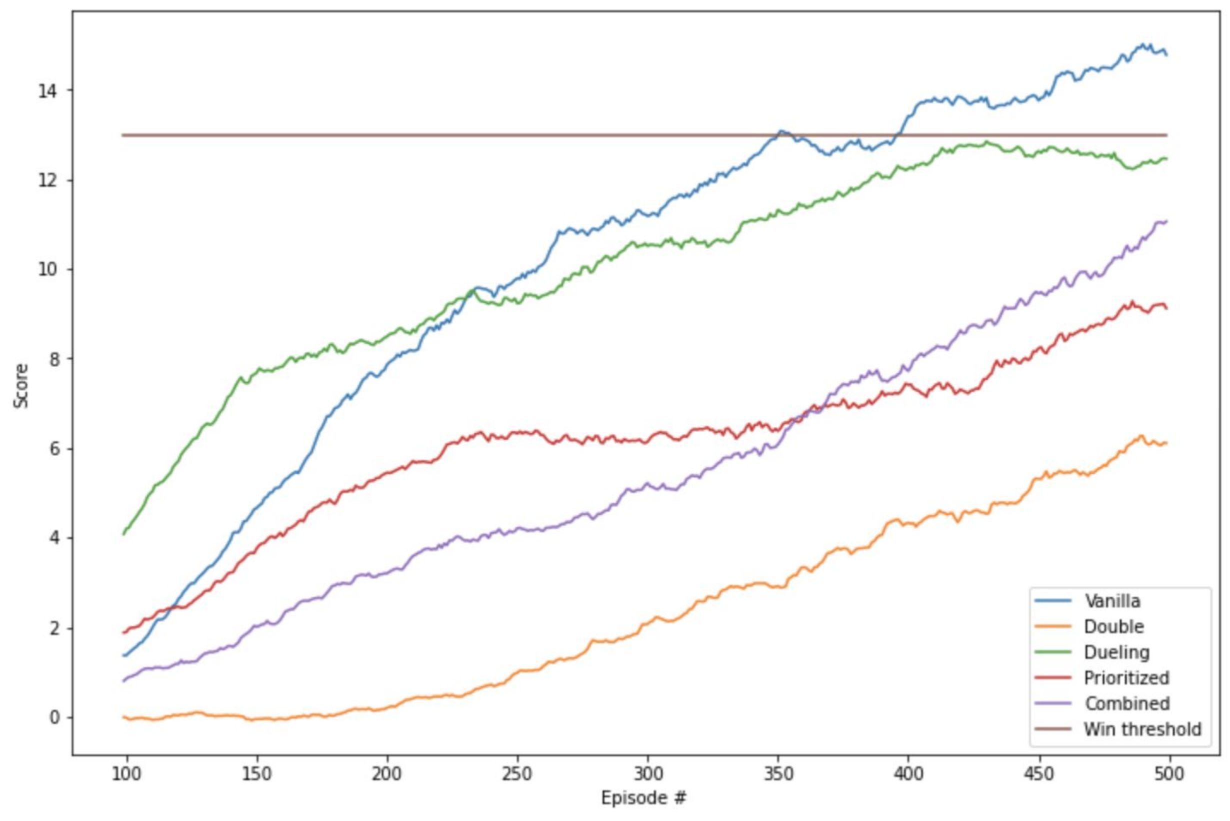
Combined approach

The last agent we test implements all of the above variations.

Results

We train all the models with the same set of parameters and over 500 episodes. No hyperparameter optimization was performed and most parameters were chosen based on examples found in the literature (the rainbow paper being the main source). The comparison is by no means rigorous as the environment is stochastic and can yield results with significant variations over different runs. A more robust approach would be to use multiple runs for each model and average their respective results. This was not done because:

- running that many runs over a single machine would be prohibitively long
- the environment provided for linux machines without visualization crashed, limiting our ability to parallelize the work in a cloud setting



The simplest DQN performs better, and solves the environment in ~440 episodes. We make the hypothesis that because the environment is quite simple, more straightforward approaches might do better. It's also possible that the ranking is more a reflection of the variability of the agent / environment system than actual performance of the various algorithms.

Future efforts

Our current implementation suffers from several limitations that could be addressed in future work:

- Hyperparameter optimization through grid search
- Better performance by implementing the full agent as a tensorflow graph (currently only the model is implemented in tensorflow)
- Additional refinements to the DQN architecture
    - Noisy DQN
    - Multi-step learning
    - Distributional DQN