MA2823 - Project final report

CentraleSupelec

Benoît Sepe
Centrale Paris
benoit.sepe@student.ecp.fr

Badr Youbi Idrissi
Centrale Paris
badr.youbi-idrissi@student.ecp.fr

Alexandre Huet
Centrale Paris
alexandre.huet@student.ecp.fr

ABSTRACT

Machine learning as a whole has taken a turn towards raw input based learning. State of the art algorithms now rarely use ad hoc hand crafted features. Instead there has been a surge of end to end solutions. This trend coincides with massive increase in computational power and its availability to individuals through the cloud. Models became more and more complex with clever regularization techniques to prevent overfitting, proper initialization, carefully chosen activation functions, and efficient optimization techniques. This especially holds true for deep learning in computer vision. We've seen huge progress in image classification accuracy thanks to Convolutional Neural Networks. Reinforcement learning is no exception. In this project we implement a deep reinforcement learning algorithm known as Deep Q Learning that learns to play a "simple" 2D platformer know as the Chrome Trex Runner. This easter egg game that pops up when the chrome browser is offline is a fun and mindless one at first sight but nonetheless a challenging one for an artificial intelligence for reasons that we'll explain later. Throughout this report we will go over the challenges faced, the choices we made, the implementation details, the results we arrived to and what are some possible improvements. Given the time and ressources we have, the results we arrived to are quite satisfying.

lastpage

1 INTRODUCTION

For our machine learning project, we came up with the following problem : **Given** a "simple" 2D platformer game, **Use** Deep reinforcement learning (Deep Q Learning) **To** maximise the score. We then had to choose the game we would work on : it had to be simple enough for us to have a chance to make it work, but complex enough to be interesting.

We chose the Chrome T-Rex Runner Easter egg, a hidden game in Google Chrome, where you play a dinosaur jumping over cactus and birds to get as far as possible. The speed of the dinosaur increases as you go further and further in the game. We forked a pygame implementation and adapted it to our needs (See the Chrome T-rex Gym Environment Annex 7.1)

We chose to implement and compare two algorithms:

- A **deep** Q learning algorithm, that takes the raw pixels of the game as input and outputs an estimation of the value function Q. The algorithm was first introduced in 2013 (by mnih et al) and later improved significantly in 2015;
- A Q learning algorithm, that takes the distance to the next object as input and outputs a table with the reward associated to each action.

The application to this game may seem like a scholar exercise, but our example could be expanded to real world usage : our algorithm does image recognition, and therefore is capable of learning from images received from a camera for example.

2 PROBLEM DEFINITION

2.1 Markov Decision Process

"A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker." [5]

More formally it is a 4 tuple (S, A, P_a, R_a) where

- $\mathcal{S}$ is a finite set of states,

MA2823, January 2019, CentraleSupelec, France
2019. ACM ISBN 123-4567-24-567/08/06…$0
https://doi.org/10.475/123_4

- $\mathcal{A}$ is a finite set of actions
- $P_a(s, s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a

One can also define a policy π as a function from $\mathcal{S}$ to $\mathcal{A}$. It is a policy in that it outputs an action given the current state.

In this setting, a transition is defined as a 4 tuple (s, a, r, s') where s is a state, a an action, r the reward resulting from doing action a in state s and s' the state that follows s after doing action a.

2.1.1 Our State space. In our case the state space is the set of possible input images. However there is a problem with this definition. We cannot determine in which state we are with only a single image since multiple scenarios could lead to the same image. For example if the dinosaur is midair you could be either jumping or falling back down. So to properlly represent a state we need a sequence of images. The original article defining Deep Q Learning chose a sequence of 4 images as a good representation of a "state".

2.2 Value function – Q function

In a Markov decision process we define the value function under policy π as the expected discounted future reward from a state s onward after doing an action a.

$$Q_\pi(s, a) = \mathbb{E}\left[\sum_{t=t_0}^{T} \gamma^{(t-t_0)} r_t\right]$$

Where $r_t = R_{a_{t-1}}(s_{t-1}, s_t)$ and $a_t = \pi(s_t)$

The value function or Q function is then defined as

$$Q^*(s, a) = \max_\pi Q_\pi$$

The aim of reinforcement learning is to find this function since if we have this function we can very easily deduce the optimal policy as follows

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s, a)$$

2.3 Bellman equation

The value function satisfies what we call the bellman equation. This equation states that for a state s and an action a that leads to state s' with reward r we have

$$Q^*(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right]$$

This simply says that if we knew the function Q^* for the state s' and all actions then we simply have to perform the action that maximises $Q^*(s', .)$ and add it as a discounted reward to obtain $Q^*(s, a)$

Most reinforcement learning algorithms use this iteratively to converge to Q^*. Indeed if we construct a table that represents the Q function and we intialize it to 0 then update it with the following :

$$Q^{i+1}(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^i(s', a') \mid s, a\right]$$

When $i \longrightarrow \infty$ it has been proven that $Q^i \longrightarrow Q^*$

2.4 Deep Q learning

In many cases (Which happen to be the most interesting ones) the state space is infinite or at least way too large to compute a Q-table for it. For example all control cases like robot locomotion and stability the state space is continuous and hence very large. In our case the state space has also a very high dimensionality since we take as input the raw pixels. Then how can we still do Q-learning? Well instead of constructing a unreasonably large table we could use an *estimator* $Q(s, a, \theta) \approx Q^*(s, a)$ that we would then update iteratively to converge to Q^*. For simple tasks, people have made linear estimators that work very well. The basic idea of deep Q learning is having a deep neural network (That we refer to as the Q-network) instead that estimates Q^*. This idea seems simple enough but making the estimator a neural network introduces a lot of complications that we will go into later.

2.5 How to update the parameters

We have introduced the idea of having a parametric estimator of Q^* but we still don't know how to update these parameters to converge to Q^*. Here we use the same basic idea as Q-learning with a table by trying to get closer to the *target* $y_i = \mathbb{E}\left[r + \gamma \max_{a'} Q(s', a', \theta_{i-1}) \mid s, a\right]$ And to do so we define the following loss function

$$\mathbb{L}_i(\theta_i) = \mathbb{E}\left[(y_i - Q_i(s, a, \theta_i))^2\right]$$

Two things to be noticed here : the loss function changes after every update of the parameters because the *target* is also a function of θ. This introduces one of the biggest challenges of Deep Q learning which is the unstationary character of this optimisation problem. In supervised learning gradient descent leads us to a local optimum of the loss function. The challenge here is that everything changes as we change our parameters and the local optima shift around in the parameter space. The second thing to be noticed is how the target depends on the previous θ. This is a deliberate choice that was made. Indeed we could've had the y depend on θ and update the parameters as the simulation data came in. This is called online learning. It seems to be the most natural choice for this kind of learning where we make an action and observe the state and reward that result from it in a loop all while learning at each iteration. Unfortunately this turns out not to work very well because the optimisation diverges catastrophically. Instead we will learn offline. This means that for every iteration we will gather simulation data for a specified amount of time and only update the parameters at the end of that iteration. This method has several benefits : it first and foremost makes the problem a lot more stable with respect to optimisation because we average the consecutive updates and hence make the shifting target more tame. It also makes our problem much closer to supervised learning. That is because in supervised learning, all algorithms suppose that the data has a fixed underlying distribution. Which of course is far from being the case here. Since we made our problem more similar to supervised learning, we can then use gradient descent to minimise the loss function.

2.6 Experience Replay

Here we discuss another major challenge of getting Deep Q Learning to work. This issue is known as catastrophic interference (or catastrophic forgetting). It is a negative trait of neural networks that makes them able to completely and suddenly forget something they learned before while learning new data. We humans also forget as time passes by, however our memory is progressive and makes older memories fade away slowly overtime if they are not used. Important memories however stay mostly intact. Neural networks don't seem to have the notion of importance for "memories" (which indeed is hard to define). Hence they may sometimes suddenly forget something (that could be important) they had previously learned. This is something that has to be addressed in Deep Q Learning. And it is dealt with by using experience replay. Experience replay is storing a large number of transitions (4 tuples (s, a, r, s')) in an array (that we refer to as memory) and sampling a mini batch from it to learn at each iteration. In the previous section we talked about how the loss function shifts from update to update and how offline learning makes this closer to supervised learning. One other issue that makes Deep Q Learning challenging with our current understanding of Deep Learning (Which comes mostly from supervised learning) is that our data is not Independent and Identically Distributed (iid). It is on the contrary extremely correlated. For example when simulating a game, a state s and and the state after it are nearly identical apart from some pixels that correspond to how the action affected the game right away. Experience replay tackles both the issue of catastrophic forgetting and the correlation between states. And this is done with the random sampling from the memory array. This is because by sampling randomly we take transitions that are more likely further apart and hence that are no longer correlated. By randomly sampling we also take older transitions and we therefore make our neural network more robust to catastrophic forgetting by relearning older transitions.

3 RELATED WORK

This project is for the most part an implementation of Deep Q Learning that was first introduced in 2013 by mnih et al in their paper *Playing Atari with Deep Reinforcement Learning* [3] and later improved in 2015 [4]. These papers were the first to introduce a successful deep learning approach to reinforcement learning. The first article to use neural networks for reinforcement learning is TD-Gammon which used a Multi Layer Perceptron with a single hidden layer with sigmoid activation as estimator and achieved near top human level performance in Backgammon at its time (1996). However when applied to other games, it didn't work at all. Most other articles were about making linear approximators with better convergence guarantees. In a completely different setting, there were interesting articles that proposed evolutionary algorithms that came up with neural networks to control agents. One of these algorithms is HyperNEAT which stands for Hyper Neuroevolution of Augmenting Topologies. This genetic algorithm mutates and crosses over neural networks in a "population" to evolve them as natural selection would. This algorithm is interesting in the sense that it also changes the structure of these neural networks and makes them more and more complex until they are complex enough to accomplish the task.

4 METHODOLOGY

This was quite a daunting project to take up at first since we didn't really have the basics of reinforcement learning so the first step was to read many articles to get a feeling for the main algorithms used in the field. The bonus lecture given in our Machine Learning course helped us a lot to nail down some important implementation details of the basic Q Learning algorithm and introduced us to the Open AI gym environments which were a very helpful framework to guide us in changing the chrome dinosaur's environment.

After this substantial step, we started the actual implementation of the project by making a gym environment out of an existing implementation of the chrome trex game. See Annex 7.1.

In the following subsections we will present the different implementation details of the two algorithms we used to tackle this problem.

4.1 The Environement

The gym environment has a step function that advances the game a single frame and does an action. In our case the action is either 0 : Nothing 1 : Jump 2 : Duck. This step function returns the next observation, reward and whether or not the game is finnished.

In the case of the Deep Q Learning algorithm, the environement also preprocesses the input image in order to make it smaller and hence making training less costly computationally speaking. The original image is 600 pixels wide by 150 pixels tall. We first convert the image to black and white then resize it to 100 by 100 and stack the 4 most recent transformed images one on top of the other. We therefore end up with an input array of size (100,100,4).

4.2 Q learning structure

Like in every Q learning algorithm we have a Qtable, the matrix containing for each state and action Qvalues. Qvalue is a score, computed thanks to rewards getting during the game. Classically the Qvalue in Q learning is computing with the Bellman equation. In our case we will create another equation to match our problem.

For us a state is the distances to each cacti on the screen. Theses distances are stored in Indices table – it's the index of all our matrices. For each state, there is two actions : jump or not (1/0). During the game we can drop reward if the Dino pass a cacti (positive reward) or if it died (negative reward). If a reward drop, we compute, thanks to our equation, the Qvalue of all (actions, state) pair encountered from the beginning of the current game. The more an (action, state) is old, the less is the reward.

We train the algorithm with a trade-off exploration / exploitation. Exploration is choosing action at random and exploitation is choosing the best known action – meaning the action with the higher Qvalue. Finally, after many training games, we can test our algorithm with choosing for each state encountered the action with the higher Qvalue.

4.3 Specificities of our Q learning

The Qtable is not buildable at the beginning because we don't know all the possible states. Hence we need to feed the Qtable every time we see a new state. During tests this leads to issues because if we don't have seen the state before, the algorithm can't decide and we need to take it at random. So we need a lot of trains to ensure that it will not happen to many times. In Deep RL states are images of the game. Here it's impossible because it will create a to huge Qtable. That's a reason why DRL is better than QL.

One advantage of this kind of algorithm is that we can stop it and relaunch the training in the same state it was left. It's because we save the Qtable in a file and when we relaunch the algorithm we only need to reload the Qtable to have all learned information of former training and train on them.

4.4 Issues of Q learning

Unknown states. Having a Qtable without all possibles states at the beginning is an issue. If, during tests, a encountered state which is not in the Qtable, we have to randomly draw an action. If on 3 tests we have this issue and in 3 others no problem, that will distort the possible comparison. So we need a lot of tests to avoid it to compare results. For example with 1000 training games we have 8 unknown states during the 100 tests. We can improve that, changing random by the average of similar states which will be closer of the reality. That is another advantage for DRL because we can predict non seen state.

Pteras. If we reach a certain score, it's possible to meet pteras - an other obstacle for the Dino. This is not already implemented - we only take care of cactus. So our algorithm can't go over 150, 200 hi score. And it leads our algorithm to not learn the good behaviour. We could implement pteras as we have done for cactus. It will not be really interesting, except that it leads our Dino to go further in the real game. That is why, for our tests we disable pteras in the game.

Epsilon. In a classic Q learning algorithm, at the end of each episode of train we decrease the epsilon value to give more probability to have exploitation than exploration. But here it's not possible because all games are difference due to the procedural generation. So we need to have one epsilon for each encountered state. If not we will learn a lot of the firsts steps of the game and then we will pass only on exploitation policy even if we doesn't encountered states of next part of the game.

Jumping state. We need to prohibit actions during Dino's jumps. If we allow its, it will do actions with any influence of the game but the algorithm will learn about them.

Gamma fixed. The larger the gamma, the smaller the discount. This means the learning agent cares more about the long term reward. If the game go far, former jumps will have more impact on future jumps because of the density of cactus - less space to jump. So it will be useful to own a variable gamma which be function of the density of cactus - to take care more of former actions if there is more obstacles.

Asymmetry. Choosing to not jump imply that the next step will require another choice at the opposite of jumping that imply many frames without any choice. That's why the Dino jump sometimes for nothing. But there is no influence on the algorithm. That's why the equation replacing Bellman once : qvalue = qvalue + reward * (gamma**(delta time-line)), take the time-line variable – it's a clock starting at the game beginning. The reward can't be function of a list of actions (due to this asymmetry) but func

Cactus have variable sizes. We only get rect.x information that is the abscissa of the centre of the cactus. But bigger cactus has to be considered differently from the little - that isn't the same hit box. So to complete our model we need to change the definition of the state. It will not be only the distance to cactus but also the sizes of cactus. This is another parameter in the Qtable meaning a bigger table. We saw another time the tradeoff between memory, learning time and efficiency of our algorithm.

4.5 Fixing hyper-parameters

In classic Q learning algorithm, the two parameters are gamma and the learning rate. In our case we only need gamma. Another parameter is the relation between the two rewards : death and pass cacti. Here the relation : death = - pass cactus * coefficient. This coefficient is really important for a balanced system. We know that we need gamma close to 1 because there is a lot of actions during few times. So we need it decrease rewards not to fast to consider all interesting former actions. We tried with several possibilities and we found gamma = 0,99 and dead reward = -1.

4.6 Deep Q Learning Structure

In this section we do a general overview of the algorithm's structure, using all the notions we introduced in Section 2. The algorithm is the following :

```
Initialize replay memory;
Initialize Convolutional Neural Network with
  random weights;
for episode = 1 to number of episodes do
    Initialize first state;
    while Episode not done do
        Choose action using epsilon greedy policy;
        Execute action;
        Build transition tuple;
        Add it to memory;
        Sample a minibatch of transitions from
          memory;
        Predict Q value on the whole batch using
          Neural Network ;
        Calculate the loss ;
        Do a single step of gradient descent ;
    end
end
```

Algorithm 1: Deep Q Learning

4.6.1 Memory. What we refer to as memory is basically a double ended queue that we add transitions to until it reaches a maximum capacity and then we begin popping the oldest transitions. Here we used the dequeu object provided by python that is optimised for removing elements (Unlike the normal python list which has O(n) cost for popping an element).

We initialize the memory by doing random actions until we have enough transitions to be able to sample a minibatch from memory.

4.6.2 Convolutional Neural Network. CNNs are famous for their breakthroughs in Computer Vision. It seems natural to use them here since we go from the raw pixel space. We won't go here into how CNNs work since it would be too long in this report.

It basically takes in the "state" (which is simply 4 stacked consecutive images) and outputs a vector with size the number of possible actions. The element i in the output vector corresponds to the estimated Q value at the input state with action a_i

For example if we feed the CNN s_{input} then the output is going to be a vector of size 3 (since there are 3 possible actions) and each element with index i is going to be the estimated $Q(s_{\text{input}}, a_i)$. So what the CNN does is it estimates the Q value vector for all possible actions.

Here is the structure of the CNN used in this algorithm.

```
_________________________________________
Layer      Output Shape         Param #
=========================================
InputLayer (None, 100, 100, 4)  0
_________________________________________
Conv2D 1   (None, 24, 24, 16)   4112
_________________________________________
Conv2D 2   (None, 9, 9, 32)     32800
_________________________________________
Flatten    (None, 2592)         0
_________________________________________
Dense      (None, 256)          663808
_________________________________________
Dense      (None, 3)            771
=========================================
```

```
_________________________________________
Layer      Characteristics
=========================================
Conv2D 1   16 8*8 filters with stride 4
_________________________________________
Conv2D 2   32 4*4 filters with stride 2
=========================================
```

We used the famous Deep Learning framework Keras and its functional API to create this model. Keras let's you define a graph (See figures in the end of the report) to build your deep learning model.

4.7 Epsilon Greedy Strategy

One important question to ask ourselves in the training phase is what action should we do in each iteration? This happens to be one of the most important decisions in reinforcement learning since it is basically the trade-off between exploration and exploitation. In other words, should we choose actions randomly to explore the state space or should we choose the action that maximises our estimated Q value and hence exploiting what we have already learned. It turns out one should do a mix of both. We implemented what is known as an Annealed Epsilon Greedy strategy. This simply chooses a random action with probability epsilon at each iteration. In the other case it chooses the action that maximizes the Q value. The Annealed part stands for an epsilon that starts at 1 (We always choose a random action) and then decreases slowly over the course of training (We choose less and less random actions) since we have learned enough about the environment to navigate it.

4.8 Learning phase

In the learning phase of the iteration we calculate the loss and do gradient descent on it. However the loss contains only $Q(s, a)$ and not the whole vector of Q values. So to calculate this loss we built another Keras model that goes on top of the basic one. (See figure in the end of report) This keras model takes an additional input which corresponds to the action as a one hot encoding and does a dot production with the output of the base model to select only the appropriate Q value. The loss then becomes simply the squared error of the target y and the output of this last model.

4.9 Hyper parameter tuning

The challenge in Deep Reinforcement Learning is that there is no robust way to evaluate the performance of the model while training. That is because we always have a certain probability to choose a random action that in most cases ends up killing the dinosaur. So the current score while training isn't a very good indicator. The loss's absolute value doesn't mean anything since the loss function changes overtime and hence we cannot compare different loss values of different models to judge how good they are. There is also no way to do cross validation since there is not training and test sets etc... This means that the hyper parameter tuning is mostly done with a feel for how the training is going by watching it.

5 EVALUATION

5.1 Q learning results

Without birds (called pteras) and with the best hyper-parameters founded, we ran 1000 trains. After that we ran 100 test. The score average is 115. So 1h of training give the algorithm able to run not very far. We conclude that it's working but we need huge time to train. Especially, a major part of death are due to unknown states – so more trains – and not bad actions. That is why DRL is so powerful : it reduce the training time and avoid this big issue of unknown states. ion of the date of acting.

5.2 Deep Q Learning results

We have trained our model for about 45 minutes and it learned to nearly completely avoid cactus (As you'll see in the video or in the test program with the pre-trained provided model). However after 45 minutes of training the algorithm stalled in the sense that it never went far enough to encounter the birds (pteras). So it doesn't know how to react to them. We ran 100 tests without the birds and had a max score of 1063 and an average score of 420. Which is a very good score even for humans.

6 CONCLUSION

After testing and evaluating the two algorithms, we can conclude the Deep Reinforcement Learning is way more efficient that the Q Learning, because it can be applied larger problems and therefor is able to predict the actions.

7 ANNEX

7.1 Chrome Trex Gym Environment

The changes we made consisted in removing unnecessary things such as the menu, the game over screen and such. We also replaced the game event loop by a step function that allows us to control the game frame by frame. We wrapped all of this as an open ai gym environment. Open ai's gym is a library that defines what an "environment" is and implements it as an interface [1]. The community (Mostly researchers) then implement this interface as a middleware between the reinforcement learning algorithm and the environment in order to have easily swappable environments that are controlled in the same way. These environments have an observation space, an action space, a step function that takes an action, performs it and returns the resulting observation, reward, a boolean that tells us whether this state is terminal or not and finally other info to debug the process (This info shouldn't be used for learning). They also have a reset function that returns the simulator to its initial state and returns the initial observation. There was an existing open ai gym environment for this chrome trex game that didn't meet our needs at all since it didn't provide enough flexibility and, required running an instance of chrome which isn't ideal, didn't wait for the algorithm to perform an action but just played and was also limiting the execution speed to 60 frames per second. We decided to create our own environment. To contribute to the open source community we made a separate repository for this environment on github and also uploaded it as a package to pypi under the name of chrome_trex_gym and it can easily be installed with pip [2].

REFERENCES

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *arXiv e-prints* (June 2016), arXiv:1606.01540.
[2] Badr Youbi Idrissi. 2018. Chrome Trex Gym. https://github.com/BadrYoubiIdrissi/chrome-trex-gym. (2018).
[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints* (Dec. 2013), arXiv:1312.5602.
[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. 518 (Feb. 2015), 529–533. DOI: http://dx.doi.org/10.1038/nature14236
[5] Wikipedia contributors. 2018. Markov decision process — Wikipedia, The Free Encyclopedia. https:

//en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=873296935. (2018). [Online; accessed 6-January-2019].

Figure 1: Structure of CNN

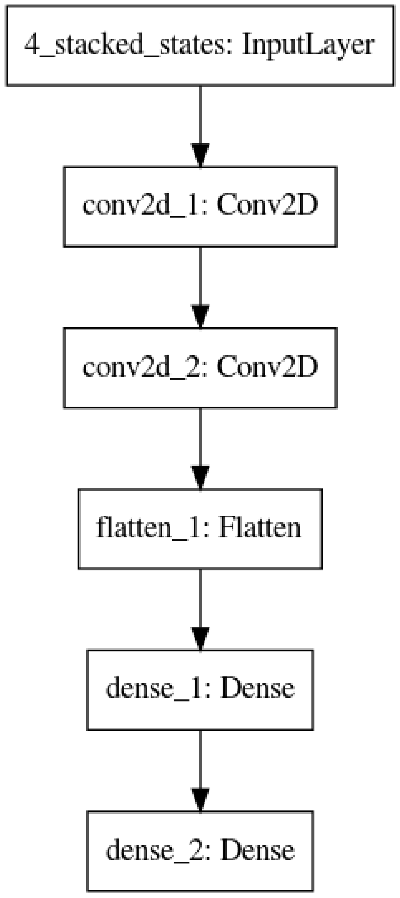
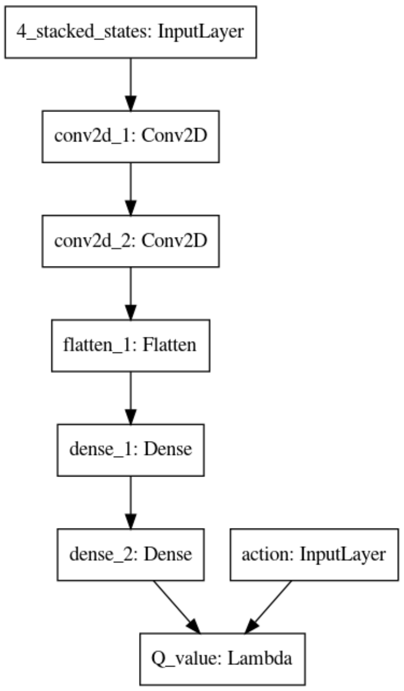


Figure 2: Structure of CNN for training


4_stacked_states: InputLayer
conv2d_1: Conv2D
conv2d_2: Conv2D
flatten_1: Flatten
dense_1: Dense
dense_2: Dense
action: InputLayer
Q_value: Lambda