55

DQN (Deep Q Networks)

Deep Q Learning

↓

algorithm that learned to play many Atari video games better than humans.

↓

raw pixel data (exactly what a human would see on screen)

↓

now a neural network operating like a function approximator

↓

pass in image of the game state, and o/p is a vector of action values, with the max value indicating the action to be taken.

↓

As reinforcement signal, fed back the change in game score at each time step.

↓

NN initialized with random values, actions taken are all over the place, but over time it begins to associate situations & sequences in the game with apt actions & learns to actually play the game well.

↓

DQN : designed to produce q value for ~~each~~ every possible action in a forward pass

(action for the state can then be chosen stochastically or greedily)

↓

architecture : 3 layered CNN (to exploit spatial relationships)

↓

(connective vector not available for images)

↓

RLU activation

(Regularized linear units)

↓

1 fully connected HL (RLU)

↓

1 fully connected o/p layer that produced vector of action values

state space for atari (210 x 160 colored)

↓

(84 x 84 monoscale)

↓

deepmind performed some minimal processing

↓

sq images allowed them to use many optimized NN operations on GPU

(56) Experience replay

↓

originally proposed to make more efficient use of observed experiences.

↓

for each time step, we get the tuple, learn from it, and then discard it, moving onto the next tuple in the following time step.

agent ⇄ environment

↓

$< S_t, A_t, R_{t+1}, S_{t+1} >$

↓

a little wasteful → we could maybe learn a little bit more, if we stored the tuples somewhere

↓

some states are pretty rare, and some actions are pretty costly

↓

nice to be able to recall such experiences

↓

replay buffer

(store each experienced tuple in this buffer as we are interacting with the environment, and then sample a small batch to learn from them)

↓

As a result, we are able to learn from individual tuples, multiple times, recall rare occurrences, and make better use of our experience.

every action A_t affects the next state S_t in some way. which can mean that a sequence of tuples can be highly co-related.

↓

A naive Q learning approach that learns from these experiences in sequential order runs the risk of being swayed by the effects of this co-relation.

↓

(randomized)
Since sampling will lead to random sequences, the sequential correlation based learning can be avoided. (avoid action values from oscillating or diverging catastrophically)

↓

example: tennis: rallying against the wall,

↓

more confident with my forehand, than my backhand. and I can hit the ball fairly straight, hence the ball keeps coming back in the same area. i.e. to the right, and hence we keep hitting forehand shots

↓

but rest of the state space is not being explored,

So I try different combination of states and actions and sometimes mistakes are made but eventually best policy is learnt

using epsilon greedy method ←

↓

for discrete space, this seems okay, but for continuous space, things may start to fall apart.

every action A_t affects the next state S_t in some way. which can mean that a sequence of tuples can be highly co-related.

↓

A naive Q learning approach that learns from these experiences in sequential order runs the risk of being swayed by the effects of this co-relation.

↓

(randomized)
Since sampling will lead to random sequences, the sequential correlation based learning can be avoided. (avoid action values from oscillating or diverging catastrophically)

↓

example: tennis: rallying against the wall,

↓

more confident with my forehand, than my backhand. and I can hit the ball fairly straight, hence the ball keeps coming back in the same area. i.e. to the right, and hence we keep hitting forehand shots

↓

but rest of the state space is not being explored, So I try different combination of states and actions and sometimes mistakes are made but eventually best policy is learnt

using epsilon greedy method ←

↓

for discrete space, this seems okay, but for continuous space, things may start to fall apart.

* The ball can come anywhere between the extreme left & extreme right

↓

descretization of the state space may lead to no learning for specific holes i.e. (not visited during exploration) margin for deviation is not much.

↓

Better to use a function approximator using RBF kernels or a Q-network that can generalise learing across the space.

↓

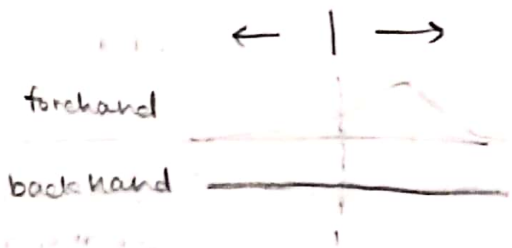
when the ball comes to the right and we successfully employ a ~~[illegible]~~ forehand, state value function changes slightly. i.e. it becomes more +ve around the exact region where the ball came, but also raises the value for the forehand shot in general across the state space.

↓

of course the effect is less pronounced away from the exact spot, but over time it can add up. and that's exactly what happens when we try to learn while playing, processing each exp. tuple in order.

↓

state correlation problem here leads to the agent playing forehand everywhere



i) stop learning while ~~playing~~ practicing

↓

this time is best spent in trying out different shots, playing randomly and exploring the state space

ii) hence it becomes imp. to memorize the experience, then later analysing what went wrong & what went right. , when I am at home and resting or I take a break

↓

advantage : more comprehensive set of examples, we can generalize patterns from across these, recalling them in whatever order we please

↓

after a round of learning, I can go back getting a round of experience, and the process is repeated.

(learning more robust policy)

↓

building a database, and learning a mapping from them, which in a way reduces the RL problem or atleast value learning portion to a supervised learning scenario.

↓

multiple passes over the same experience

and this allows us to apply machine learning techniques, models and algorithms prevalent in supervised learning literature

Fixed Q targets

↓

Q learing → form of TD learning

↓

(older value)

→ target value

$$\Delta w = \alpha \left(R + \gamma \max_a \hat{q}(s', a, w) - \hat{q}(S, A, w) \right) \nabla_w \hat{q}(s, a, w)$$

Difference : TD error

current estimate

↓

goal is to reduce the difference between these values

↓

The TD target here is supposed to be a replacement for the true value function $q_\pi(S, A)$, which is unknown to us.

↓

we defined the squared error loss function wrt $q_\pi(S, A)$, later differentiated with respect to w to get gradient descent update rule. $q_\pi(S, A)$ not dependent on approximation parameters, resulting in a simple update rule.

↓

But our TD target is dependent on these parameters, which means simply replacing the $q_\pi(S, A)$ with our TD target, ~~might~~ is ~~not be the best~~ method to approach this. (mathematically incorrect)

(*) we might get away with that in practice because the changes are pretty small and change generally occurs in the right direction.

↓

but if $\alpha = 1$, and changes being made are significant, then issues creep in.

↓

less of a concern when lookup table or dictionary is used, since (s, a) pair values are stored separately. , but here the values are correlated and change at some pair impacts nearby values as well. , as values are intrinsically tied together through function parameters.

↓

more like chasing a moving target (not optimal)

↓

better to set a short target, reach it & then change it. and repeat.

↓

decoupling the target position from the donkey's action. giving it a more stable environment and stops it from oscillating and all.

↓

(Fix the function parameters used to generate our target)

↓

$w' \leftarrow w$

↓

i.e. move towards one w' for certain timesteps and then update w' with the current w, and so on. ← changing w for next steps, then update w' with latest w and so on.

i.e. we employ

immediate return for choosing a certain action

$$\Delta w = \alpha \cdot \left(R + \gamma \max_{a} \hat{q}(s', a, w^-) - \hat{q}(S,A,w) \right) \nabla_w \hat{q}(S,A,w)$$

where w^- are the weights of a separate network that are not changed during the learning step.

x → feature vector for environment state repr

initialization methods.

* Deep Q Learning Algorithm : part 1 : sampling
↓ (building experience)

a) choose action A from state S using policy

$$\pi \leftarrow \epsilon\text{-greedy}(\hat{q}(S,A,w))$$

b) Take action A, observe reward R, next input frame x_{t+1}

c) prepare ~~next~~ next state

$$s' \leftarrow \phi((x_{t-2}, x_{t-1}, x, x_{t+1}, x_{t+2}))$$

d) Store experience Tuple (S, A, R, S') in replay memory D

$$S \leftarrow S'$$

part 2 : learn phase

a) Obtain random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from D.

b) set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, w^-)$

c) update : $\Delta w = \alpha(y_j - \hat{q}(s_j, a_j, w)) \nabla_w \hat{q}(s_j, a_j, w)$

$$w = w + \Delta w$$

d) every C steps, reset : $w^- \leftarrow w$

These tasks are not directly dependent on each other. Therefore multiple sampling ~~becks~~ steps, then one learning steps or even multiple learning steps with different random tuple batches.

↓

Other part of the algorithm is meant to support this core basis of the DQN.

↓

a) initialize empty memory buffer D (capacity : N)

(since this buffer is finite in size, we can use a circular queue, that retains the N most recent experience tuples)

b) initialize action-value function $\hat{q}$ with random weights

like the ones available in pytorch.

(network weights)

c) $w^- \leftarrow w$ (initial)

* Now for each episode & each time step, we observe a raw screen image or i/p frame x_t, which needs to be preprocessed, (to capture temporal features)

↓

Stack i/p frames to build state vector

↓

$S \leftarrow \phi(x_t)$

* memory is not cleared after each episode, this enables us to build and learn from experience across episodes.

* Improvements over DQN

(most prominent ones)

↓

a) Double DQN

b) Prioritized Experience Replay

c) Dueling DQN

think budget over estimate

↑

a kind of bias where we have +ve error (value is more than the actual value)

· Over estimation of action values that Q-learning is prone to

↓

TD target

↑

$$\Delta w = \alpha \left(R + \gamma \max_a \hat{q}(s', a, w^-) - \hat{q}(s, a, w) \right) \nabla_w \hat{q}(s, A, w)$$

TD target : Q value for the state s', and the action that results in the maximum Q-value among all possible actions from that state

↓

Choosing based on max value of action value function might not be the best strategy. as the accuracy of our Q-values depend a lot on what actions have been tried, and what neighbouring states have been explored

← since Q values are still evolving, the agent won't have gathered enough information to figure or work with best action.

↓

over estimation of Q-values. (max among set of noisy numbers)

↙

more robustness

↓

Double Q-learning

select best action as per one set of parameters w,

but evaluate using w'

↓

this is like 2 separate function approximators

agreeing upon the best possible action.

↓

select best action (from local network)

$$R + \gamma \hat{q}(s', \arg\max_a \hat{q}(s', a, w), w')$$

evaluate on basis of that action

2 value functions
↓
randomized pick of
drop.

↓

If w picks an action that is not best according to w'

Q-value returned is not that high

↓

In the long run, this prevents the algorithm from prepogating incidental high rewards that may have been obtained by chance & do not reflect long-term returns.

•) prioritized experience replay:

basic principle behind experience replay

↓

sample state tuples, store in buffer, and randomly pick sample of tuples from buffer to learn.

↓

breaks correlation b/w consecutive experiences & stabilizes the algorithm

But some experiences might be more important than others

↓

these exp. might be rare as well

↓

uniform sampling might result in very small chance of these experiences getting selected.

↓

also, since finite memory, these exp. might get lost.

hence the requirement for prioritized experience replay comes in.

↓

criteria to assign priorities to each tuple?

↓

a) TD error delta

(the bigger the error, the more we expect to learn from that tuple)

$$\delta_i = R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A, w)$$

↓

$p_i = |\delta_i|$ → store with each corresponding tuple in the replay buffer

↓

and when creating batches, we can use these values to compute sampling probability

↓

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

↓

and as tuple is picked, corresponding values can be updated using a newly updated TD error using the latest q values.

a) if TD error = 0, Probability of being picked = 0

↓

doesn't necessarily mean, we have nothing to learn from the experience.

↓

might be the case of limited learning i.e

it is possible our estimate was close due to the limited samples we visited till that point.

↓

to prevent from starvation,

~~[illegible]~~ $p_i = |\delta_i| + e$

b) greedily using these max. values might cause only certain states to be explored. (replayed over & over)

↓

resulting in a sort of overfitting on that subset.

↓

to avoid this, introduce concept of uniform sampling. (random)

↓

adds another hyper parameter A, which is used to redefine the sampling probability as,

↓

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

A can be varied to adjust the contribution of greedy & random sampling.

A = 0 : pure uniform randomness

A = 1 : greedy priority choosing

68

modified update rule:

$$\Delta w = \alpha \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^{b} \delta_i \nabla_w \hat{q}(S_t, A_t, w)$$

size of buffer

↓

b to control how much these weights affect learning.

importance sampling weight

to compensate for introduced bias.

→ more imp. towards the end, when the Q-values begin to converge

↓

b can be moved from a low value to 1, over time.

• Dueling Networks.

↓

core idea is to use 2 streams

↓

conventional architecture for this is a series of CNN layers (for feature extraction), and then fully connected dense layer to come up with probabilities.

i/p state (feature vector)

↓

get action function values (Q values)

→ 1 stream estimates state value function : v(s)

↓

and other estimates the advantage of each action

↓

* branch off with their respective fully connected layers.

finally the desired Q values obtained by combining the state & advantage values.

↓

The ~~advantage~~ values of most states don't vary a lot across actions, so it makes sense to try & directly estimate them, but we still need to capture

↓

V(s)

the difference different actions make in each state

↓

advantage function.

(open ai gym)

↓

some modifications are required to be made on the normal model architecture

↓

other extensions: a) learning from multi-step. bootstrap targets

b) Distributional DQN.

c) Noisy DQN.

↓

each of the six extensions address a different issue with the original DQN implementation

↓

The algorithm that tested performance using all the ~~these~~ six extensions together: rainbow (deepmind)

↓

outperforms individual modifications & achieves state of the art on Atari 2600 games.