2.103: Formalising the problem: States, actions, rewards

Summary

- States and actions
- Rewards
- Markov Decision Process

How can we create an AI that can play games?

Most games involve an iterated process of observing and acting

Most games involve some sort of positive or negative result

Think of a arcade game:
What are the observations?
What are the available actions?
What is the result and when is it received?

Example: breakout

The state is the current view of the game

The actions are move left, move right or stay still

The reward is 1 point each time you hit a brick or game over if you miss the ball

Formalising the game: states and actions

The player observes state *s*
The player takes action a
The player observes the next state s’

s,a → s’

This is stochastic, so we can add probability P to the transition:

P(s’ | a,s)

Rewards

There is a reward associated with the s,a,s’ construct:

s,a,s’,r

So – taking action a in state s lead to state s’ and a reward of r

We can say that the reward is a function on the (s,a,s’) construct
R(s,a,s’)

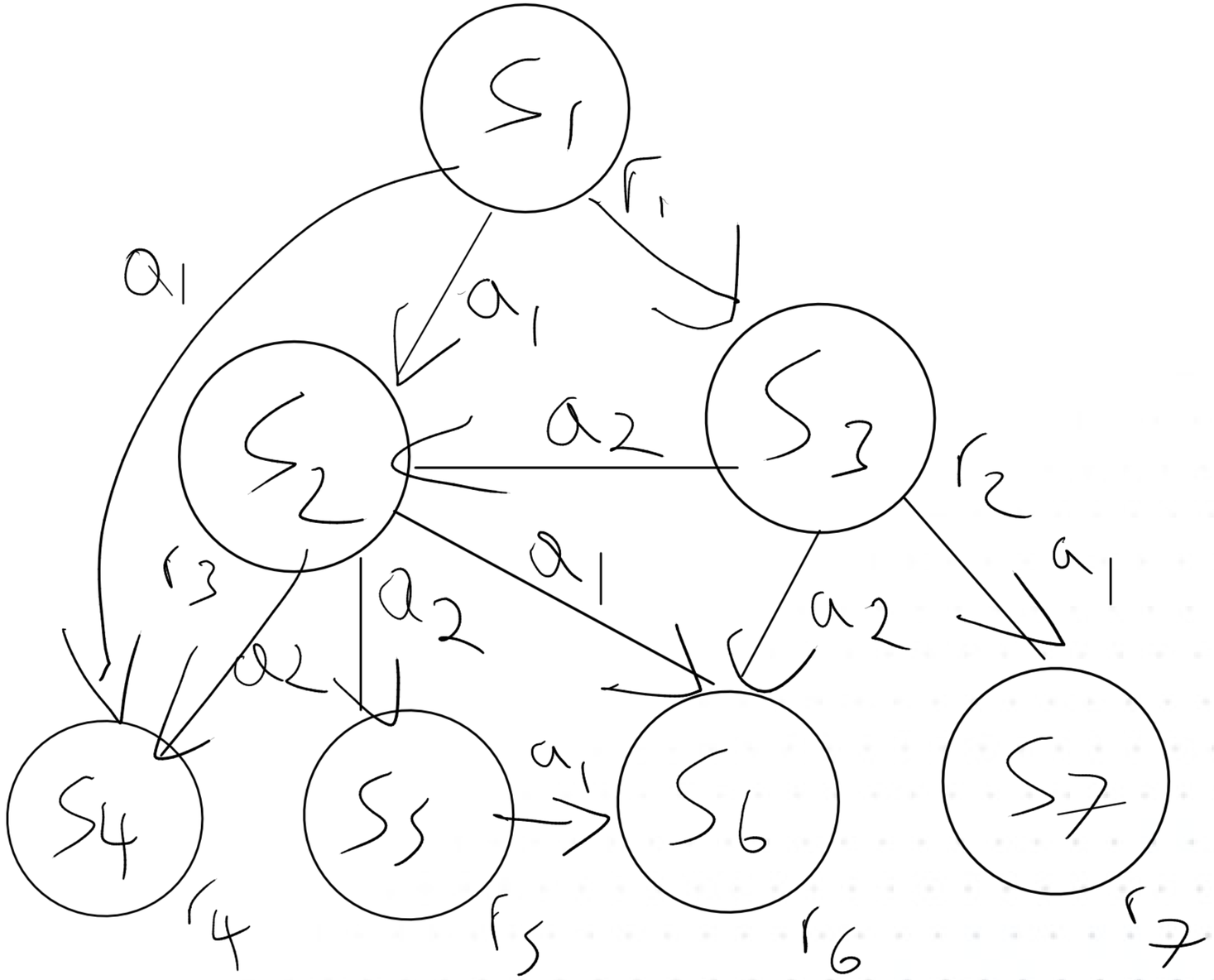
Think of breakout – when you go from a state with 7 bricks to 6 bricks, you gain a point

Markov Decision Process

We have ourselves a Markov Decision Process which is a way to formalise a stochastic sequential decision problem

$P(s' | s, a)$ state transitions

$R(s,a,s')$ reward function


S1
S2
S3
S4
S5
S6
S7
a1
a1
r1
a2
r2
a1
r3
a2
a2
a2
a1
a1
r4
r5
r6
r7

Summary

- States and actions
- Rewards
- Markov Decision Process
- Action policy
- Q-learning

2.105: Value functions and Q learning

Summary

- Define action policy
- Optimal action policy
- Future reward
- Q-learning
- Deep Q Network

How do we decide what the action is?

The **action policy** tells us what to do in a given state.

Policies are denoted ‘π’. So:

$\pi(s) \rightarrow a$

‘The policy for state s is to take action a’

What is an optimal policy?

This is a sequential decision problem, not one shot. Reward might come in the far future.

An optimal policy maximises reward over a sequence of actions

So the action you take now should unlock maximum potential rewards going into the future

Bellman equation(ish) defining the value of a given action in a given state based on future reward

$$V^*(s,a) = \max_{\pi} \sum [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi]$$

\

for states s at time t, increasingly discounted by factor gamma raised to the power of t. We did not include probability of state transitions here.

Note gamma (γ) is < 1

But… what are the future states and rewards?

The **state transition matrix** tells us how the state will change over time based on our chosen actions

The **action policy** dictates what the chosen actions will be in each state. It involves choosing the highest value action

The problem is we need to know the complete transition matrix and the associated rewards to make an action policy – not easy for Atari games!

Reminder why complete state transition matrix is not easily knowable for Atari

States are the pixels on the screen – too many possible combinations

Also, how to know what the possible combinations actually are?

And most states have zero reward

The Q-function and Q-learning

The answer to the incomplete MDP data is to approximate the value function – that approximation is the **Q function**

Q-learning involves learning the best Q-function that we can: Q*

$$Q^*(s,a) = \max_{\pi} \sum [r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | s_t = s, a_t = a, \pi]$$

Q-learning

There are various methods to do Q-learning

Pssst… most of them don’t work for real problems

The one that works here is to approximate the value function using a deep network:

Deep Q Network! (DQN)

Summary

- Define action policy
- Optimal action policy
- Future reward
- Q-learning
- Deep Q Network

2.108: DQN Agent architecture

Summary

- Architecture overview
- Replay buffer
- Epsilon greedy exploration

What is an agent?

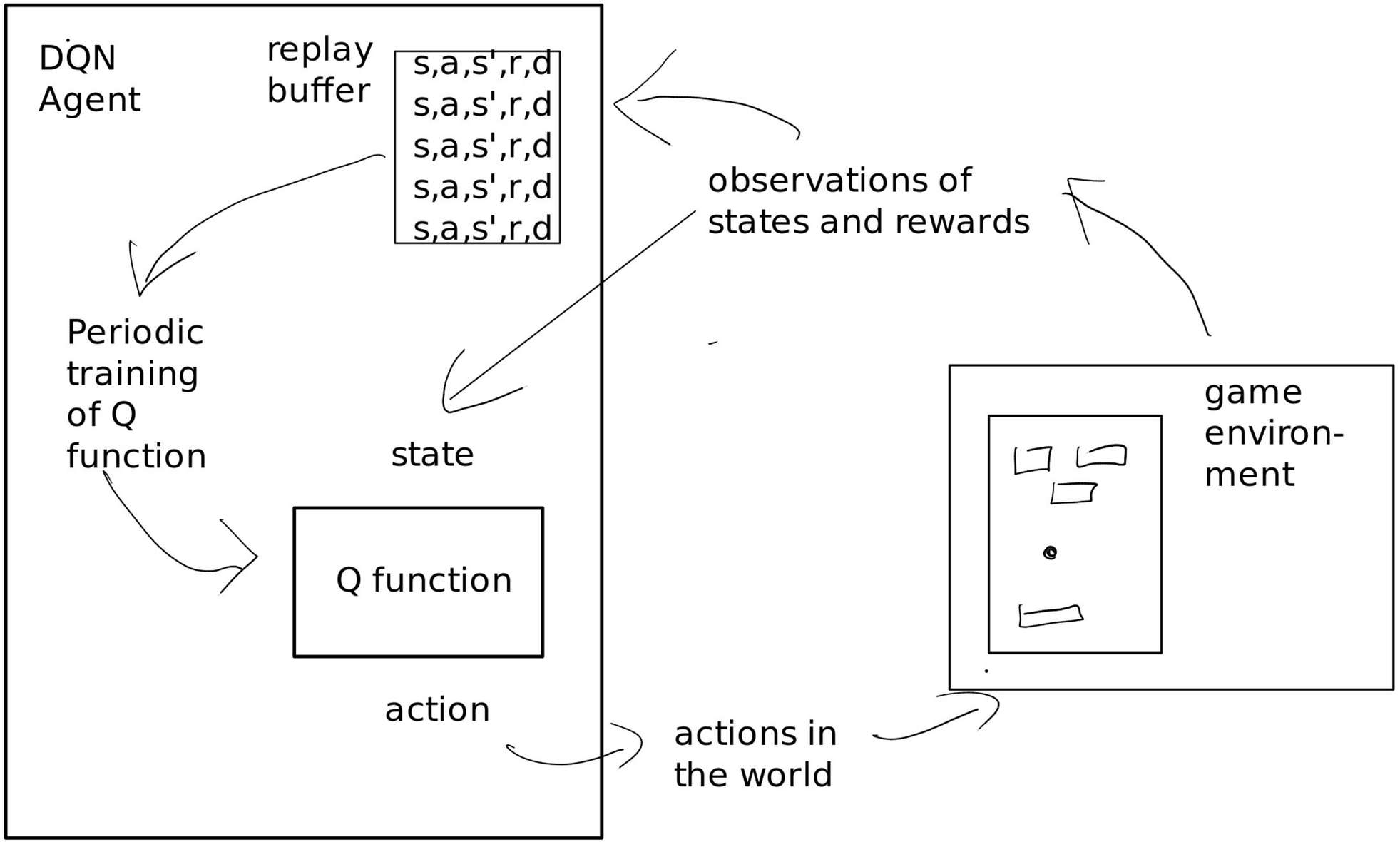
An entity that can observe and act autonomously

Agent architecture

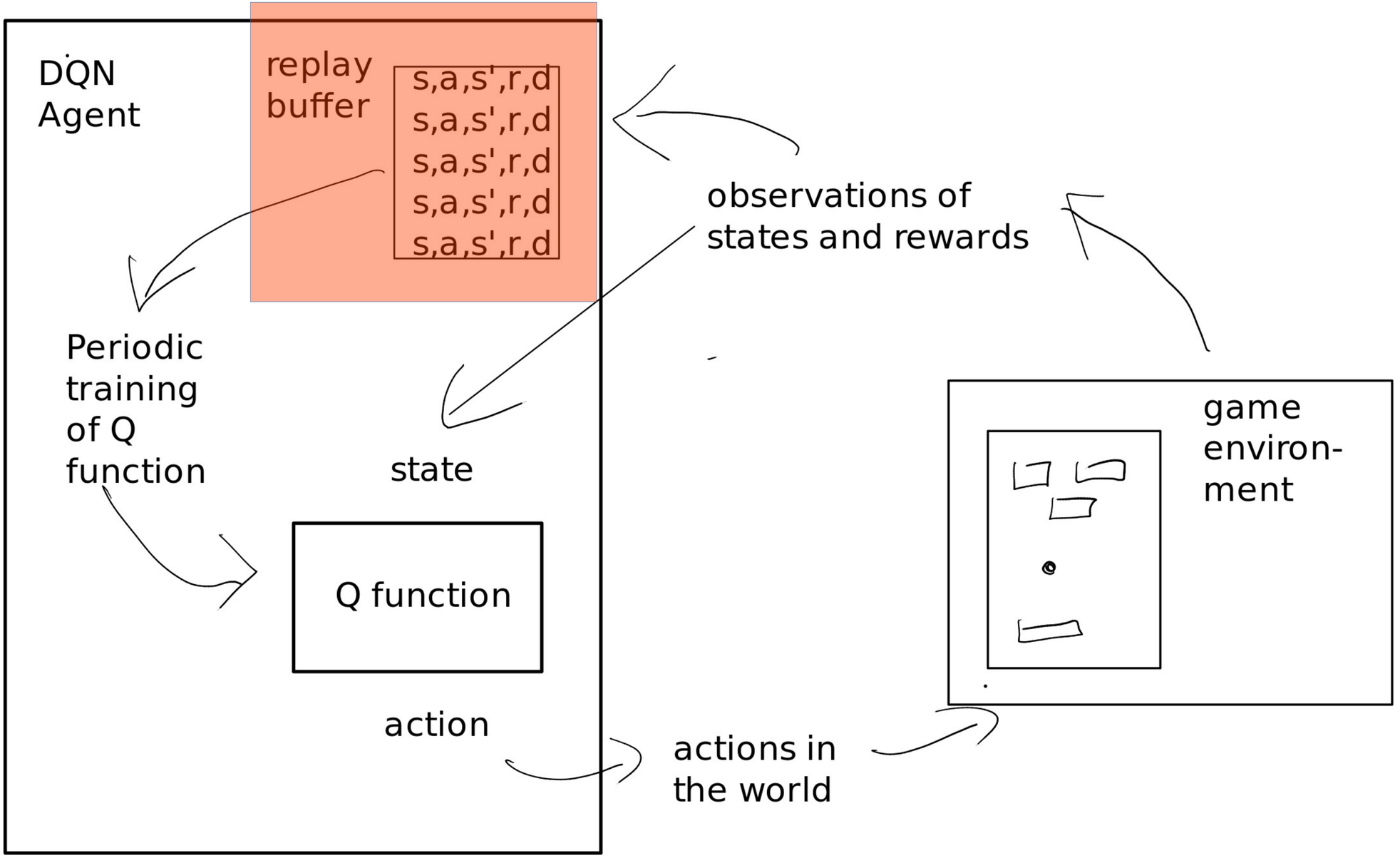
We need an agent architecture that solves two problems:

1) No state transition matrix
2) No action policy

Here is the agent architecture – we'll see the actual code implementation later



First off, the replay buffer



Replay buffer acts as the state transition matrix

AKA the training data for the Q function

How to generate the replay buffer

We **explore that game** and make observations of the form:
s,a,s’,r,done

s = state now,
a = action taken
s’ = next state
r = reward
done = true/false is the game finished?

For DQN, this is the ‘replay buffer’

Data in the replay buffer

Over time, the agent fills up a large replay buffer

Example for one state s1 and the the three actions a1,a2,a3:

s1,a1->s2,r1,d0
s1,a2→s3,r2,d0
s1,a3→s4,r3,d1

Epsilon greedy exploration

From random to using Q function

Summary

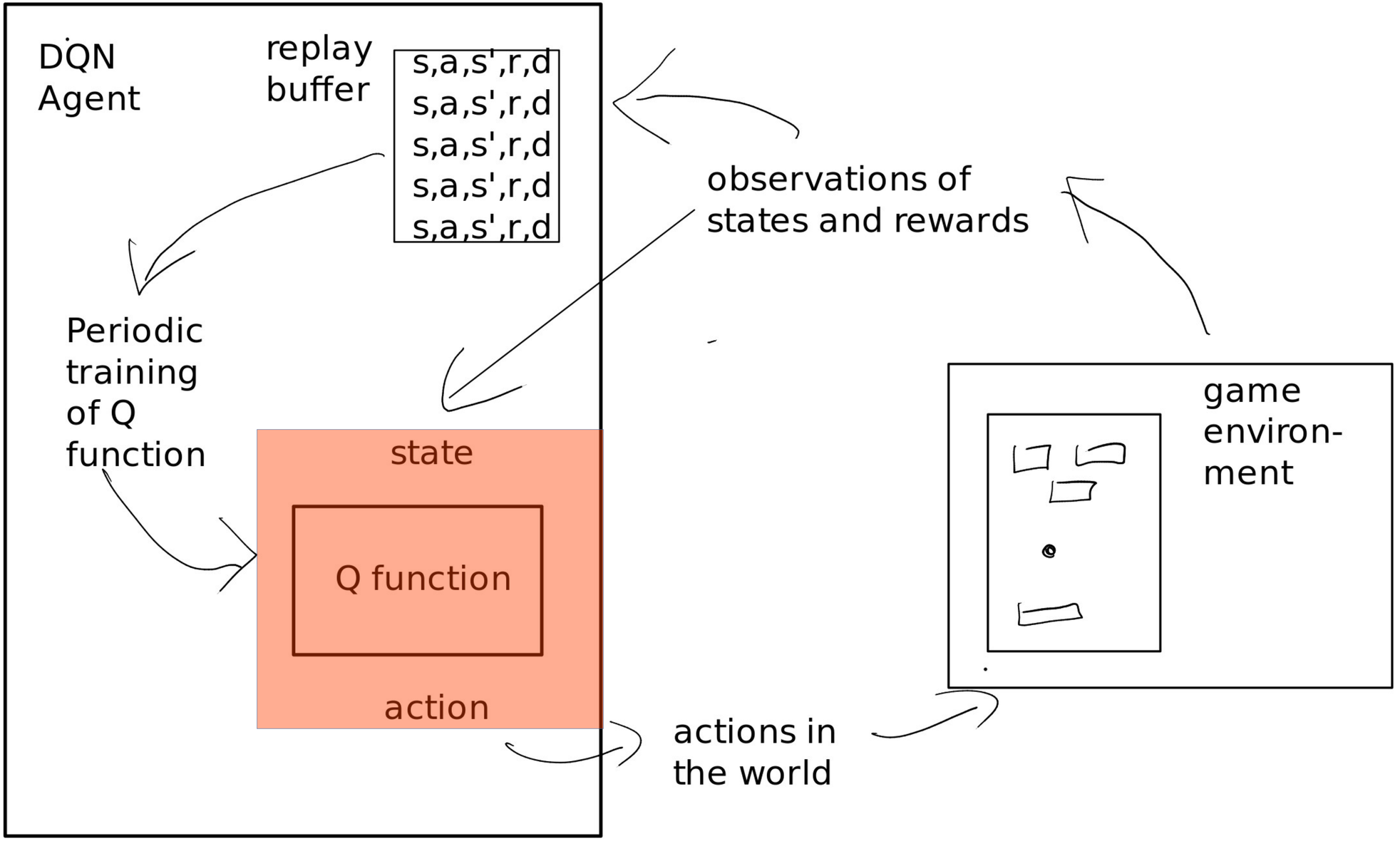
- Architecture overview
- Replay buffer
- Epsilon greedy exploration

2.110: The loss function for DQN

Summary

- Q in the architecture
- How to train a neural network
- The DQN loss function

The Q function selects actions in given states



The Q function is a neural network

We train it on the state transition data

But remember it does not infer next state

It infers the value of an action in a state taking account of future rewards

We don’t really have a ground truth of that

How do we train a neural network?

We normally have a labelled dataset with outputs and inputs.

We train on 80% and test on 20%

But here, we have the replay buffer

Before we get to that, what do normal loss functions do?

Inputs, output pairs:

input	->	correct output	network output	-	output
[1,2,3]	->	[4,5,6]	-	[4,5,5]	
[1,2,2]	->	[5,5,6]	–	[5,5,5]	
[2,1,1]	->	[6,6,6]	–	[6,1,5]	

...

Training involves feeding the inputs in, looking at the error between the desired and received outputs then propagating the error back into the weights of the network

The loss function is how you calculate the error

The answer is the loss function … which is a bit of a beast!

$$L_i(\Theta_i) = \Sigma_{s,a,r,s'}\ U(D)\left[\left(r + \gamma \max_{a'} Q(s', a'; \Theta_i^-) - Q(s, a; \Theta_i)\right)^2\right]$$

We'll get into this now, and we will see it again later in the code

The DQN loss function

$$L_i(\Theta_i) = \Sigma_{s,a,r,s'}\ U(D)\left[\left(r + \gamma \max_{a'} Q(s', a'; \Theta_i^-) - Q(s, a; \Theta_i)\right)^2\right]$$

buffer

Theta-i is the weights of an older version of the network
Theta i is the weights of the current network
We are comparing the output of our network to a combinations of the known reward (Ground truth ish) plus an estimated Value for the policy’s chosen actions across the sample

Summary

- Q in the architecture
- How to train a neural network
- The DQN loss function

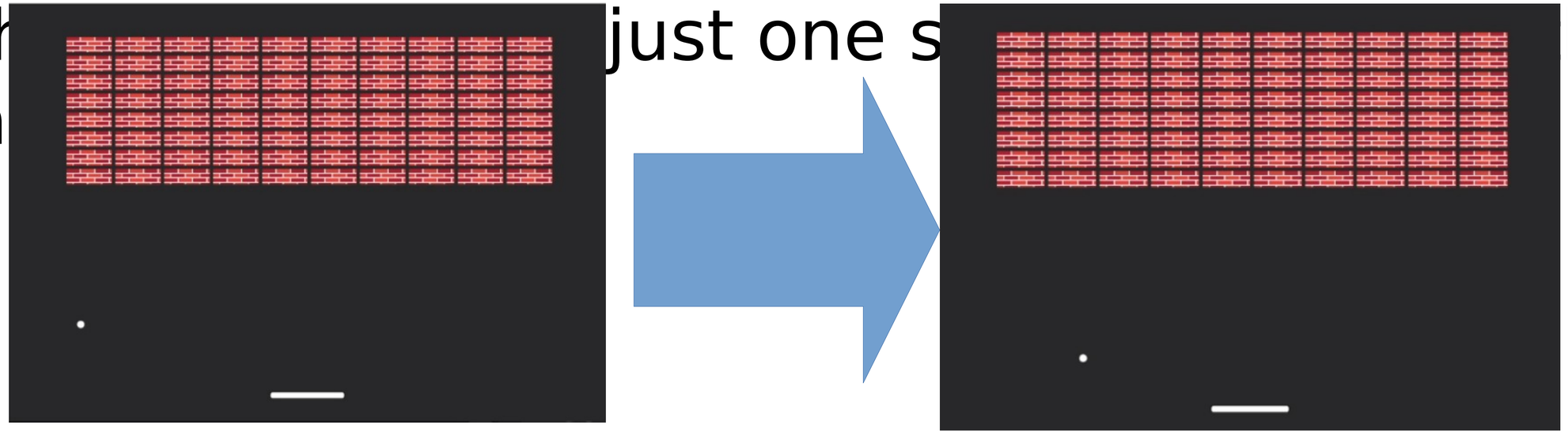
2.112: Visual processing and states in DQN

Summary

- State secret sauce
- Max colour
- Greyscale
- Resize

My observation: no sense of time!

Th just one s he
la



State secret sauce: three frames

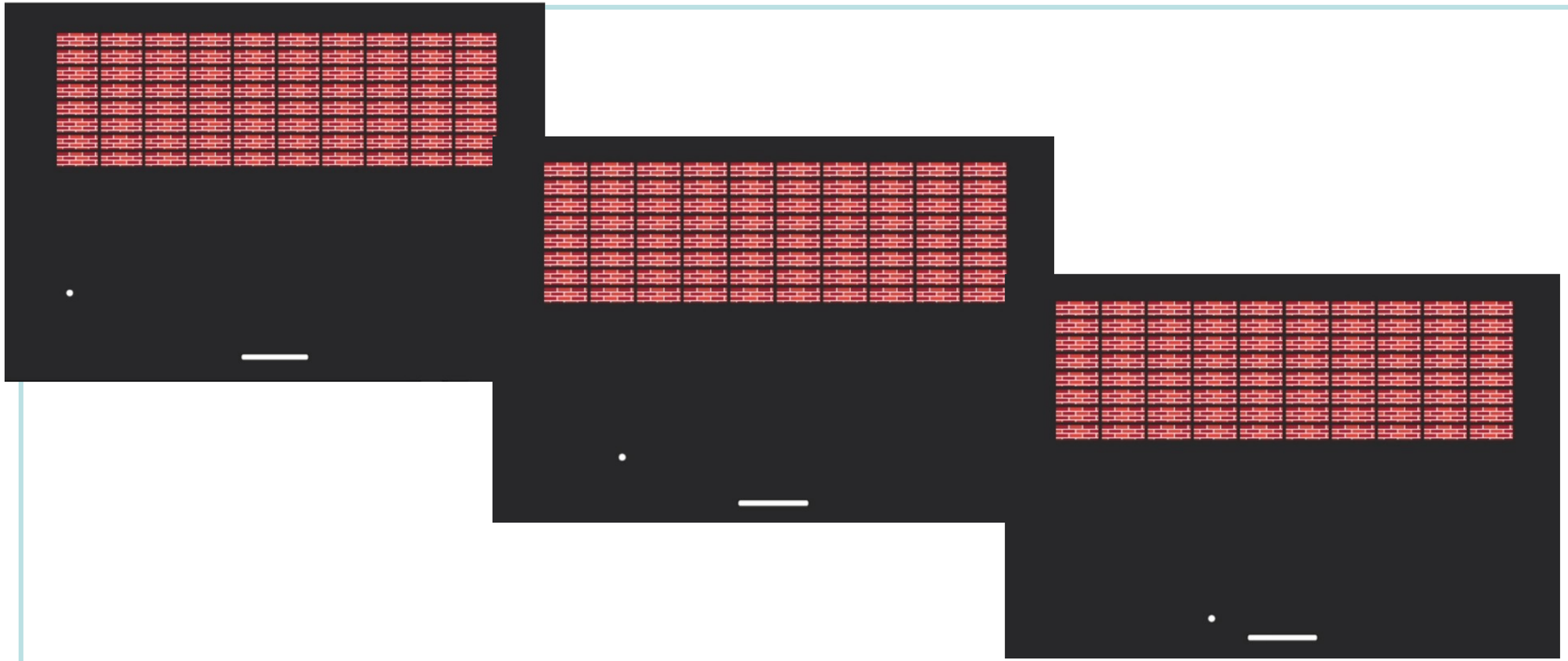


Image processing: max value

Max → fixes max sprite problem

“we take the maximum value for each pixel colour value over the frame being encoded and the previous frame. This was necessary to remove flickering that is present in games where some objects appear only in even frames while other objects appear only in odd frames, an artefact caused by the limited number of sprites Atari 2600 can display at once.” - 2015 DQN Nature paper

```
max_frame = self._obs_buffer.max(axis=0)
```

Image processing: grey scale

“we then extract the Y channel, also known as luminance, from the RGB frame”

AKA Grey scale:

(cv2 is the opencv library)

frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

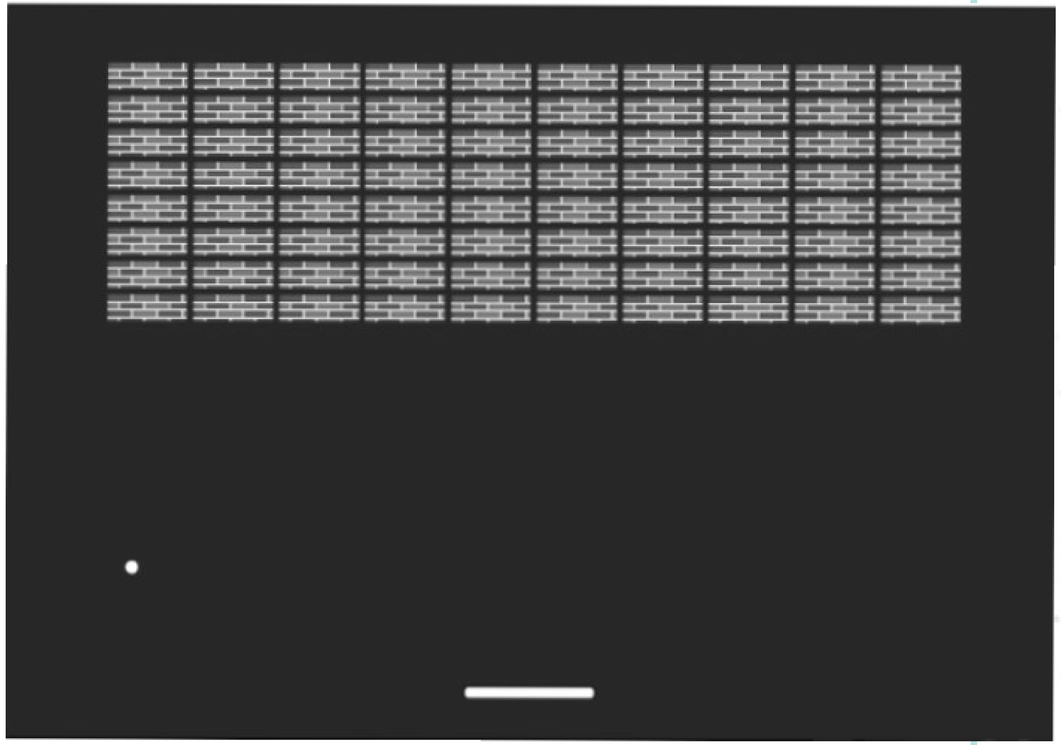
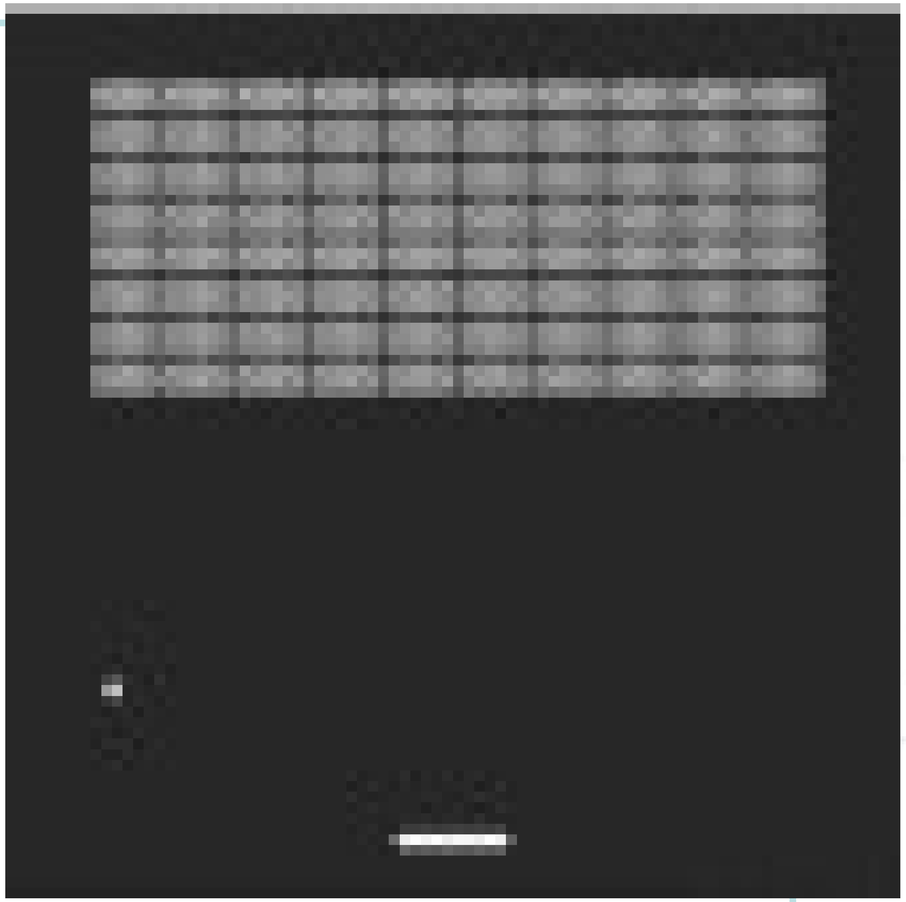


Image processing: resizing

Resize to 84x84:

“..and rescale it to 84 x 84”

```
frame = cv2.resize(
            frame, (self._width, self._height),
interpolation=cv2.INTER_AREA
    )
```



Summary

- State secret sauce
- Max colour
- Greyscale
- Resize