Section 4:
Deep Reinforcement Learning

Goals for today's section:

Solidify understanding of RL

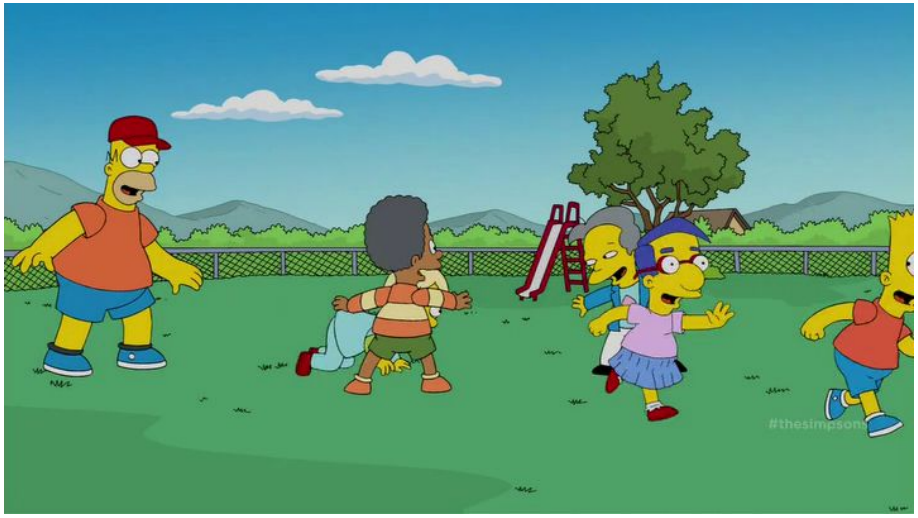
Understand Q-learning / Q function approximation

Introduce Deep Reinforcement Learning

Can we learn to control an agent directly from sensory input?

Games: A Playground for RL Agents

- Games provide less complex environment than the real world
- Games can be easily simulated and reproduced
- Rewards are clearly defined, e.g. as scores or there is a winner



→ Game environments are great for trying out RL algorithms!


Remember
Breakout?

Breakout Game Description

- Ball travels across the screen and bounces of the walls and the paddle
- Ball is lost when it falls off the bottom of the screen
- Move the paddle left or right to prevent ball from falling off
- Earn points by destroying bricks with the ball

https://www.youtube.com/watch?v=y_P7fTjl7eU

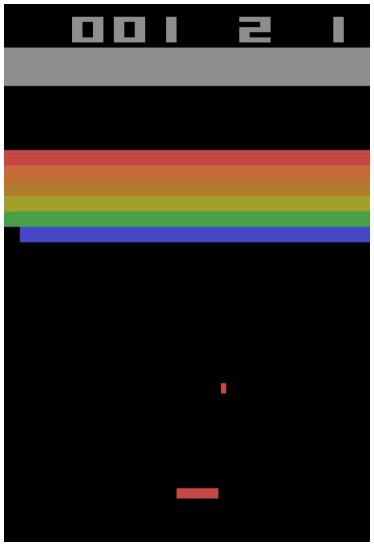
Breakout Game Description

Formally:

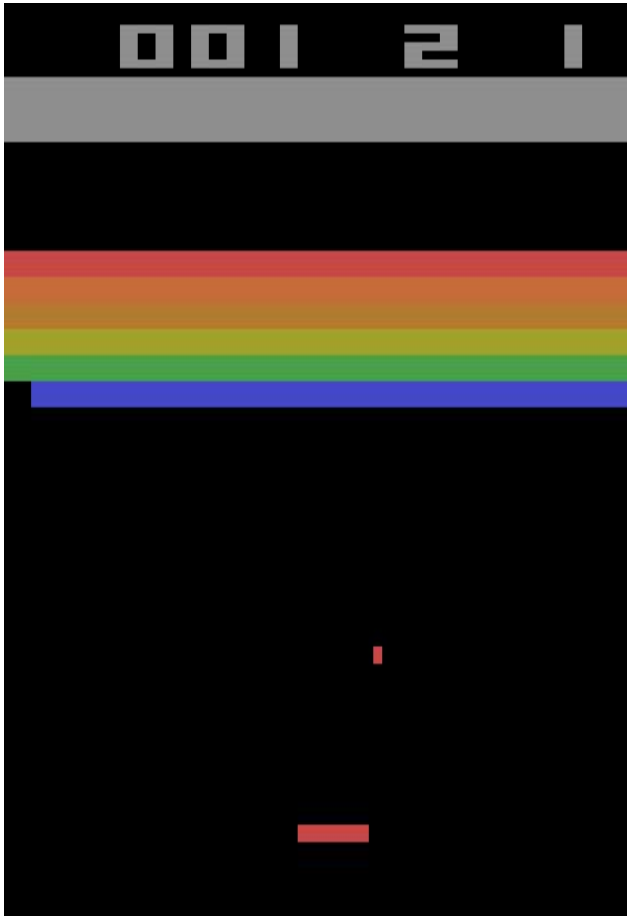
- *Actions*
  - move_paddle_left
  - move_paddle_right
  - do_not_move_paddle
- *Rewards*
  - If ball hits brick, reward = 1
  - Otherwise, reward = 0
- *End condition*
  - If ball falls off the screen, game ends

Can we learn to control an agent directly from sensory input?

In Breakout, sensory input would be a game screen frame.



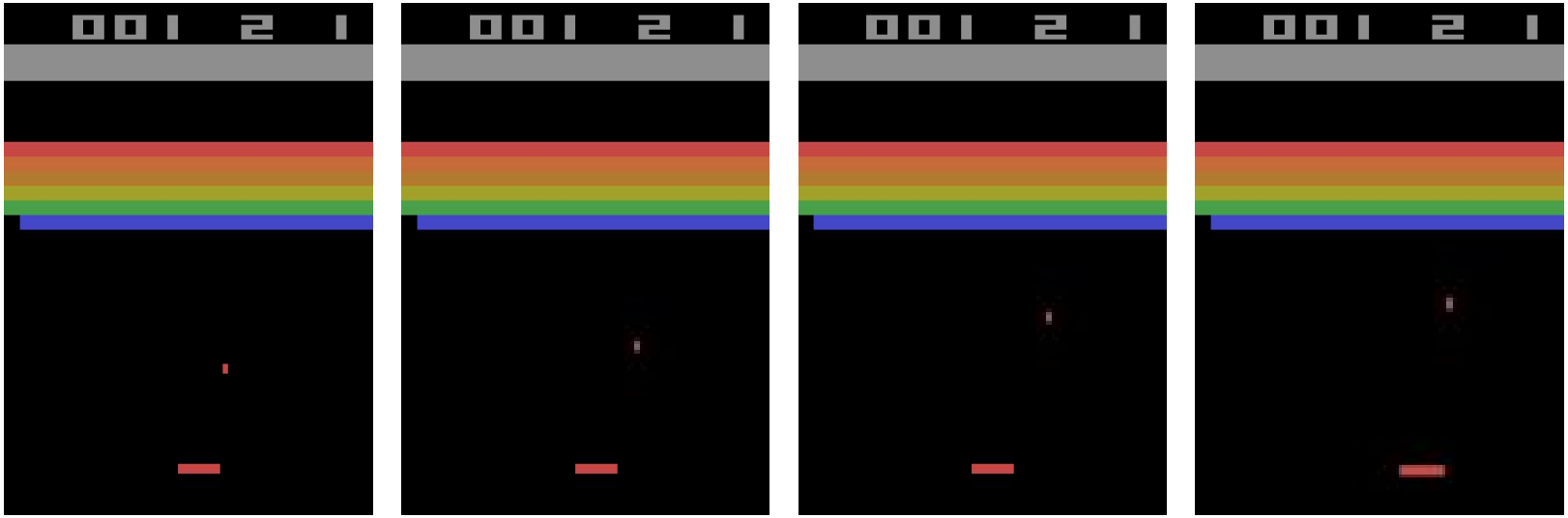
Finding a state representation



Consider this frame.

- Can you capture information like direction of the ball?
- Can you capture velocity?

Use a small number of consecutive frames for each state.

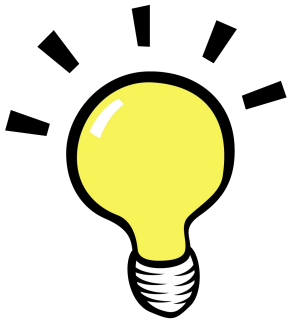


Estimate # of possible states

(assume 84x84 pixels per screenshot, where each pixel can take on 256 values, and 4 screenshots per state)

$$\#\ \text{states} \approx 256^{84 \times 84 \times 4}$$

How do we learn a policy given that the # of states is so large?



Use function approximation!

Q-value function (Review)

Q_{opt}(s, a)**:* Expected utility of first taking action ***a from state ***s*** and then following optimal policy ***opt.***

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')]$$

Once you have ***Q_{opt}(s, a)***, infer optimal policy by picking the action ***a*** that maximizes ***Q_{opt}(s, a)*** for every state.

→ *Sounds great, but how do we learn the Q-value function?*

Q-Function Approximation (Review)

Key idea: linear regression model

Define **features** $\phi(s, a)$ and **weights** $\mathbf{w}$:

$$\hat{Q}_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$$

Q-Function Approximation (Review)

Objective Function:

$$\min_{\mathbf{w}} \sum_{(s,a,r,s')} (\underbrace{\hat{Q}_{\text{opt}}(s,a;\mathbf{w})}_{\text{prediction}} - \underbrace{(r+\gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}})^2$$

*How do we learn the weights **w** which optimize this function?*

Q-Function Approximation (Review)

Optimize objective function with good old gradient descent.

Algorithm: Q-learning with function approximation

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}]\phi(s, a)$$

How would you design the features $\Phi(s, a)$ for Breakout?

Designing features Φ(s, a)

- Include properties from state ***s***
- Include action ***a***
- For example:
  - Φ(s, a)[0] = x location of paddle
  - Φ(s, a)[1] = y location of paddle
  - Φ(s, a)[3] = x distance between paddle and ball
  - Φ(s, a)[5] = x velocity of ball
  - …
  - Φ(s, a)[21] = 1 if action is “move_right”, 0 otherwise
  - …

→ ***Handcrafting features is very difficult!***

→ ***We also need to build a paddle detector, etc.***

Drawbacks of current feature extraction

- We have to **“handcraft”** features **Φ(s, a)**
- Performance depends on the quality of features **Φ(s, a)**
- **Not generalized**: features are specific to Breakout, rely on **domain knowledge**
- Building a feature detector (e.g. to get location of paddle) might still seem reasonable for Breakout, but could be much harder for more complex games.

How do we generalize the features?

(so features are not specific to Breakout, but could be extended to other Atari games)

What if we used the pixel values themselves as features?

Finding a new feature representation

Previously:

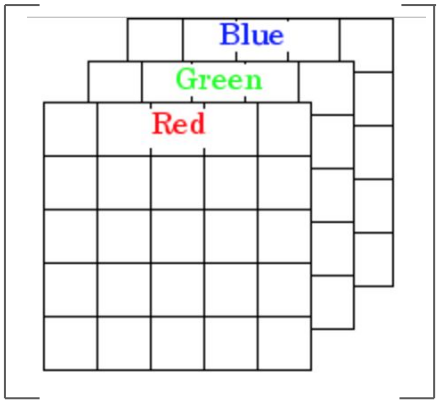
Handcrafted features

‘Paddle Position’: [8.0, 0.0]
‘Ball Position’: [12.0, 11.0]
‘Ball Velocity’: [3.0px/s, 5.0px/s]
‘Num Bricks’: 34
‘Brick Pos’: [(a, b), (c, d), …]

→ RL

Now:

→ Φ



→ ? → RL

Would a linear regression model work well with pixel features? Recall:

Key idea: linear regression model

Define **features** $\phi(s, a)$ and **weights** $\mathbf{w}$:

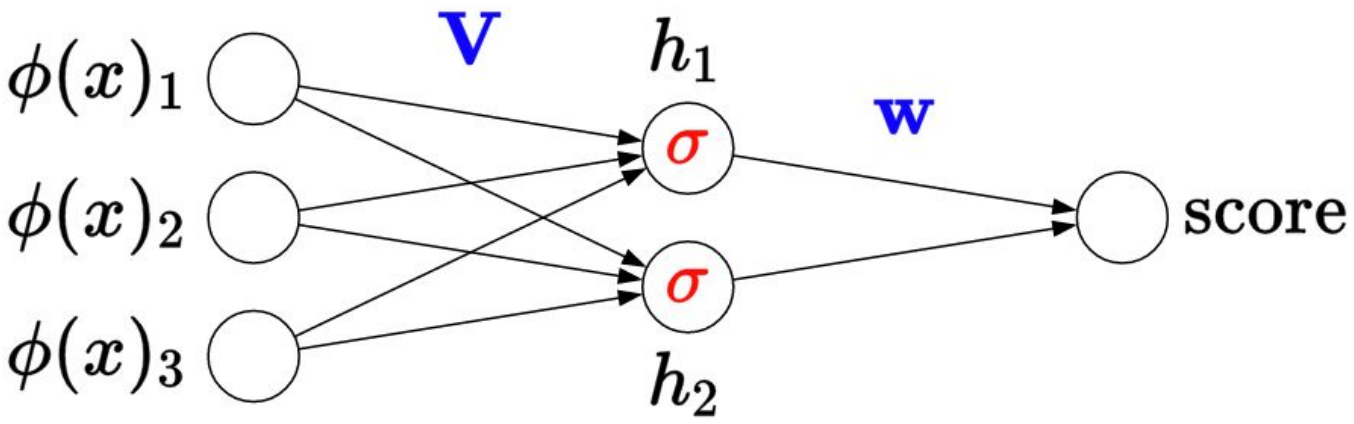
$$\hat{Q}_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$$

Relationship between pixel values and Q-value most likely not linear!

What other function approximators could we use?

Deep Neural Nets (Review)

Neural network:



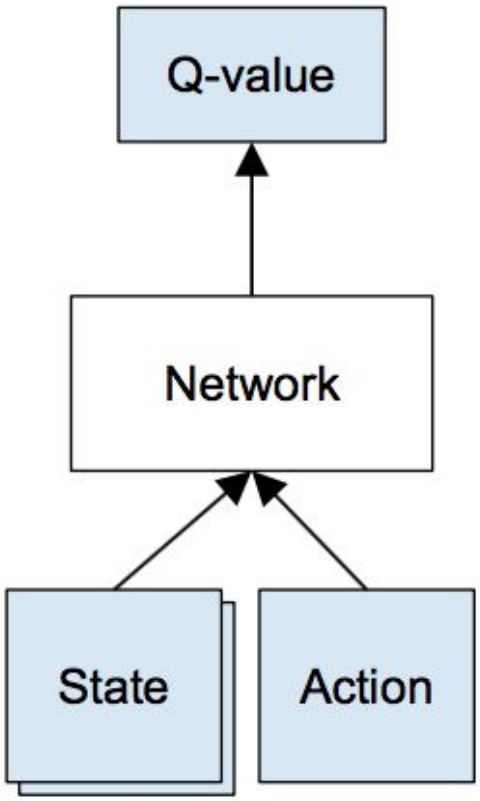
Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

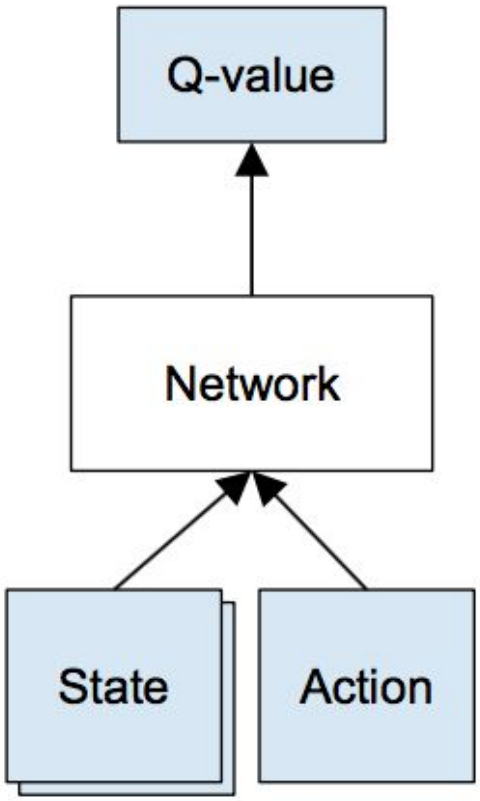
Neural Networks as Q(s, a) approximators



- State and action pair passed as inputs to a neural network.
- Neural network predicts the Q-value for the input action.

Can we make this even more efficient?

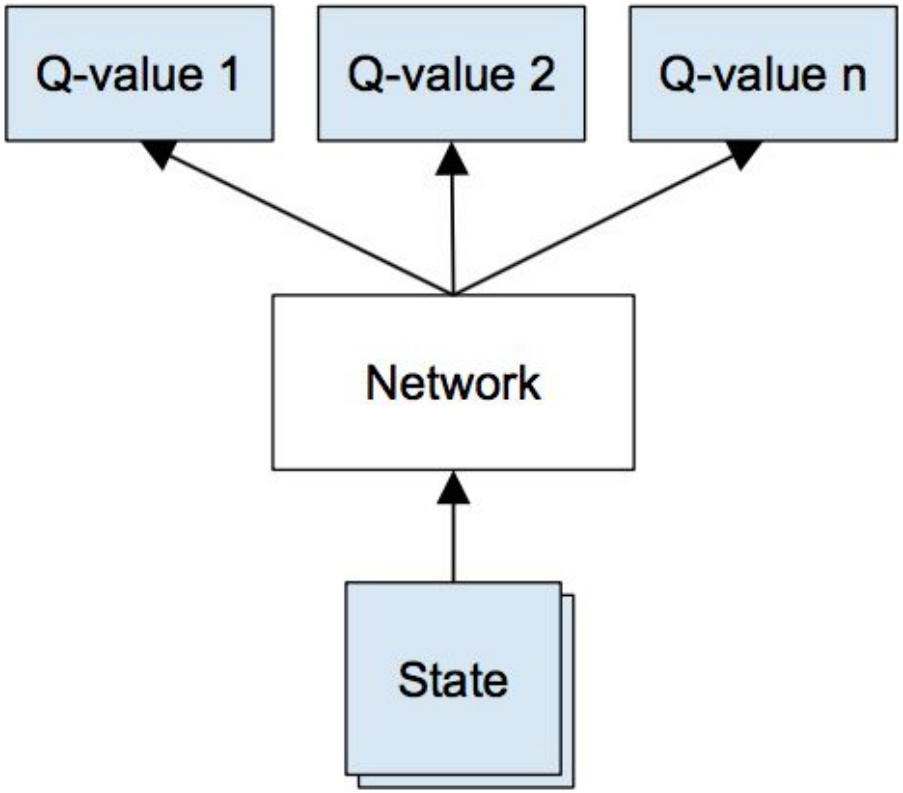
Neural Networks as Q(s, a) approximators



- State $\in R^{84x84x4}$, Action $\in R$
  State dimensions are much larger than action dimensions. State will overshadow action.
- Network predicts Q-value for a single action. Consequently, multiple runs required, one for each action

How do we circumvent these problems?

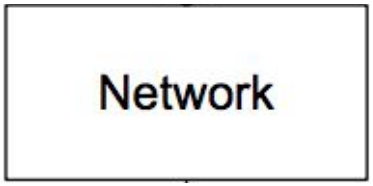
Neural Networks as Q(s, a) approximators



- State is the only input into the neural network.
- Network outputs a Q-value for every possible action.
- Action corresponding to the highest Q-value is chosen.

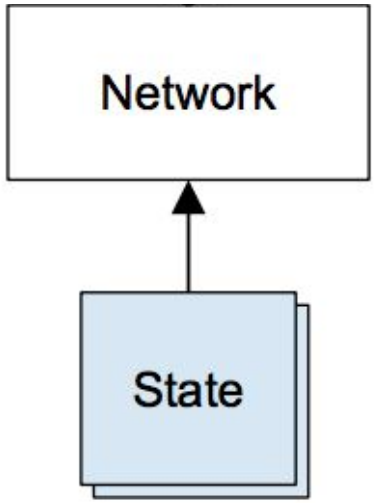
A single network to predict Q(s,a) for all possible states and actions!

Training Deep-Q-Networks (DQN)



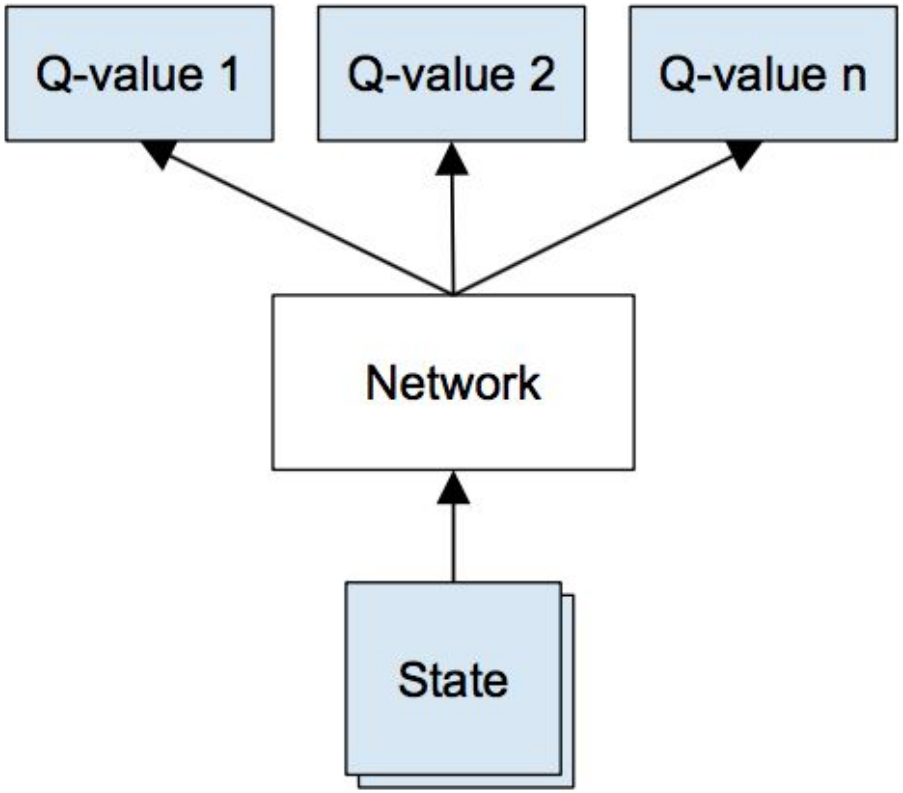
Initialize weights randomly!

Training Deep-Q-Networks (DQN)



States are passed as input.

Training Deep-Q-Networks (DQN)

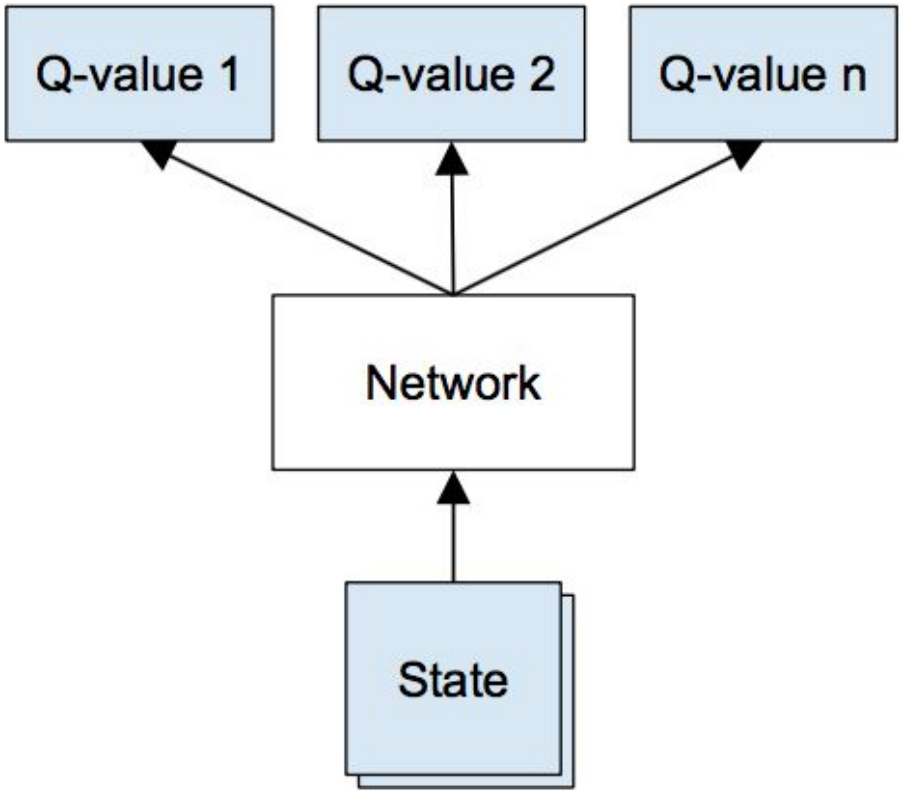


- Network outputs Q-values for each possible action.
- Action space for Breakout:

  [left, right, no-op]

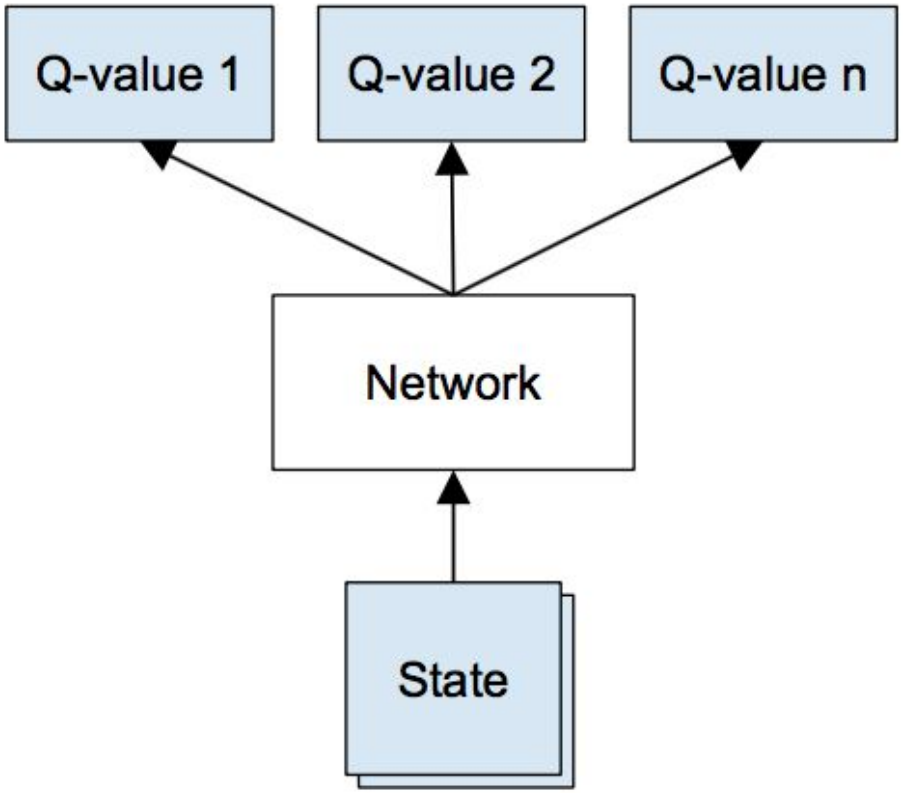
- Since initial weights are random, Q-values are random.

Training Deep-Q-Networks (DQN)



- Execute action that maximizes Q-value.
- Environment may or may not react to that action with a reward.
- Obtain next state.

Training Deep-Q-Networks (DQN)



- Run gradient descent on Q-learning loss.

Training Deep-Q-Networks

- Initialize weights randomly.
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action.
  - Execute action (a) that maximizes Q-value.
  - Obtain reward (r) and new state (s’).
  - Perform gradient descent on Q-learning loss using (s, a, r, s’)

Training Deep-Q-Networks

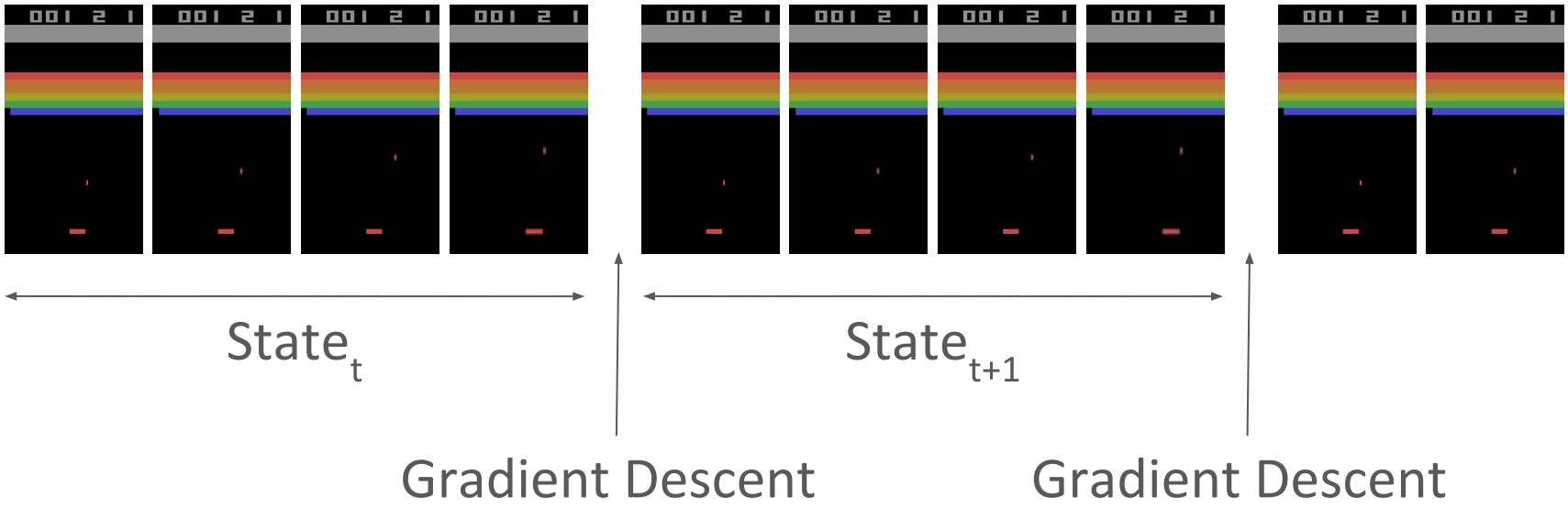
- Initialize weights randomly.
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action.
  - Execute action (a) that maximizes Q-value.
  - Obtain reward (r) and new state (s’).
  - Perform gradient descent on Q-learning loss using (s, a, r, s’)

Training Deep-Q-Networks

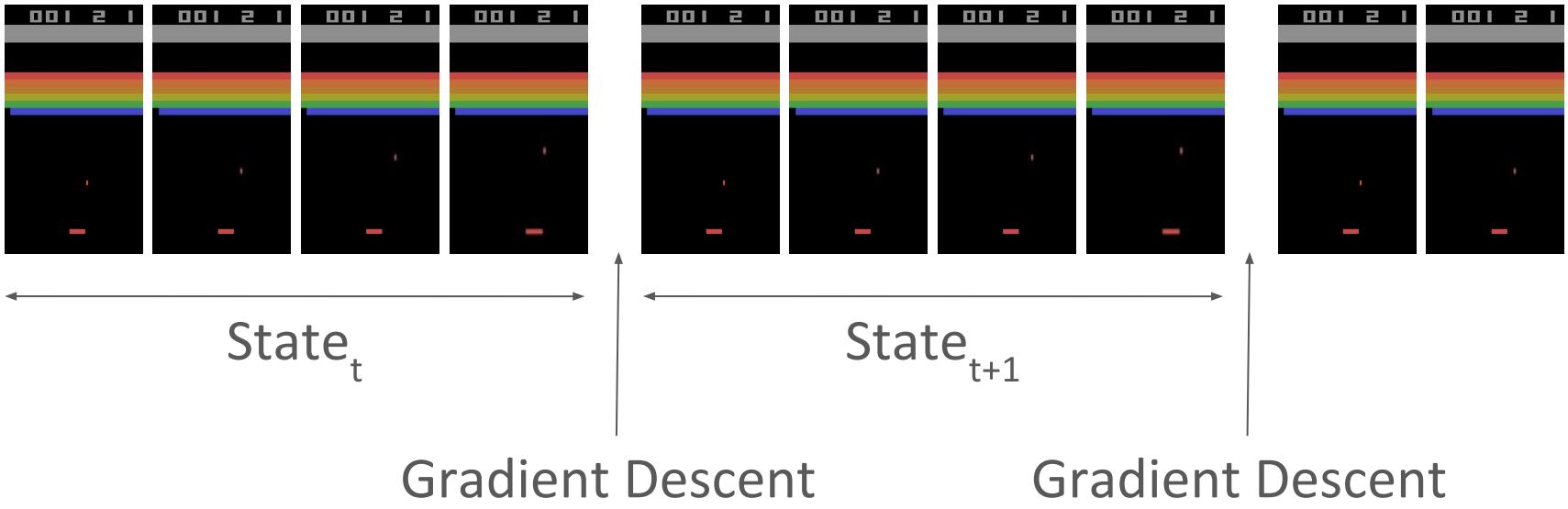
- Initialize weights randomly.
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action.
  - Execute action (a) that maximizes Q-value.
  - Obtain reward (r) and new state (s').
  - Perform gradient descent on Q-learning loss using (s, a, r, s')

Unstable and inefficient!

Training DQNs can be difficult



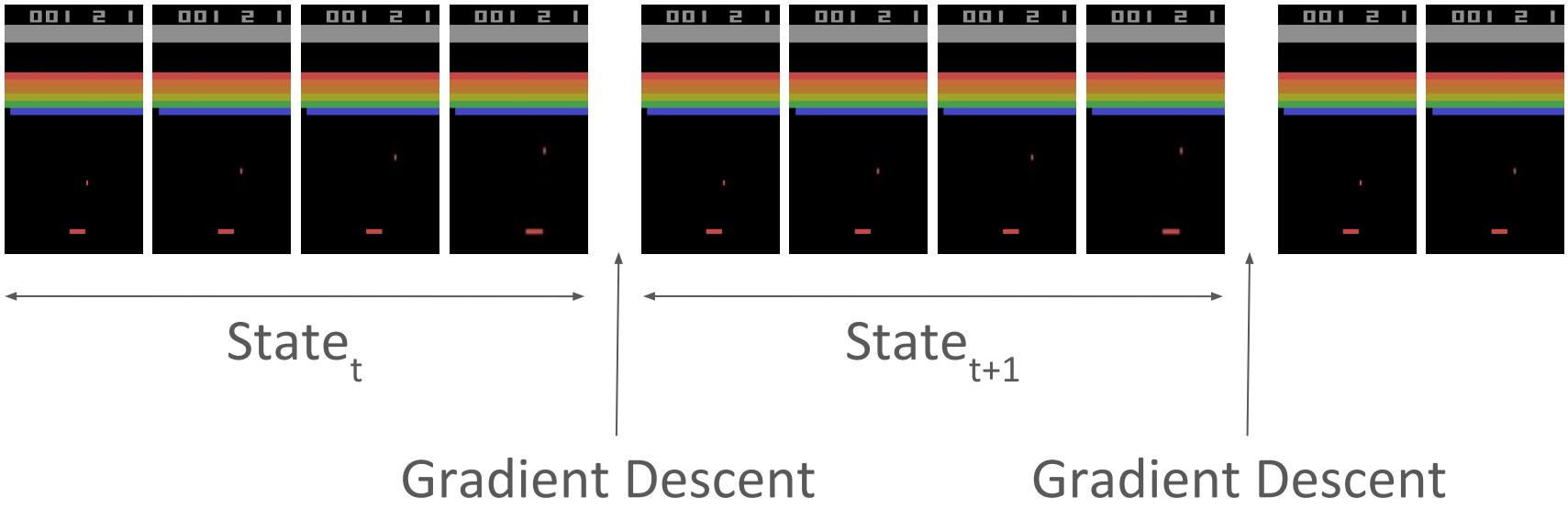
Training DQNs can be difficult



- State$_t$ is highly correlated to State$_{t+1}$
- Gradient descent after consecutive steps ⇒ erratic updates

How do we fix this?

Training DQNs can be difficult



- State$_t$ is highly correlated to State$_{t+1}$
- Gradient descent after consecutive steps ⇒ erratic updates

How do we fix this? Randomly sample states for updates.

Training Deep-Q-Networks

- Initialize weights randomly.
- Initialize memory (D) with capacity N.
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action.
  - Execute action (a) that maximizes Q-value.
  - Obtain reward (r) and new state (s').
  - Store (s, a, r, s') in D
  - Randomly sample $(s, a, r, s')_D$ from D
  - Perform gradient descent on Q-learning loss using $(s, a, r, s')_D$

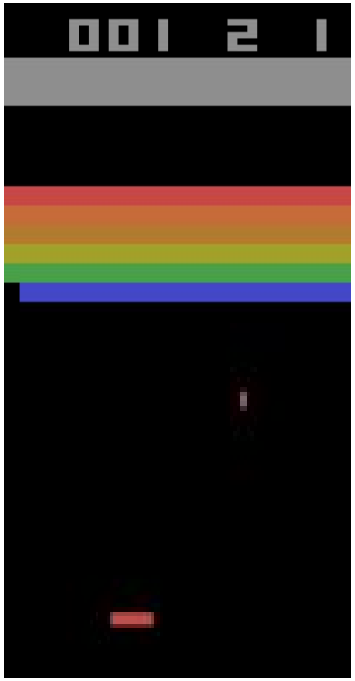
Training Deep-Q-Networks

Experience Replay

- Initialize weights randomly.
- Initialize memory (D) with capacity N.
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action.
  - Execute action (a) that maximizes Q-value.
  - Obtain reward (r) and new state (s').
  - Store (s, a, r, s') in D
  - Randomly sample $(s, a, r, s')_D$ from D
  - Perform gradient descent on Q-learning loss using $(s, a, r, s')_D$

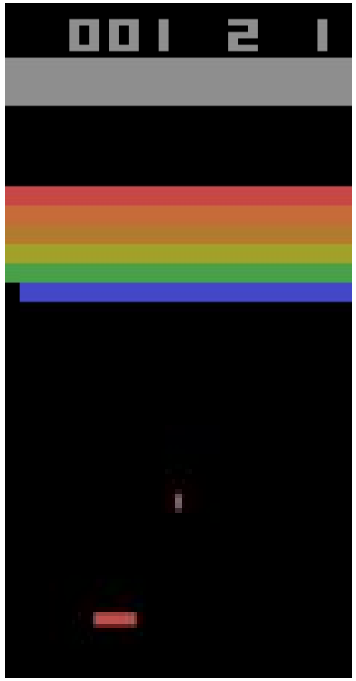
Training DQNs can be difficult - Part Two

Consider four distinct scenarios:



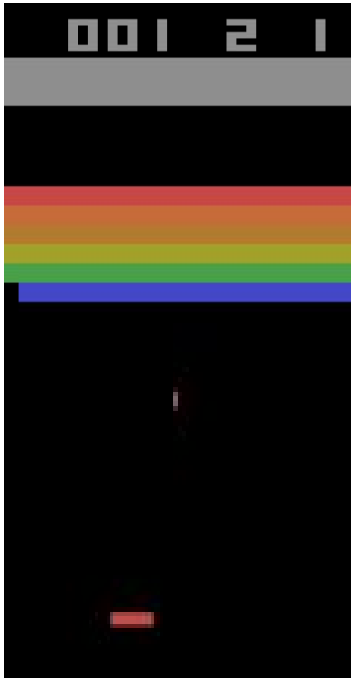
Action: right

Reward: pos



Action: right

Reward: pos



Action: right

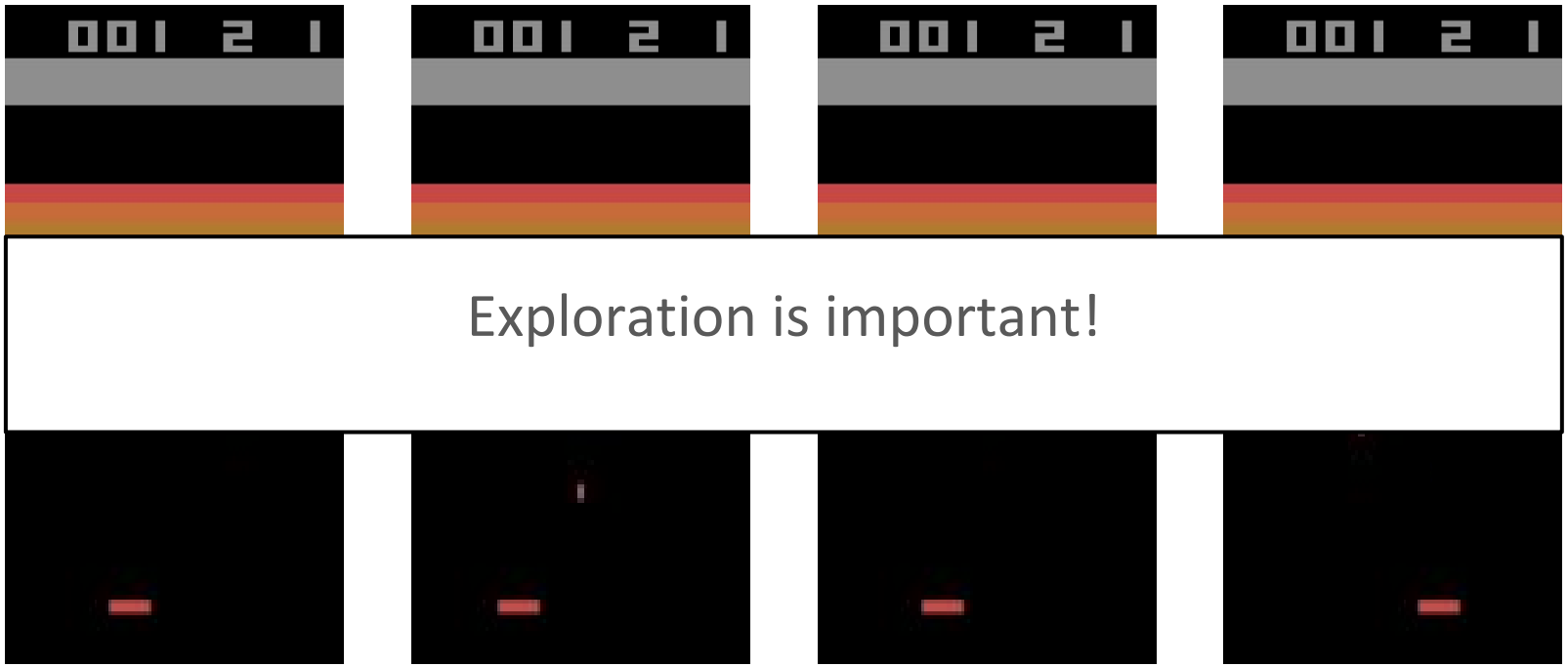
Reward: pos

Action: right?

Reward: neg

Training DQNs can be difficult - Part Two

Consider four distinct scenarios:



Action: right	Action: right	Action: right	Action: right?
Reward: pos	Reward: pos	Reward: pos	Reward: neg

Training Deep-Q-Networks

- Initialize weights randomly
- Initialize memory (D) with capacity N
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action
  - With probability ε, execute random action (a)
  - Otherwise, execute action (a) that maximizes Q-value
  - Obtain reward (r) and new state (s')
  - Store (s, a, r, s') in D
  - Randomly sample $(s, a, r, s')_D$ from D
  - Perform gradient descent on Q-learning loss using $(s, a, r, s')_D$

Training Deep-Q-Networks

ε-Greedy

- Initialize weights randomly
- Initialize memory (D) with capacity N
- Loop:
  - Obtain current state (s)
  - Run Neural Network on s to obtain Q-value for every action
  - With probability ε, execute random action (a)
  - Otherwise, execute action (a) that maximizes Q-value
  - Obtain reward (r) and new state (s')
  - Store (s, a, r, s') in D
  - Randomly sample $(s, a, r, s')_D$ from D
  - Perform gradient descent on Q-learning loss using $(s, a, r, s')_D$

Let’s watch Deep RL in action

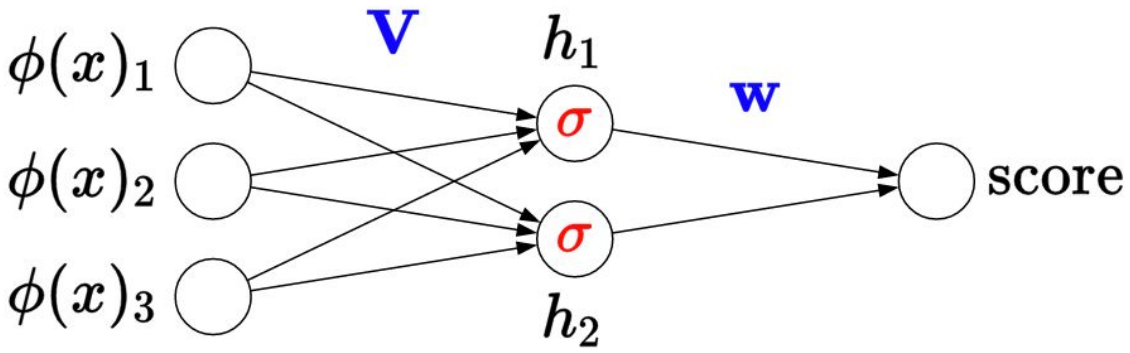
https://www.youtube.com/watch?v=V1eYniJ0Rnk

Can we analyze DQNs to understand what it actually learned?

Did it learn high-level concepts?

Looking inside a neural net

Neural network:



Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

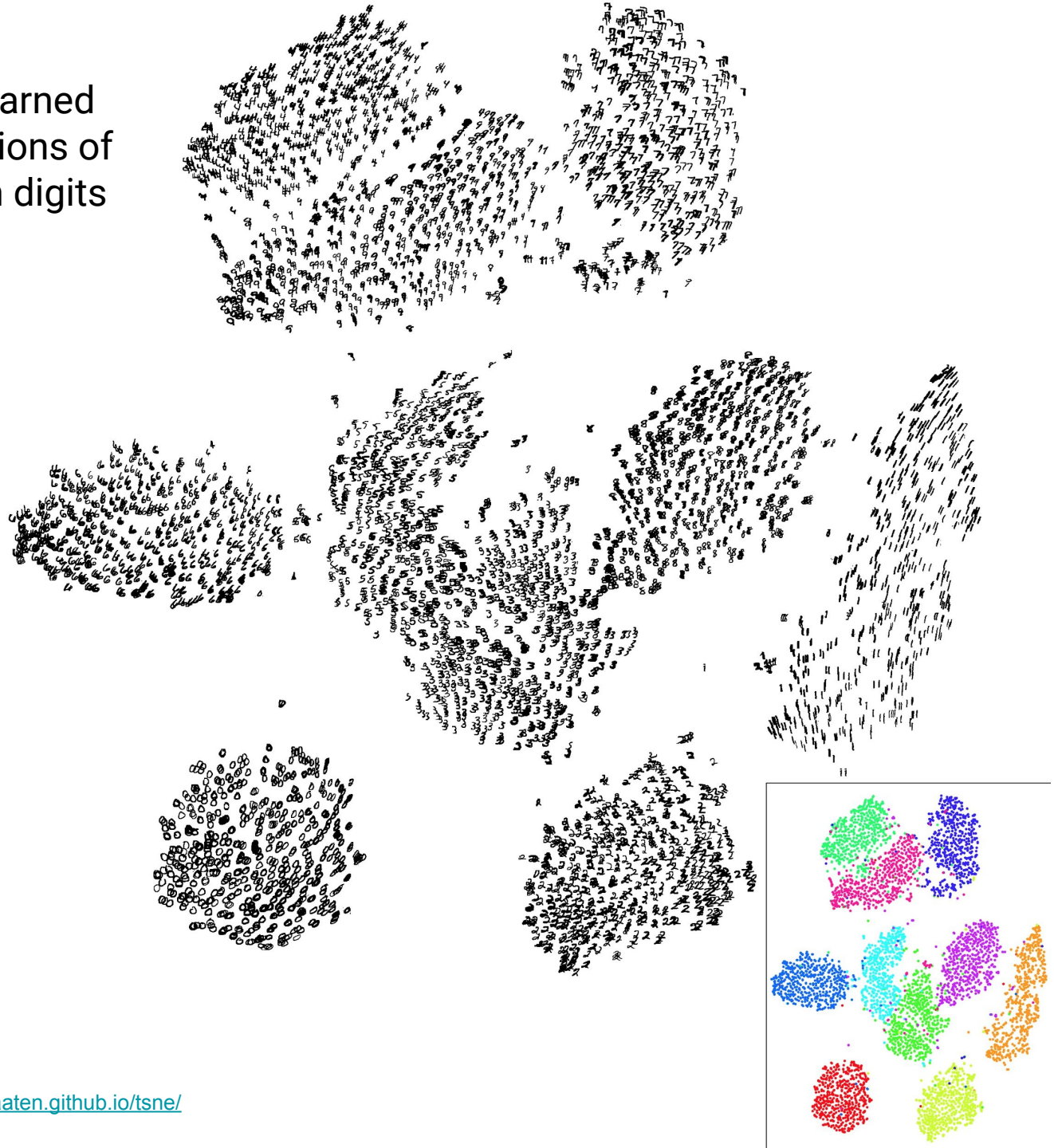
h is a hidden representation of the input ***Φ(x)***

Ideally, ***h*** captures high-level information about the input

Visualizing the hidden representations

- Truly understanding neural nets is hard. But we can visualize hidden representations!
- Representations often have more than 2 or 3 dims, e.g. 128
- We can't plot more than 3 dims!
- To visualize them, we have to perform dimensionality reduction
- One popular method is t-SNE (t-Distributed Stochastic Neighbor Embedding), which maps high-dimensional vector representations to 2 or 3 dimensions.

T-SNE on learned representations of handwritten digits (MNIST)

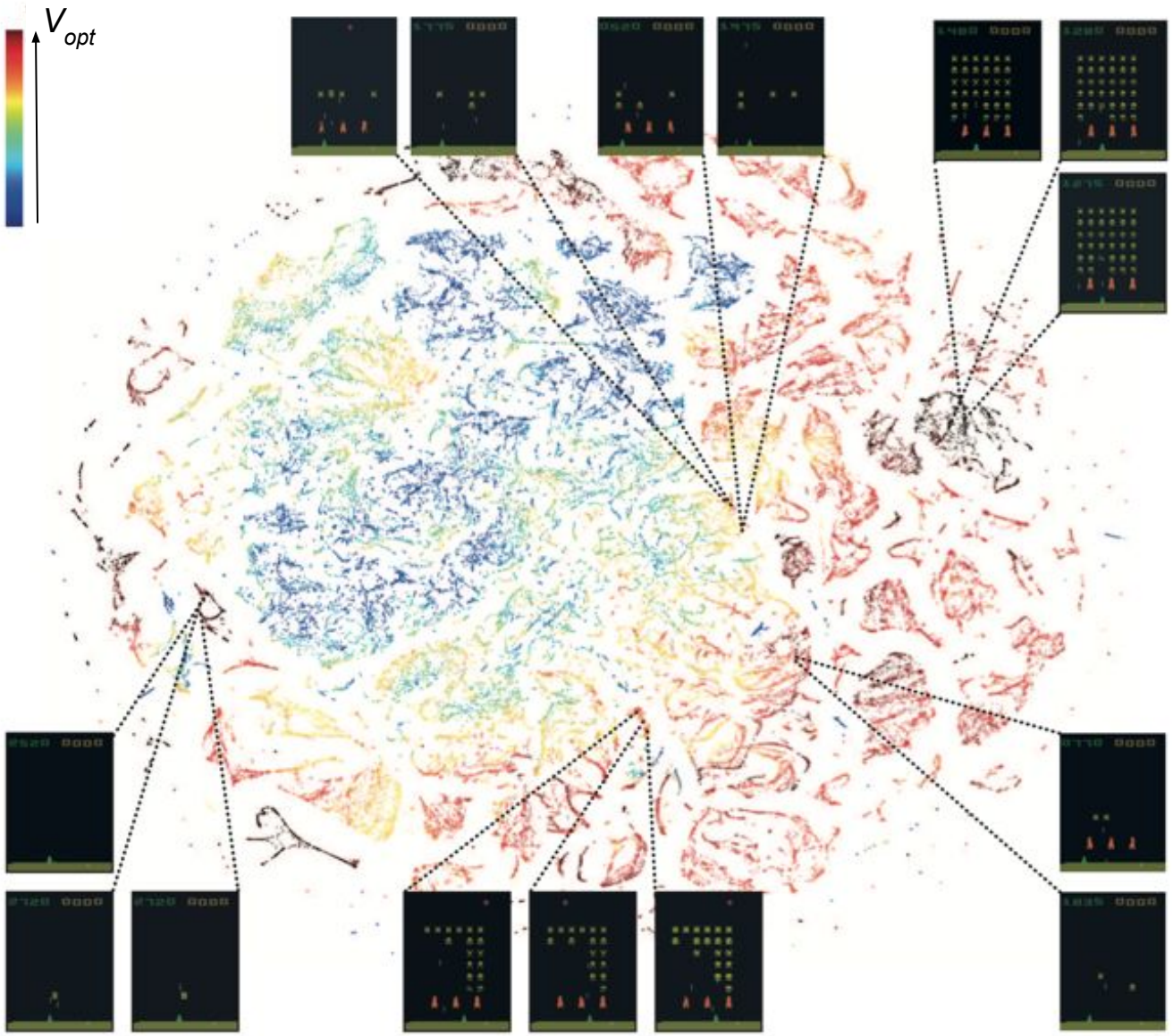


Source: https://lvdmaaten.github.io/tsne/

T-SNE on image representations (ImageNet)

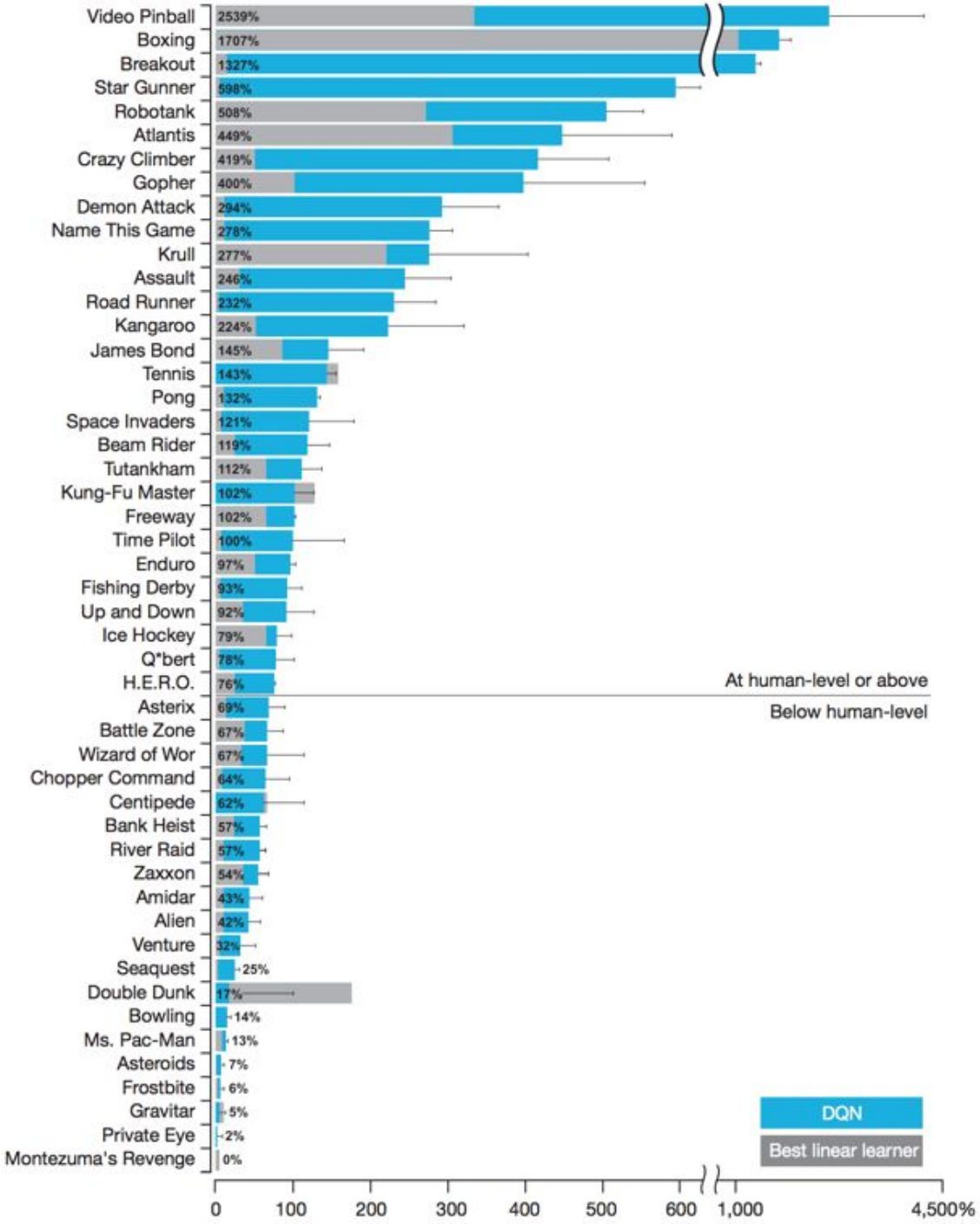
https://jeremiahwjohnson.com/2016/02/09/visualizing-imagenet-with-tsne/

T-SNE Learned State Representations for Space Invaders
(last hidden layer of DQN)



Source:
[2]

How well does DQN work on other Atari games?



Comparison of the DQN agent with the best RL methods in the literature

The performance of DQN is normalized w.r.t. A professional human games tester (that is, 100% level) and random play (that is, 0% level).

Source: Mnih et al. (2015)

Video Pinball

(DQN does very well, 2539% of human performance)

Time Pilot

(DQN reaches human gaming performance)

Montezuma's Revenge

(DQN does very poorly, similar to random gameplay)

Shortcomings of DQN

- Does not work well if environment has sparse, delayed rewards
- Does not work well in continuous action space
- Multi-agent co-operation
- Spatio-temporal abstractions on state and action spaces
- Transfer across games

Active Area of Research: Transfer Learning

Inspiration: Humans can train on tasks A, B and apply knowledge to new task C.

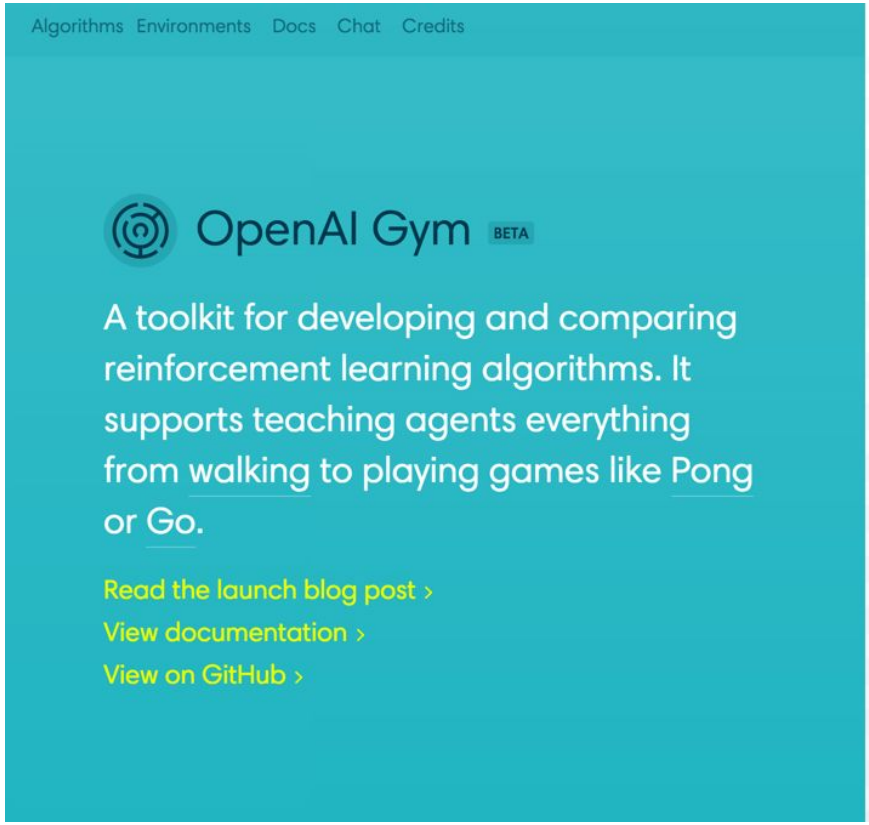
Can we train DQN across multiple tasks and “transfer” knowledge?

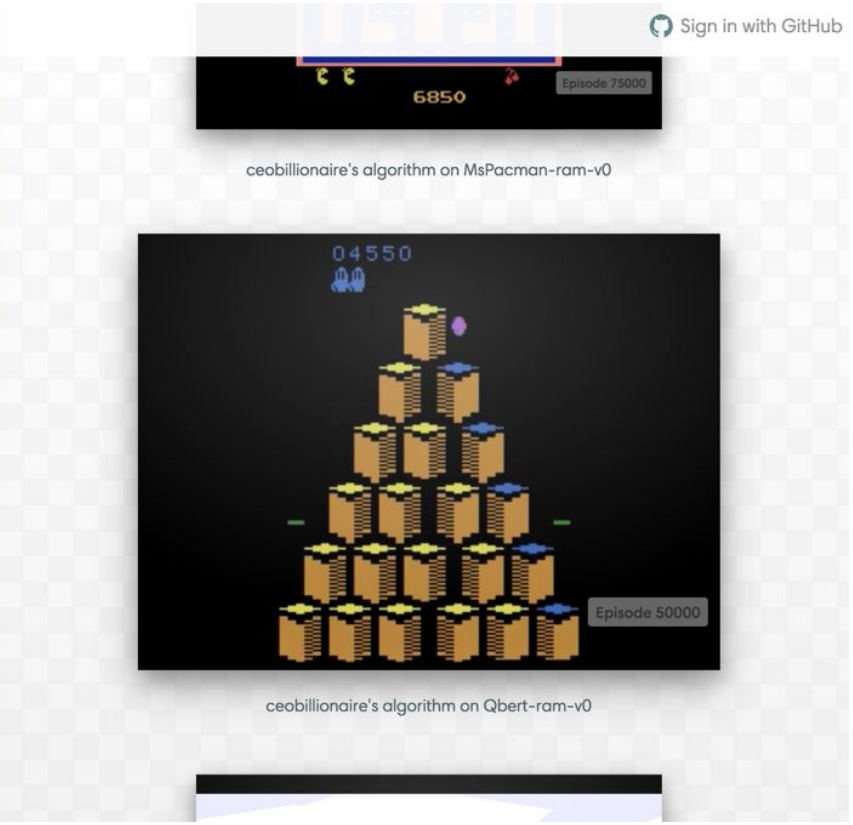
For example:

- Train DQN on multiple Atari games
- Train DQN on one game (e.g. Pong), then fine-tune network on another (e.g. Breakout)

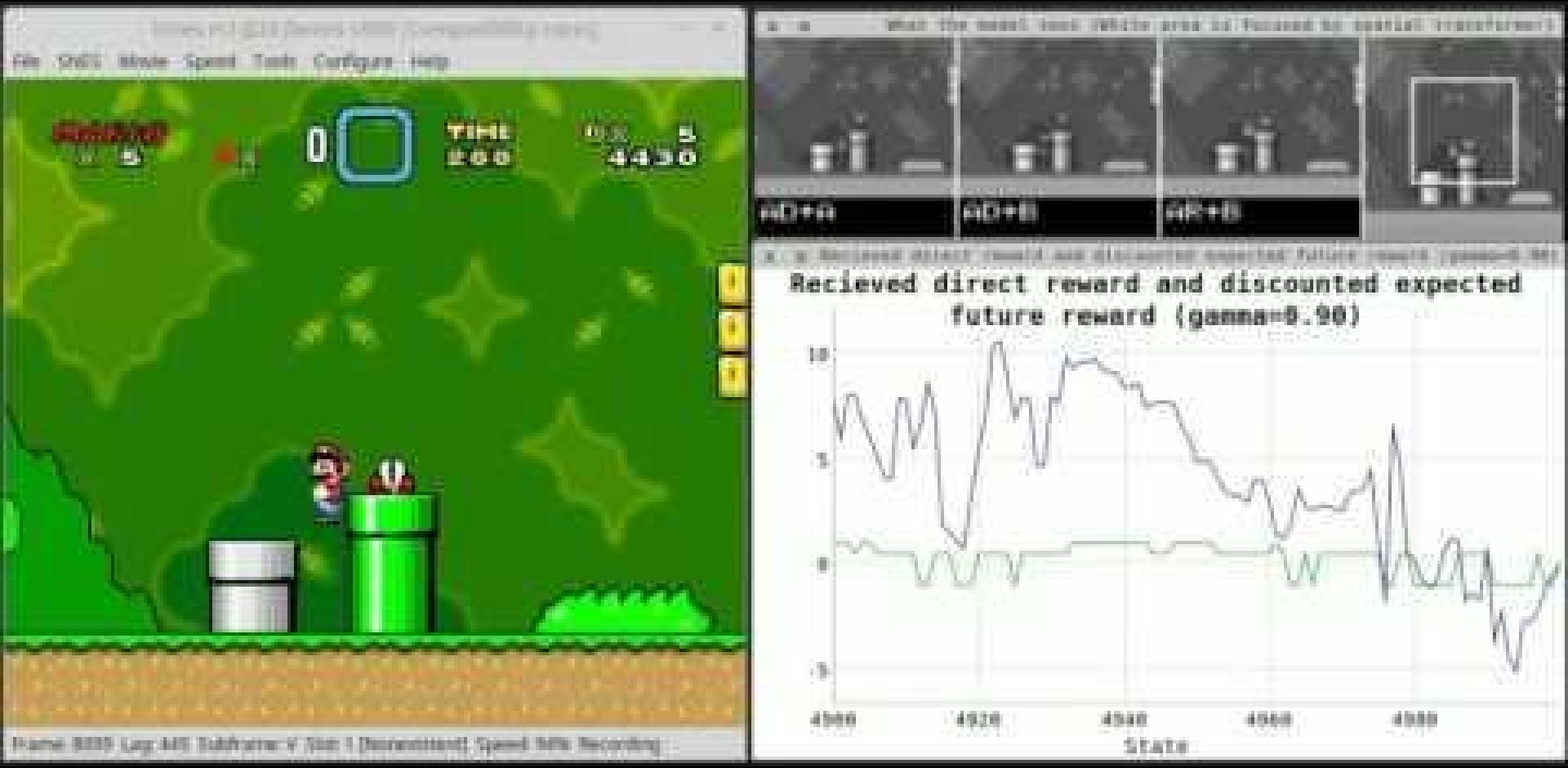
Try it out yourself!

OpenAI Gym offers many RL environments that you can play with, including Atari games.





More RL Applications!



https://www.youtube.com/watch?v=L4KBBAwF_bE

https://www.youtube.com/watch?v=spzYVhOgKBA

Summary

Deep Reinforcement Learning: combination of techniques we've learned in class

DQN: A deep neural net that predicts the Q values for all actions with sensory input

Learning from just sensory input and no domain knowledge → step towards artificial general intelligence (AGI)


You won!

Sources

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).