

# Deep Reinforcement Learning

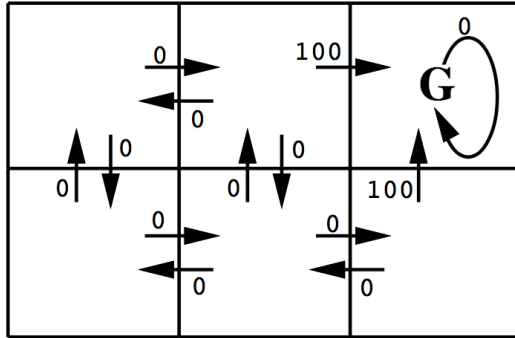
Sargur N. Srihari  
srihari@cedar.buffalo.edu

# Topics in Deep RL

1. Q-learning target function as a table
2. Learning Q as a function
3. Simple versus deep reinforcement learning
4. Deep Q Network for Atari Breakout
5. The Gym framework for RL
6. Research frontiers of RL

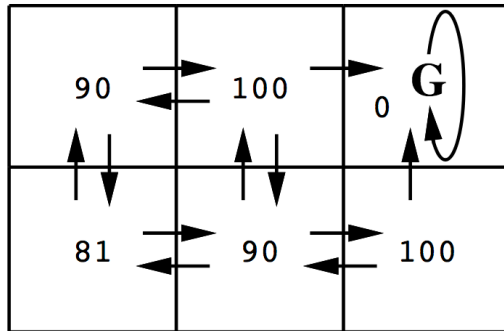
# Definitions for Q Learning & Grid world

$r(s, a)$   
(Immediate  
Reward)



$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$$

$V^*(s)$   
(Maximum  
Discounted  
Cumulative  
Reward)



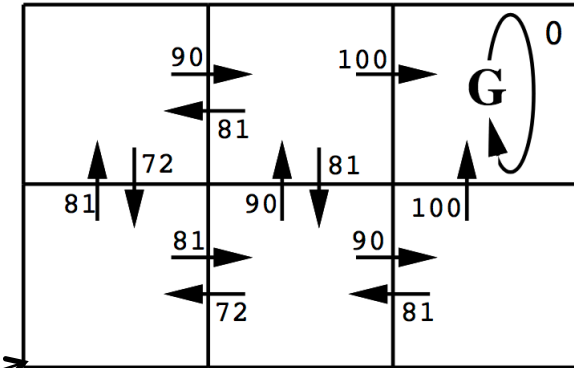
Recurrent  
Definition

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$$

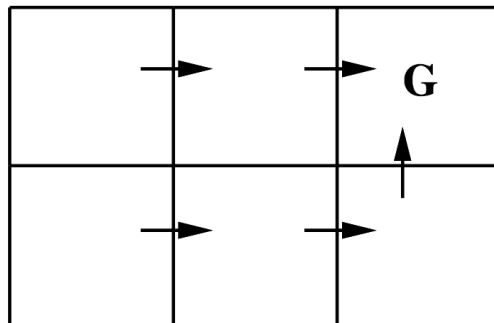
$$Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))$$

$$V^*(s) = \max_{a'} Q(s, a')$$

$Q(s, a)$   
values



One  
Optimal  
policy



$$\pi^*(s) = \arg \max_{\pi} [r(s, a) + \gamma V^*(\delta(s, a))]$$

$$\pi^*(s) = \arg \max_{\pi} Q(s, a)$$

# Q Learning → table updates

- The target function is a lookup table
  - With a distinct table entry for every state-action pair

Training rule  
(deterministic case):

$$\hat{Q}(s, a) = r(s, a) + \gamma \max_{a'} \hat{Q}(s, a')$$

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

is called Bellman's equation:

Which says, maximum future reward is immediate reward plus maximum future reward for next state

Training rule  
(non-deterministic case):

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n) \hat{Q}_{n-1}(s, a) + \alpha_n \left[ r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a') \right]$$

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	327	0	0	0	0	0	0
	.	.	.	.	.	.	.
States	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	.	.	.	.	.	.	.
States	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Q-Learning table of states by actions that is initialized to zero, then each cell is updated through training.

# Iterative Q-learning using Bellman eqn

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
     $Q[s,a] = Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$ 
     $s = s'$ 
until terminated
```

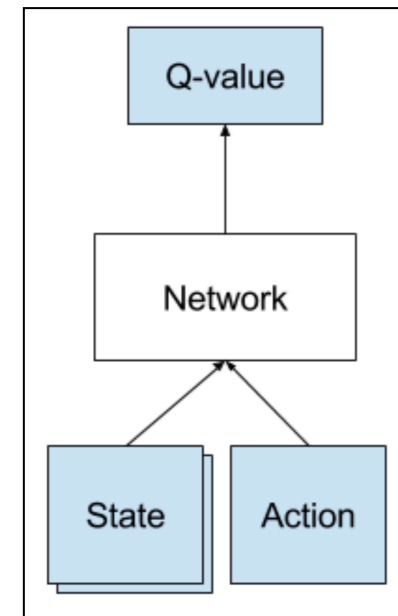
$\alpha$  is a learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. When  $\alpha=1$ , then two  $Q[s,a]$ -s cancel and the update is exactly the same as Bellman equation  $Q(s,a)=r+\gamma \max_{a'} Q(s',a')$

# Q-Learning is Rote Learning

- Target function is an explicit entry for each state-action pair
  - It makes no attempt to estimate the Q value for unseen action-state pairs
    - By generalizing from those that have been seen
- Rote learning inherent in convergence theorem
  - Relies on every  $(s, a)$  pair visited infinitely often
    - An unrealistic assumption for large or infinite spaces
- More practical RL systems combine ML function approximation methods with Q learning rules

# Learning Q as a function

- Replace  $\hat{Q}$  table with a neural net or other generalizer
  - Using each  $\hat{Q}(s, a)$  update as a training example
  - Encode  $s$  and  $a$  as inputs and train network to output target values of  $Q$  given by the training rules



Deterministic:

$$\hat{Q}(s, a) = r(s, a) + \gamma \max_{a'} \hat{Q}(s, a')$$

Non-deterministic:

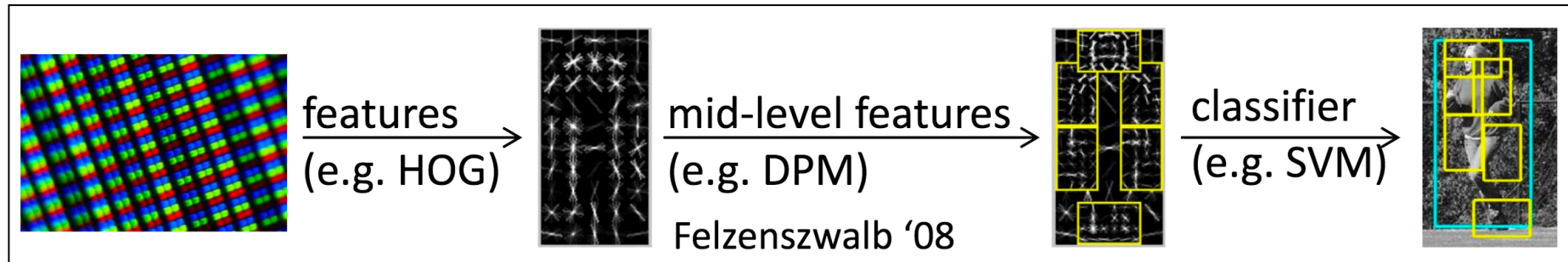
$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n) \hat{Q}_{n-1}(s, a) + \alpha_n \left[ r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a') \right]$$

Loss Function:

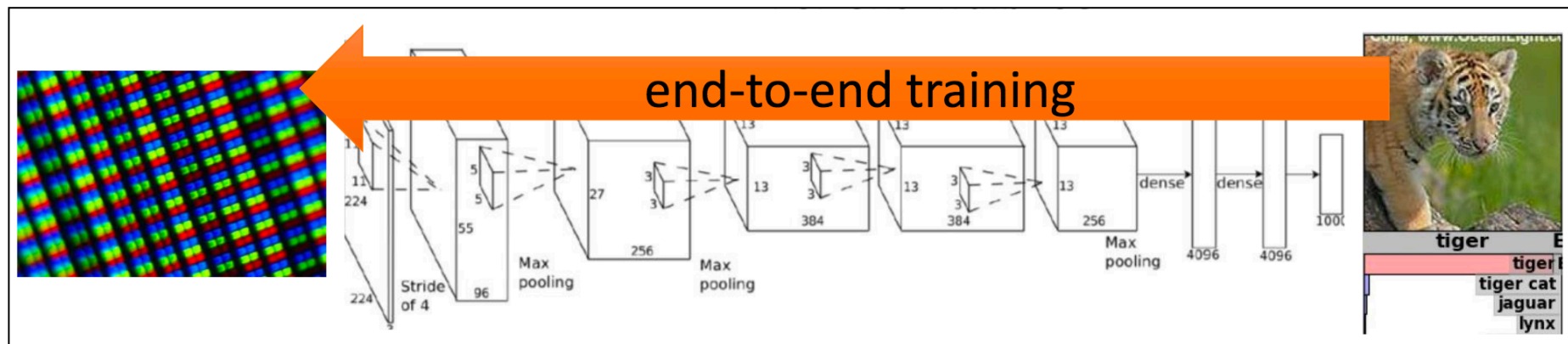
$$L = \frac{1}{2} \left[ \underbrace{r + \gamma \max_{a'} Q(s', a')}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}} \right]^2$$

# Simple ML v Deep Learning

## 1. Simple Machine Learning (e.g., SVM)



## 2. Deep Learning (e.g., Neural Net using CNN)

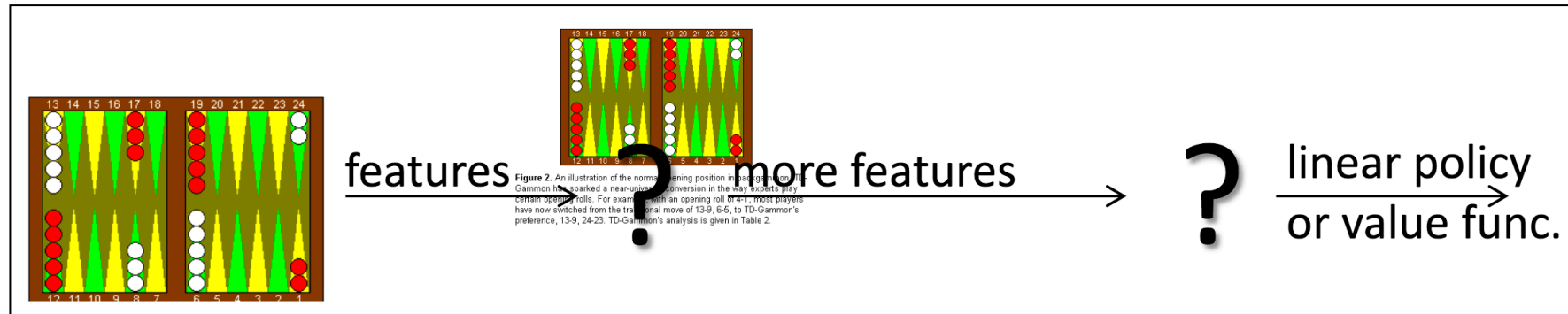


Gradient descent using Backward error propagation for computing gradients

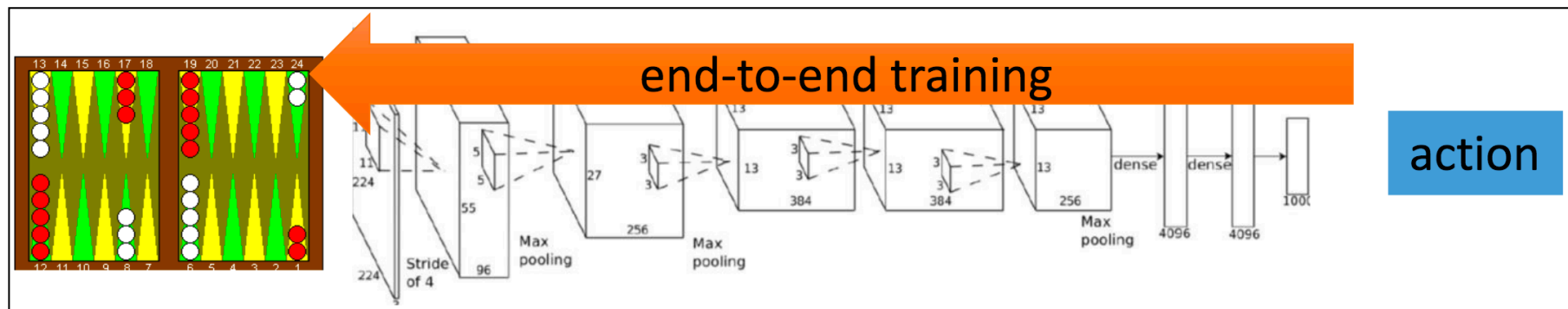


# Simple RL vs Deep RL

## 1. Simple Reinforcement Learning (Q Table Learning)



## 2. Deep Reinforcement Learning (Q Function Learning)



# Deep Q Network for Atari Breakout

- The game:
  - You control a paddle at the bottom of screen
  - Bounce the ball back to clear all the bricks in upper half of screen
  - Each time you hit a brick, it disappears and you get a reward



# Neural network to play Breakout

- Input to network: screen images
- Output would be three actions:
  - left, right or press fire (to launch the ball).
- Can treat it as a classification problem
  - Given a game screen decide: left, right or fire
  - we could record game sessions using players,
    - But that's not how we learn.
      - Don't need a million times which move to choose at each screen.
      - Just need occasional feedback that we did the right thing and can then figure out everything else ourselves
- This is the task of reinforcement learning

# What is state in Atari breakout?

- Game specific representation
  - Location of paddle
  - Location and direction of the ball
  - Existence of each individual brick
- More general representation
  - Screen pixels would contain all relevant information except speed and direction of ball
  - Two consecutive screens would cover these as well

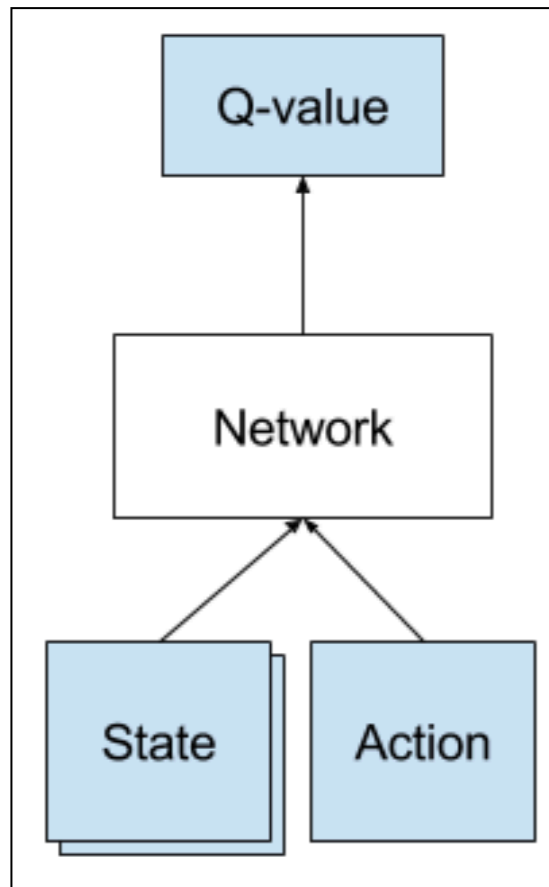


# Role of deep learning

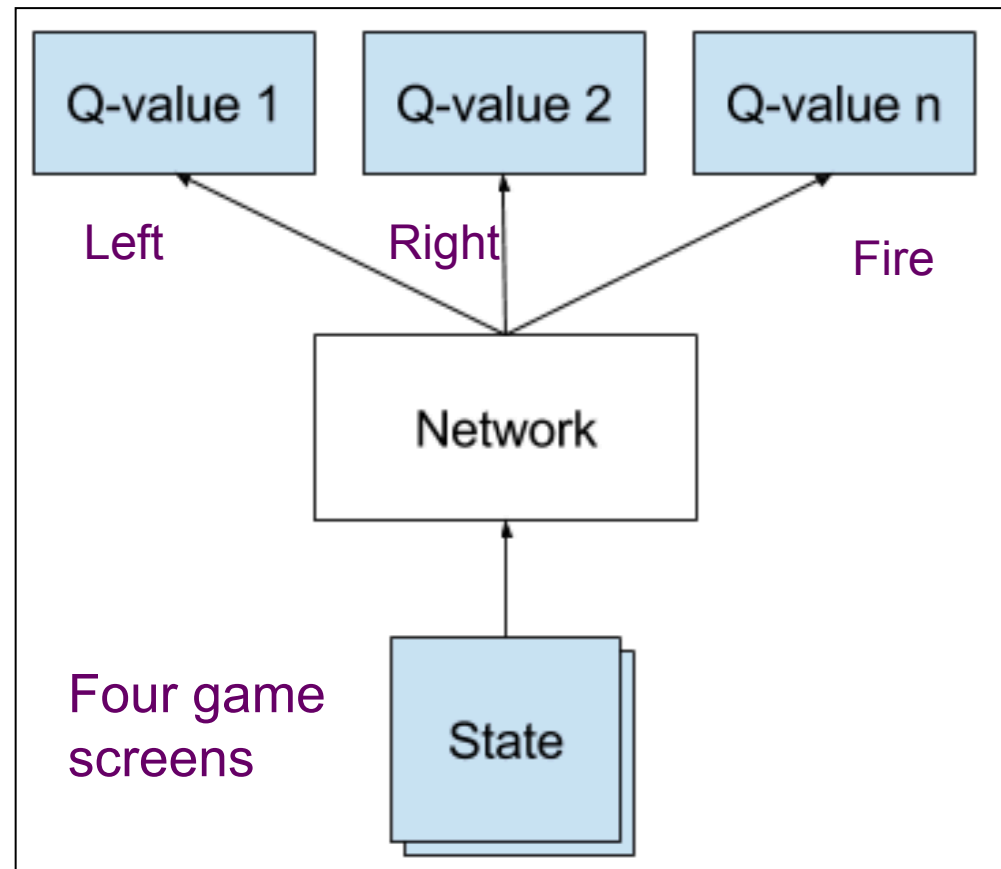
- If we take four last screen images,
- Resize them to  $84 \times 84$
- Convert to grayscale with 256 gray levels
  - we would have  $256^{84 \times 84 \times 4} \approx 10^{67970}$  game states
- Deep learning to the rescue
  - They are exceptionally good in coming up with good features for highly structured data

# Alternative architectures for Breakout

Naiive architecture



More optimal architecture

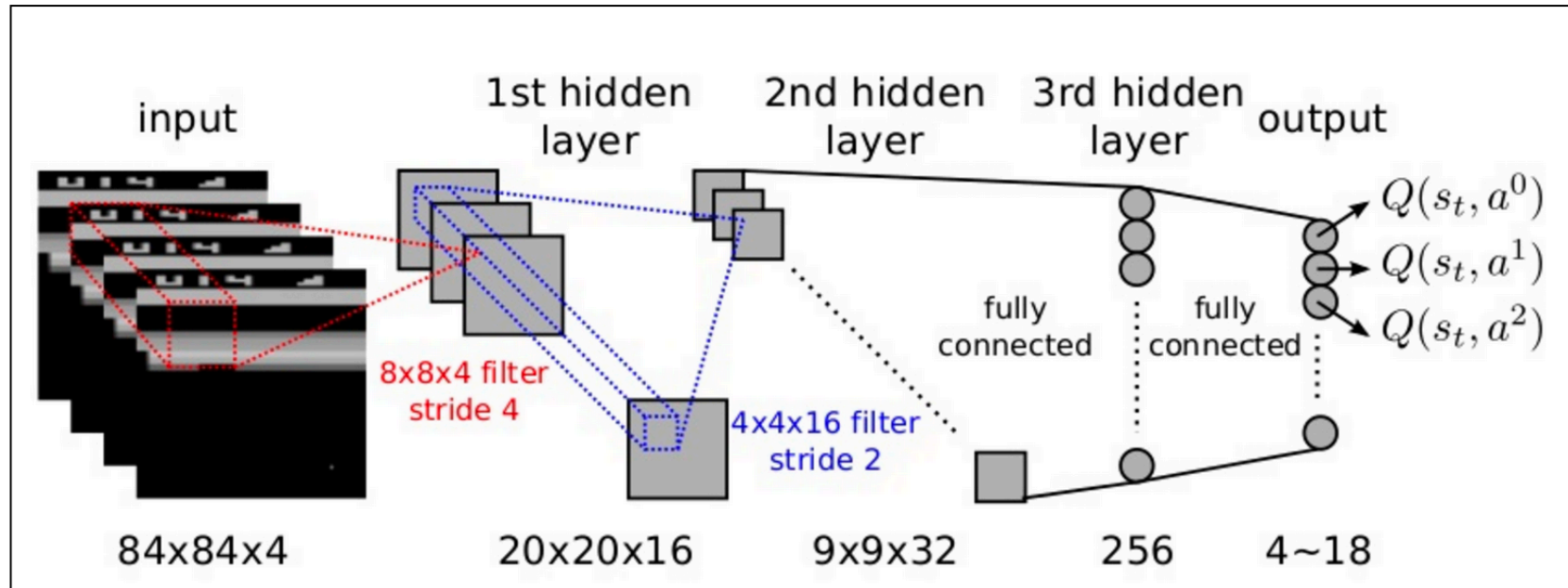


# Loss Function

- Q-values can be any real values, which makes it a regression task, that can be optimized with a simple squared error loss.

$$L = \frac{1}{2} \left[ \underbrace{r + \gamma \max_{a'} Q(s', a')}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}} \right]^2$$

# Deep Q Network for Breakout



Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18



# Q Table Update Rule

- Given a transition  $\langle s, a, r, s' \rangle$ 
  1. Do a feedforward pass for the current state  $s$  to get predicted Q-values for all actions.
  2. Do a feedforward pass for the next state  $s'$  and calculate maximum over all network outputs  $\max_{a'} Q(s, a')$
  3. Set Q-value target for action  $a$  to  $r + \gamma \max_{a'} Q(s, a')$  (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
  4. Update the weights using backpropagation.

# Experience Replay

- Approximation of Q-values using non-linear functions is not very stable
  - A bag of tricks needed for convergence
    - Also, it takes a long time, a week on a single GPU
- Most important trick is experience replay
  - During gameplay all experiences  $\langle s, a, r, s \rangle$  are stored in a replay memory
    - During training, random samples from memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples
  - Human gameplay experiences can also be used

# Q-learning using experience replay

- initialize replay memory  $D$
- initialize action-value function  $Q$  with random weights
- observe initial state  $s$
- repeat
  - select an action  $a$ 
    - with probability  $\epsilon$  select a random action
    - otherwise select  $a = \operatorname{argmax}_a Q(s, a)$
  - carry out action  $a$  observe reward  $r$  and new state  $s'$  store experience  $\langle s, a, r, s' \rangle$  in replay memory  $D$
  - sample random transitions  $\langle s, a, r, s' \rangle$  from replay memory  $D$
  - calculate target for each minibatch transition
    - if  $s'$  is terminal state then  $tt = r$
    - otherwise  $tt = r + \gamma \max_{a'} Q(s', a')$
  - train the  $Q$  network using  $(tt - Q(s, a))^2$  as loss
  - $s = s'$
- until terminated

# Gym

- Gym is a toolkit for developing and comparing reinforcement learning algorithms.
- It supports teaching agents everything from walking to playing games like Pong or Pinball
- It is compatible with any numerical computation library, such as TensorFlow or Theano
  - To get started, you'll need to have Python 3.5+ installed. Simply install gym using pip:  
`pip install gym`

# Other research topics in RL

- Case where state only partially observable
- Design optimal exploration strategies
- Extend to continuous action, state
  - <https://arxiv.org/abs/1509.02971>
- Learn and use  $\hat{\delta} : S \times A \rightarrow S$
- Double Q-learning,  
Prioritized Experience Replay,  
Dueling Network Architecture

# Final comments on Deep RL

- Because our Q-function is initialized randomly, it initially outputs complete garbage
  - We use this garbage (the maximum Q-value of the next state) as targets for the network, only occasionally folding in a tiny reward
- How could it learn anything meaningful at all?
  - The fact is, that it does
- Watching them figure it out is like observing an animal in the wild– a rewarding experience by itself