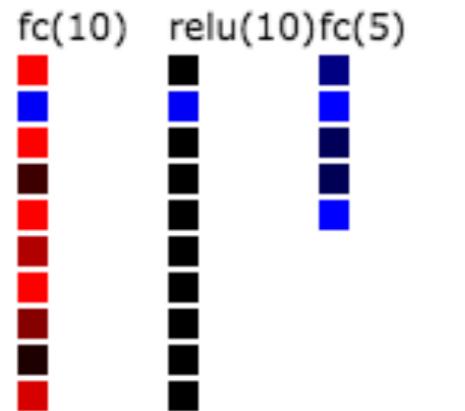
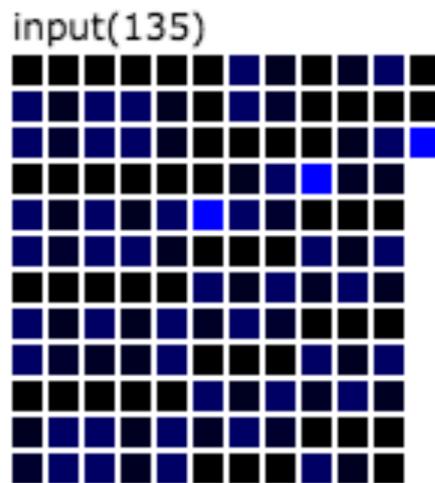




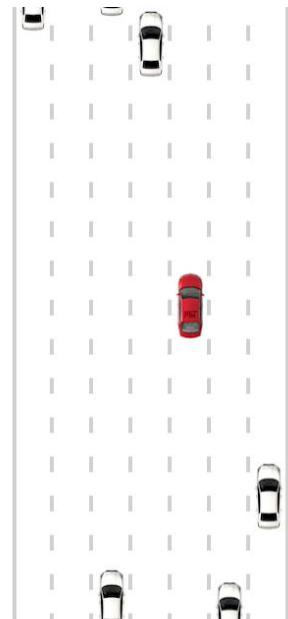
6.S094: Deep Learning for Self-Driving Cars

Learning to Move: Deep Reinforcement Learning for Motion Planning

cars.mit.edu



Administrative



- **Website:** cars.mit.edu
- **Contact Email:** deepcars@mit.edu
- **Required:**
 - Create an account on the website.
 - Follow the tutorial for each of the 2 projects.
- **Recommended:**
 - Ask questions
 - Win competition!



Lex Fridman
Instructor



Benedikt Jenik
TA



William Angell
TA



Spencer Dodd
TA



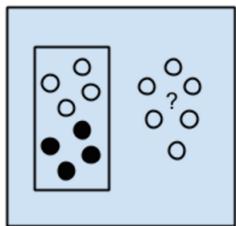
Dan Brown
TA

Schedule

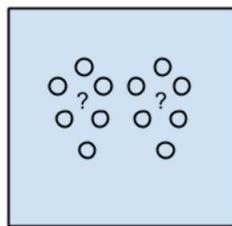
Mon, Jan 9	Introduction to Deep Learning and Self Driving Cars
Tue, Jan 10	Learning to Move: Reinforcement Learning for Motion Planning
	DeepTraffic: Solving Traffic with Deep Reinforcement Learning
Wed, Jan 11	Learning to Drive: End-to-End Learning for the Full Driving Task
	DeepTesla: End-to-End Learning from Human and Autopilot Driving
Thu, Jan 12	Karl Iagnemma: From Research to Reality: Testing Self-Driving Cars on Boston Public Roads
Fri, Jan 13	John Leonard: Mapping, Localization, and the Challenge of Autonomous Driving
Tue, Jan 17	Chris Gerdes: TBD
Wed, Jan 18	Sertac Karaman: Past, Present, and Future of Motion Planning in a Complex World
Thu, Jan 19	Learning to Share: Driver State Sensing and Shared Autonomy
Fri, Jan 20	Eric Daimler: The Future of Artificial Intelligence Research and Development
	Learning to Think: The Road Ahead for Human-Centered Artificial Intelligence

DeepTraffic: Solving Traffic with Deep Reinforcement Learning

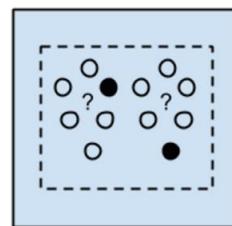
Types of machine learning:



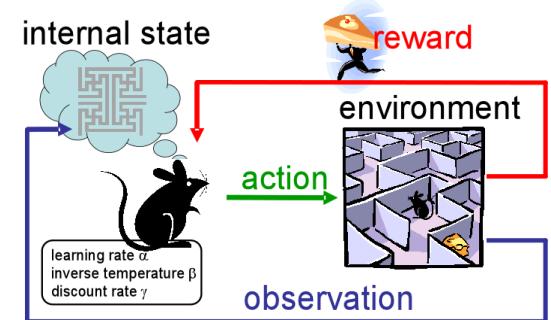
Supervised
Learning



Unsupervised
Learning

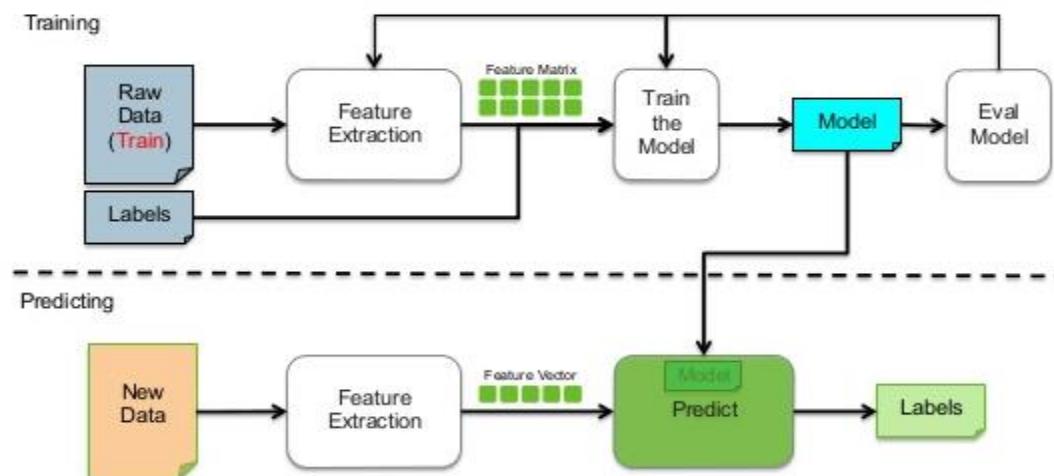


Semi-Supervised
Learning



Reinforcement
Learning

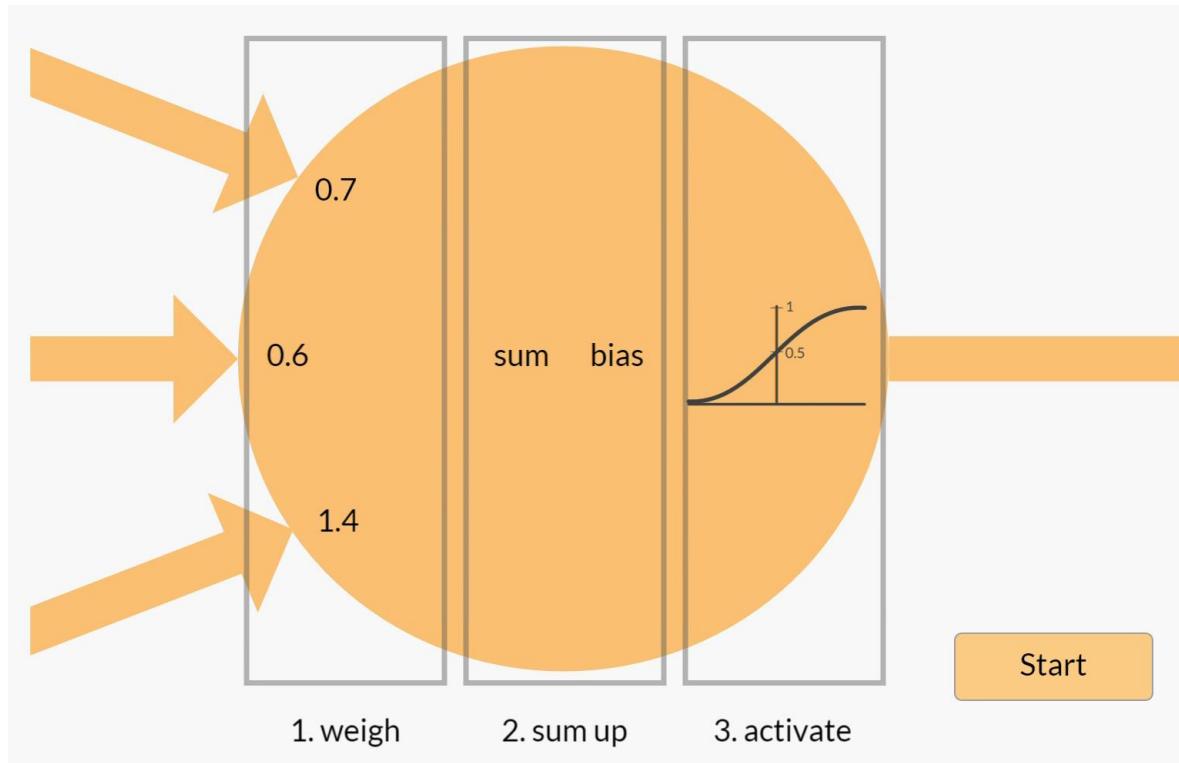
Standard supervised learning pipeline:



Perceptron: Weighing the Evidence

Evidence

Decisions



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

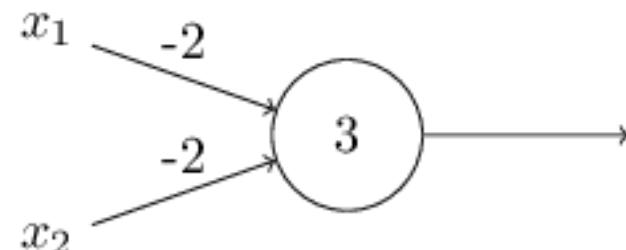
Perceptron: Implement a NAND Gate



$$Q = \text{NOT}(\text{A AND B})$$

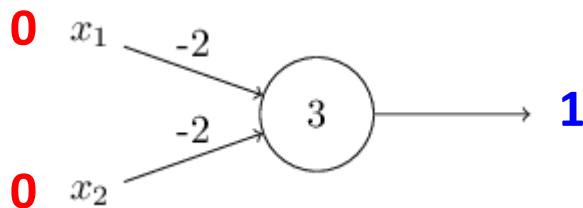
Truth Table

Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0

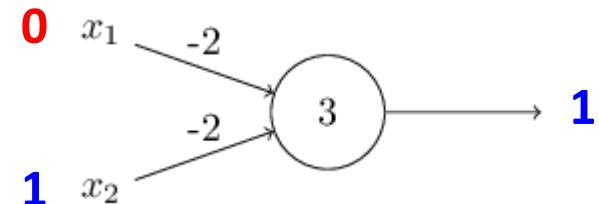


- **Universality:** NAND gates are *functionally complete*, meaning we can build any logical function out of them.

Perceptron: Implement a NAND Gate



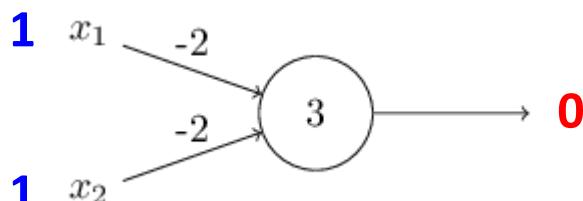
$$(-2)*\mathbf{0} + (-2)*\mathbf{0} + 3 = \mathbf{3}$$



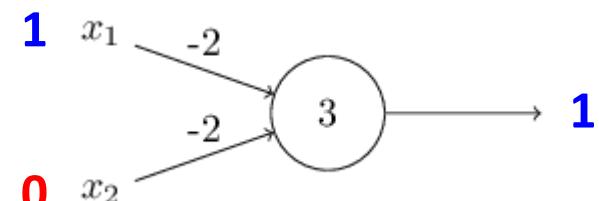
$$(-2)*\mathbf{0} + (-2)*\mathbf{1} + 3 = \mathbf{1}$$

Truth Table

Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0



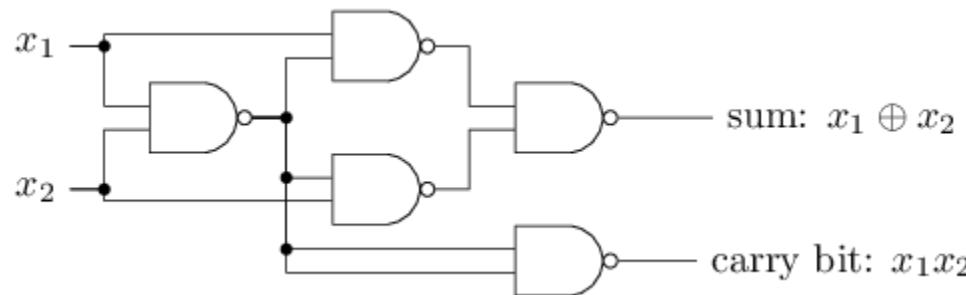
$$(-2)*\mathbf{1} + (-2)*\mathbf{1} + 3 = \mathbf{-1}$$



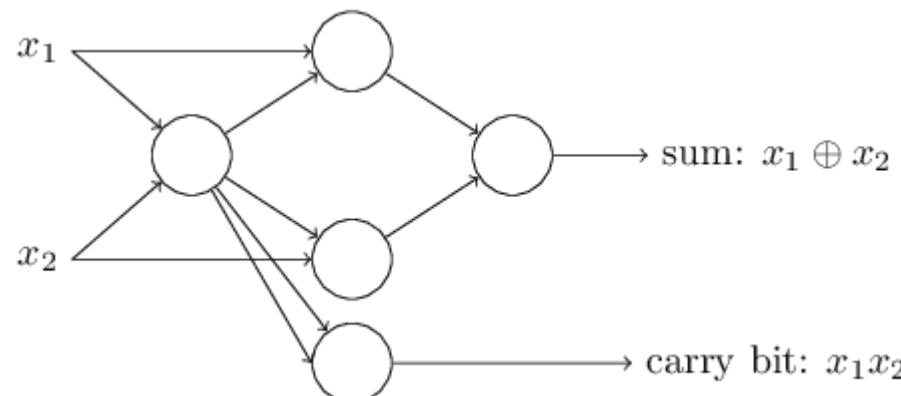
$$(-2)*\mathbf{1} + (-2)*\mathbf{0} + 3 = \mathbf{1}$$

Perceptron > NAND Gate

Both circuits can represent arbitrary logical functions:



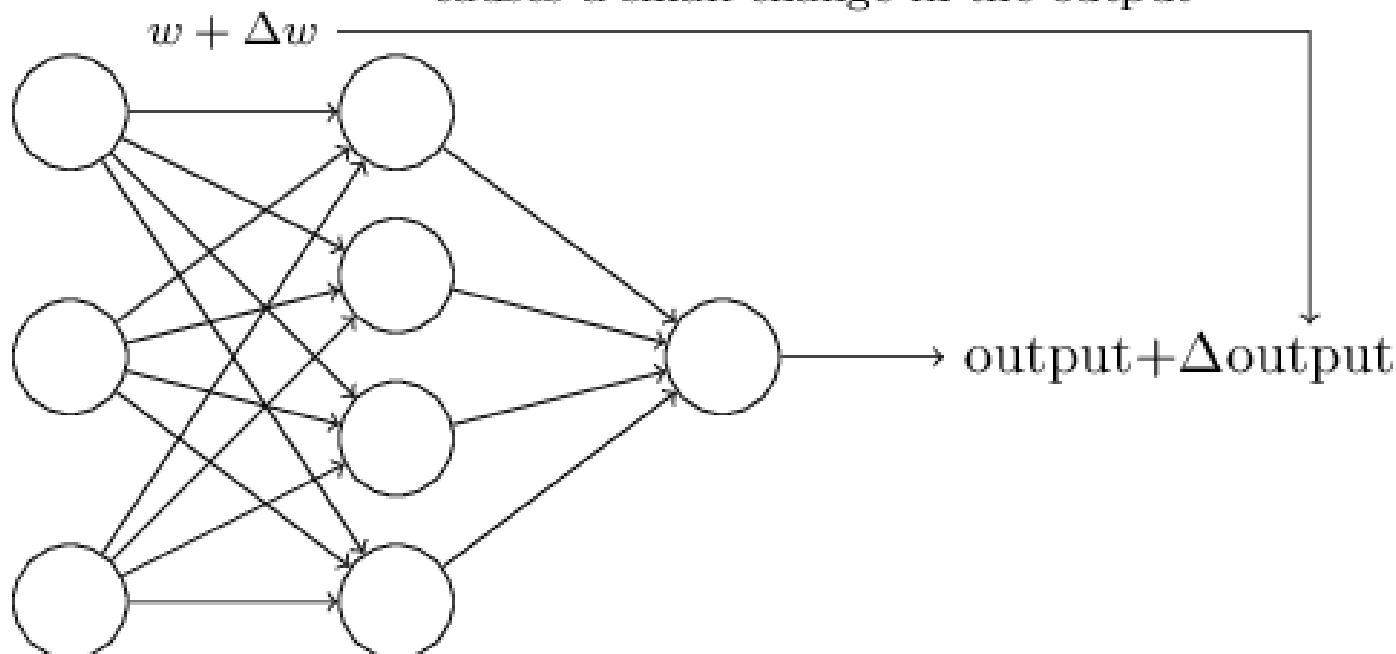
But “perceptron circuits” can learn...



The Process of Learning:

Small Change in Weights → Small Change in Output

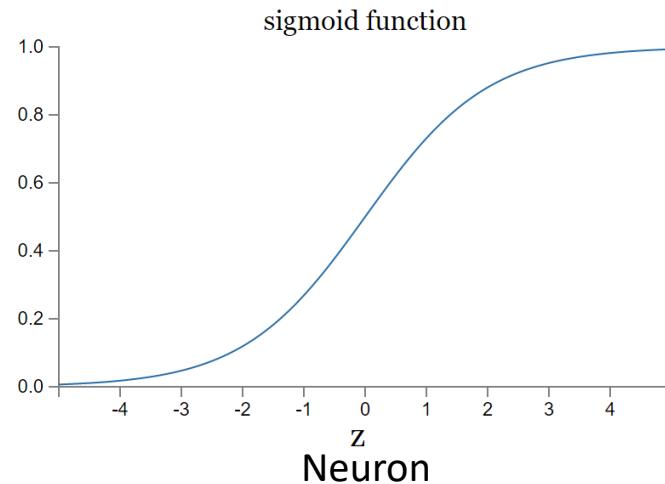
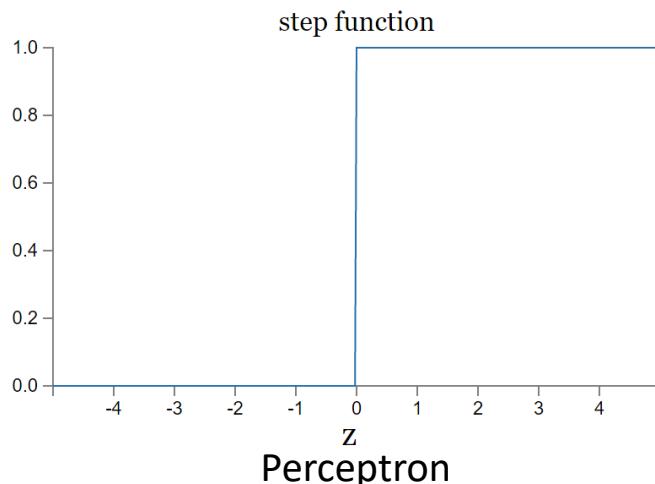
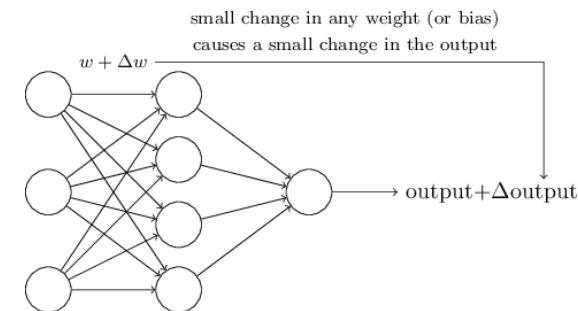
small change in any weight (or bias)
causes a small change in the output



The Process of Learning:

Small Change in Weights → Small Change in Output

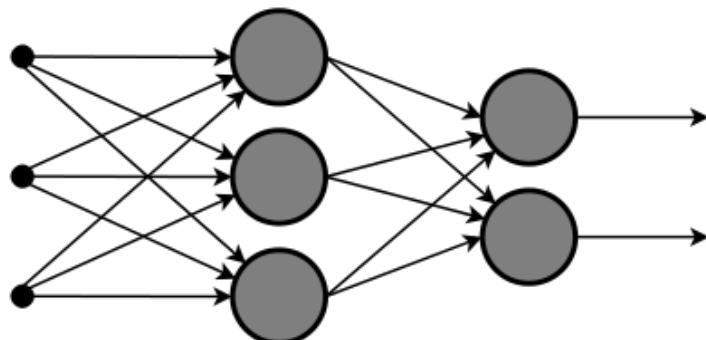
This requires a “smoothness”



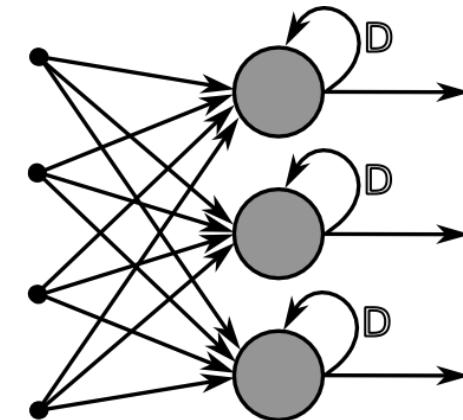
Smoothness of activation function means: **the Δoutput is a linear function of the $\Delta\text{weights}$ and Δbias**

Learning is the process of gradually adjusting the weights to achieve any gradual change in the output.

Combining Neurons into Layers



Feed Forward Neural Network

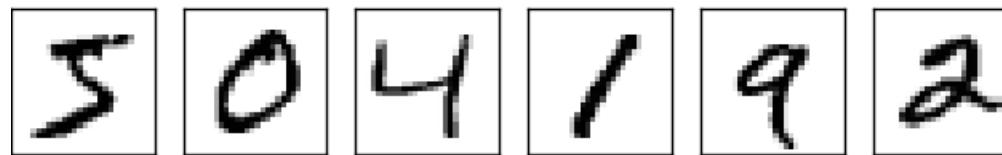


Recurrent Neural Network

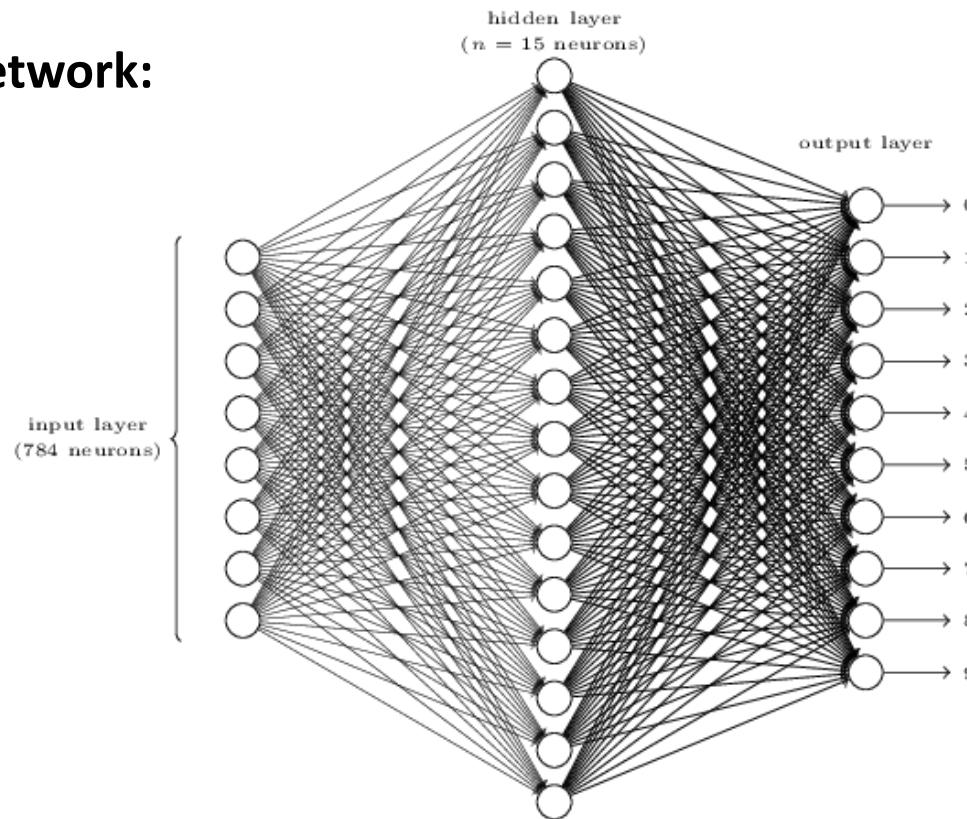
- Have state memory
- Are hard to train

Task: Classify and Image of a Number

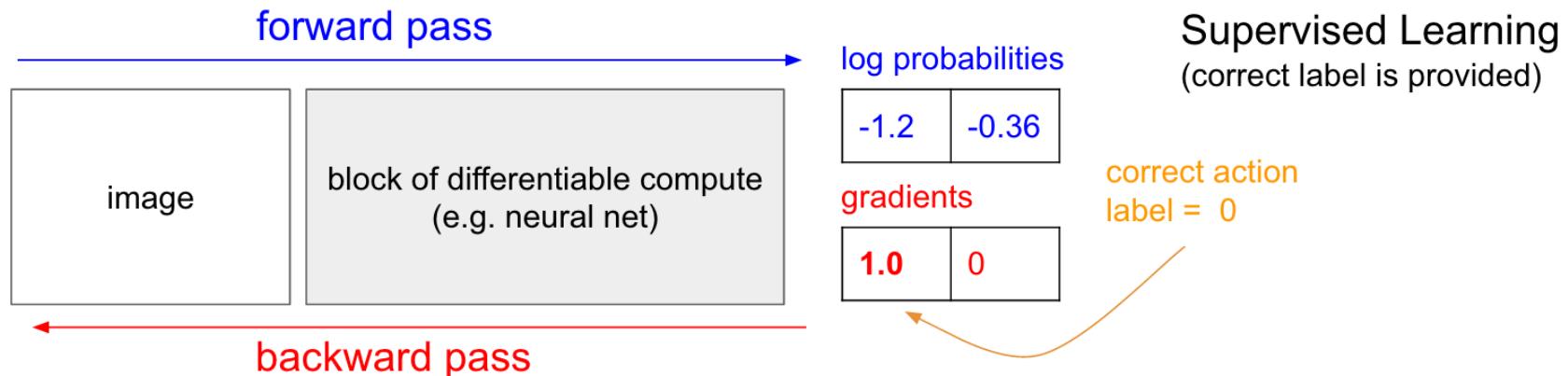
Input:
(28x28)



Network:



Task: Classify and Image of a Number

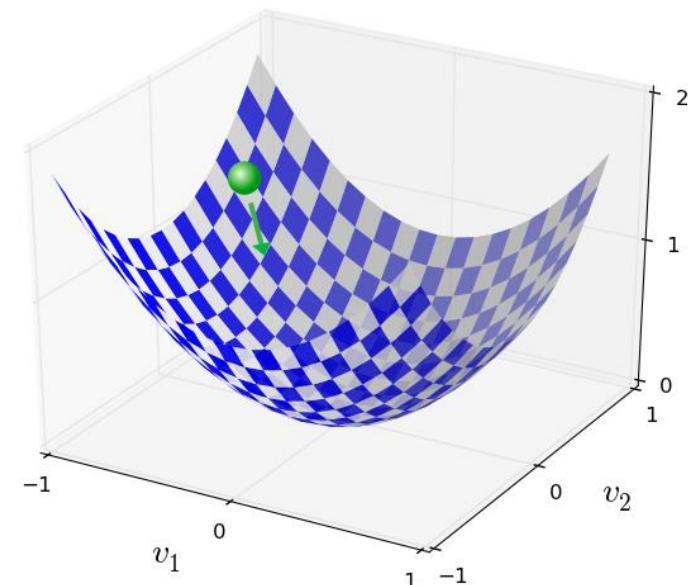


Ground truth for “6”:

$$y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$$

“Loss” function:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$



Philosophical Motivation for Reinforcement Learning

Takeaway from Supervised Learning:

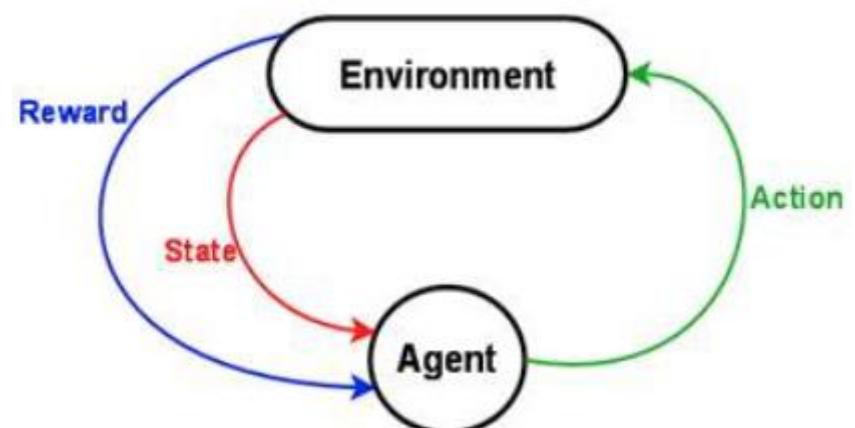
Neural networks are great at memorization and not (yet) great at reasoning.

Hope for Reinforcement Learning:

Brute-force propagation of outcomes to knowledge about states and actions. This is a kind of brute-force “reasoning”.

Agent and Environment

- At each step the agent:
 - Executes action
 - Receives observation (new state)
 - Receives reward
- The environment:
 - Receives action
 - Emits observation (new state)
 - Emits reward



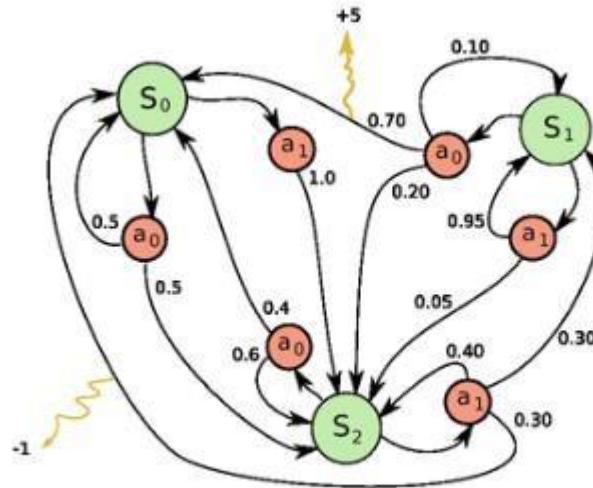
Reinforcement Learning

Reinforcement learning is a general-purpose framework for decision-making:

- An agent operates in an environment: **Atari Breakout**
- An agent has the capacity to **act**
- Each action influences the agent's **future state**
- Success is measured by a **reward** signal
- **Goal** is to select actions to **maximize future reward**



Markov Decision Process



$s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{n-1}, a_{n-1}, r_n, s_n$

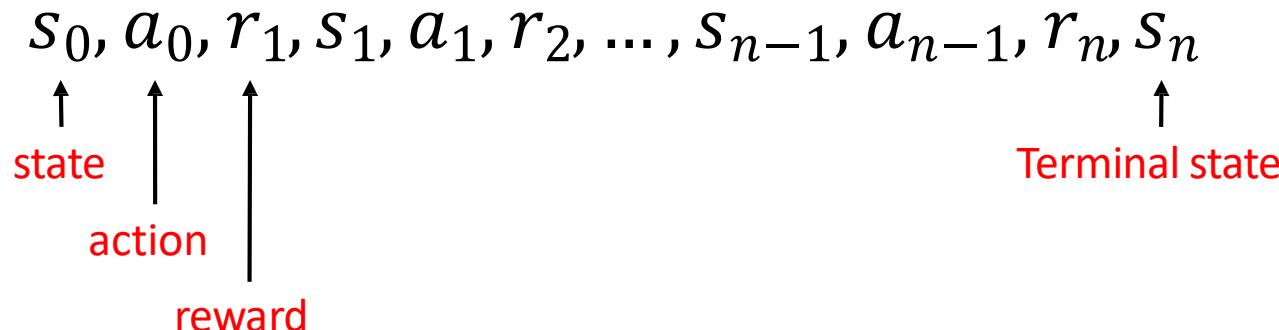
↑
state
↑
action
↑
reward

↑
Terminal state

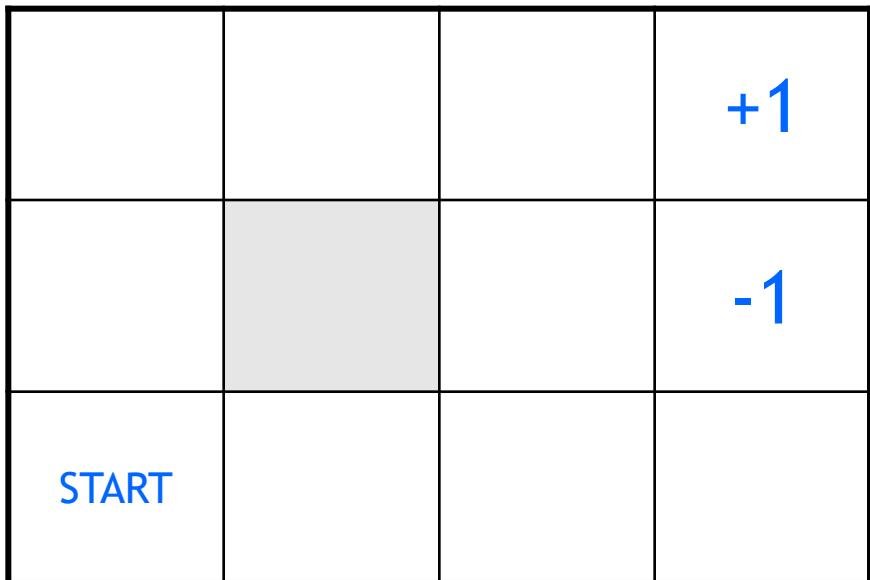
Major Components of an RL Agent

An RL agent may include one or more of these components:

- **Policy:** agent's behavior function
- **Value function:** how good is each state and/or action
- **Model:** agent's representation of the environment



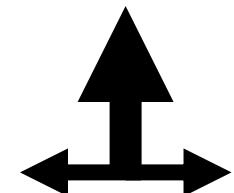
Robot in a Room



actions: UP, DOWN, LEFT, RIGHT

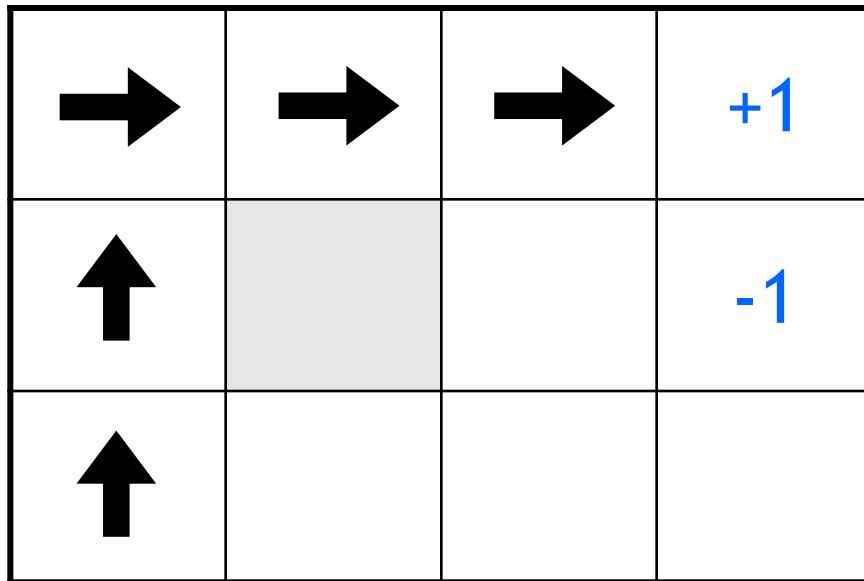
UP

80% move UP
10% move LEFT
10% move RIGHT



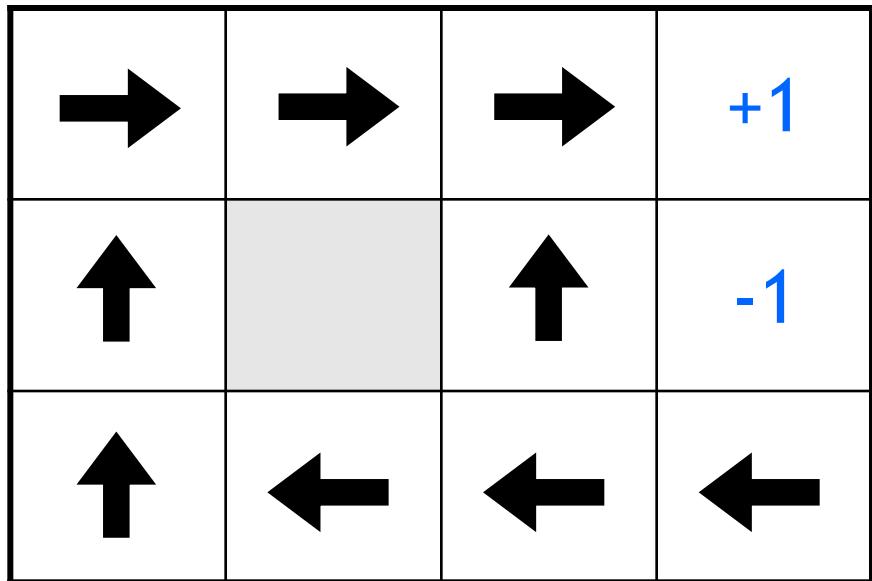
- reward +1 at [4,3], -1 at [4,2]
- reward -0.04 for each step
- what's the strategy to achieve max reward?
- what if the actions were deterministic?

Is this a solution?

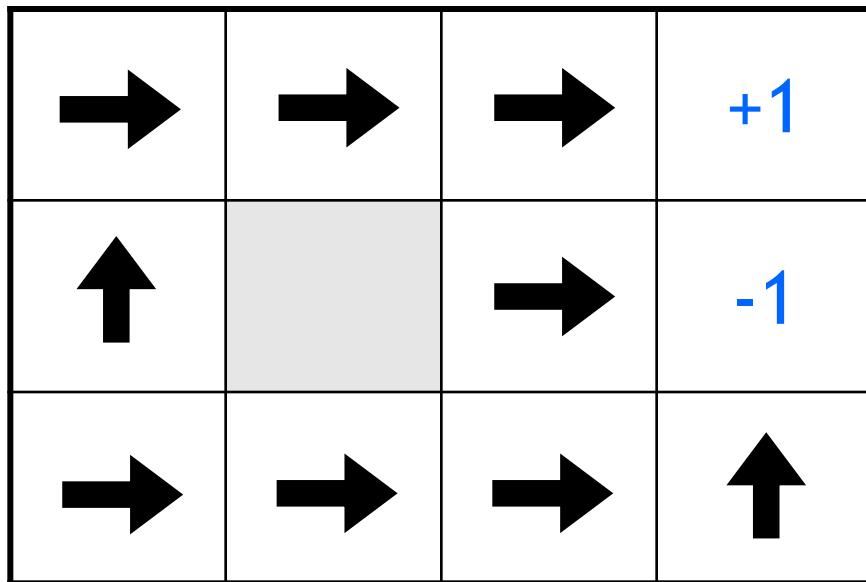


- only if actions deterministic
 - not in this case (actions are stochastic)
- solution/policy
 - mapping from each state to an action

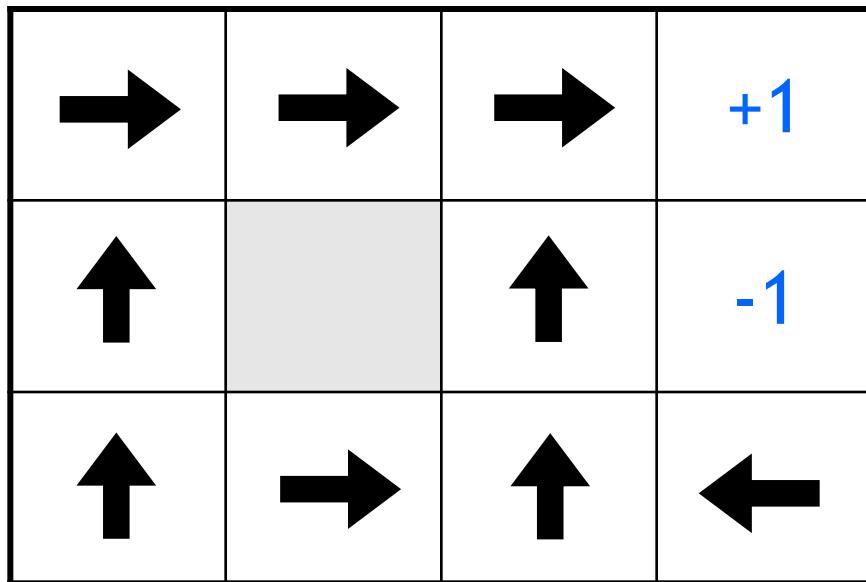
Optimal policy



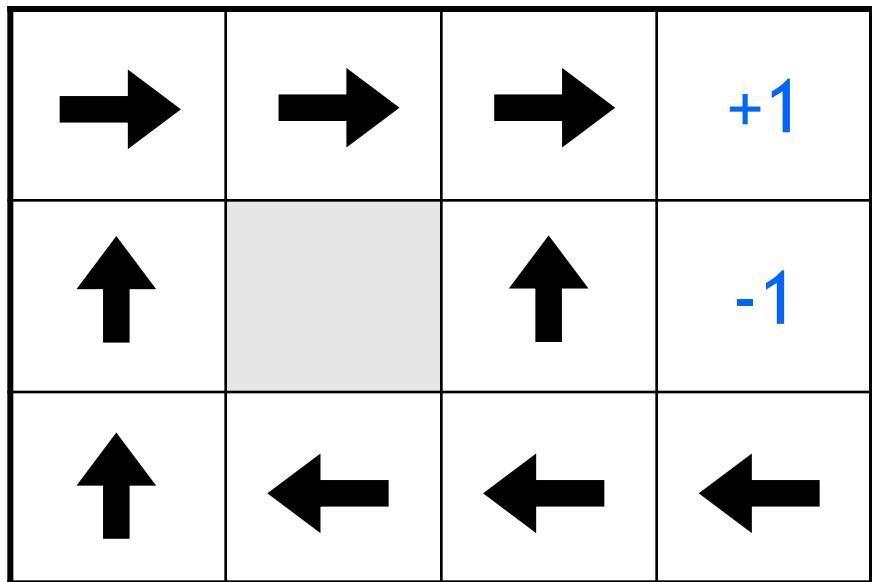
Reward for each step -2



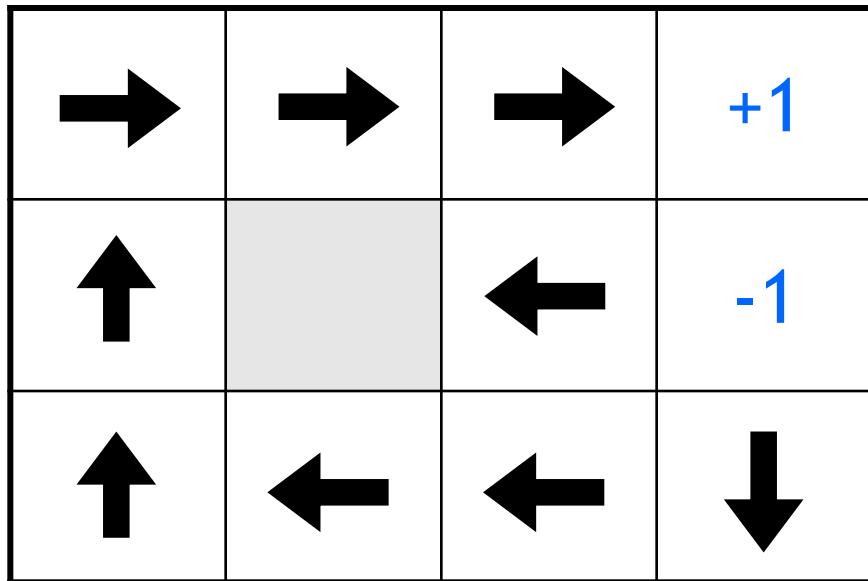
Reward for each step: -0.1



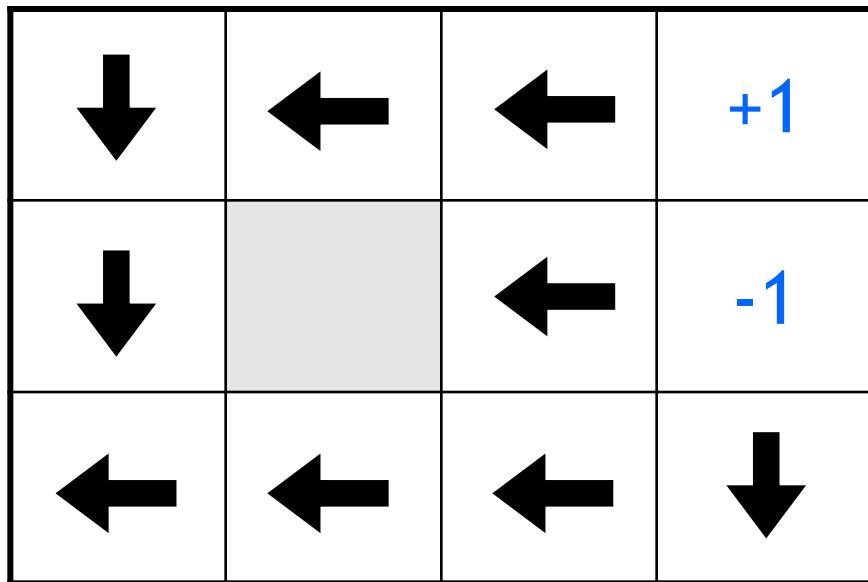
Reward for each step: -0.04



Reward for each step: -0.01



Reward for each step: +0.01



Value Function

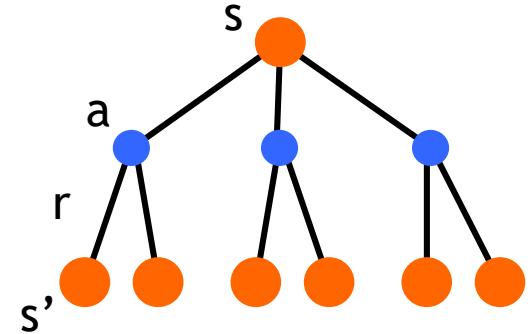
- Future reward $R = r_1 + r_2 + r_3 + \cdots + r_n$
 $R_t = r_t + r_{t+1} + r_{t+2} + \cdots + r_n$
- Discounted future reward (environment is stochastic)

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n \\ &= r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \cdots)) \\ &= r_t + \gamma R_{t+1} \end{aligned}$$

- A good strategy for an agent would be to always choose an action that **maximizes the (discounted) future reward**

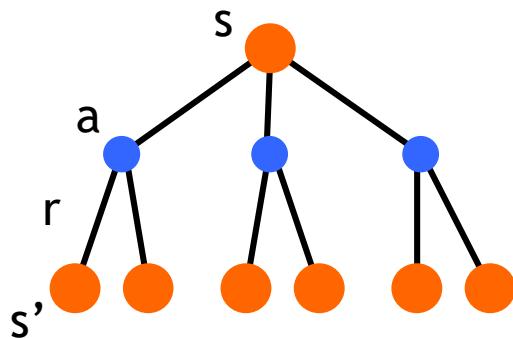
Q-Learning

- State value function: $V^\pi(s)$
 - Expected return when starting in s and following π
- State-action value function: $Q^\pi(s,a)$
 - Expected return when starting in s , performing a , and following π
- Useful for finding the optimal policy
 - Can estimate from experience (Monte Carlo)
 - Pick the best action using $Q^\pi(s,a)$
- Q-learning: off-policy
 - Use any policy to estimate Q that maximizes future reward: $Q(s_t, a_t) = \max R_{t+1}$
 - Q directly approximates Q^* (Bellman optimality equation)
 - Independent of the policy being followed
 - Only requirement: keep updating each (s,a) pair



$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right)$$

Q-Learning



$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right)$$

Learning Rate

Discount Factor

New State

Old State

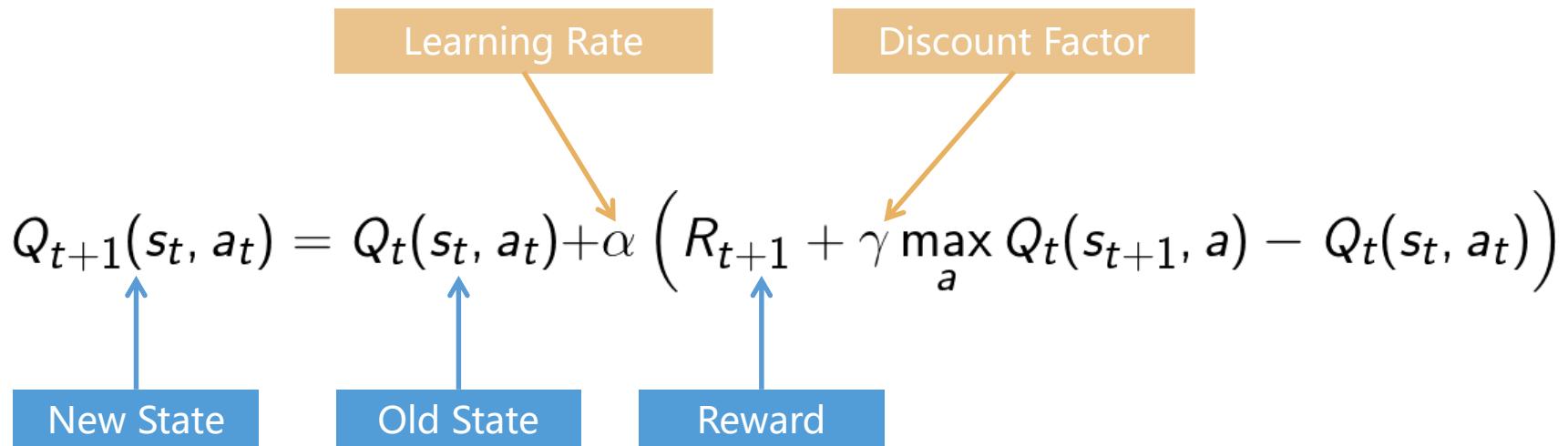
Reward

The diagram shows the Q-learning update rule. It consists of two main parts: the target value $Q_{t+1}(s_t, a_t)$ and the current estimate $Q_t(s_t, a_t)$. The target value is calculated by adding a learning rate α times the difference between the reward R_{t+1} and the discounted maximum Q-value from the next state s_{t+1} for all actions a . The current estimate $Q_t(s_t, a_t)$ is shown below the update rule. Arrows point from the labels 'Learning Rate' and 'Discount Factor' to their respective components in the update rule. Blue arrows point from the labels 'New State', 'Old State', and 'Reward' to the corresponding terms in the update rule.

Exploration vs Exploitation

- Key ingredient of Reinforcement Learning
- Deterministic/greedy policy won't explore all actions
 - Don't know anything about the environment at the beginning
 - Need to try all actions to find the optimal one
- Maintain exploration
 - Use *soft* policies instead: $\pi(s,a) > 0$ (for all s,a)
- ϵ -greedy policy
 - With probability $1-\epsilon$ perform the optimal/greedy action
 - With probability ϵ perform a random action
 - Will keep exploring the environment
 - Slowly move it towards greedy policy: $\epsilon \rightarrow 0$

Q-Learning: Value Iteration



```
initialize Q[num_states,num_actions] 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
```

Q-Learning: Representation Matters

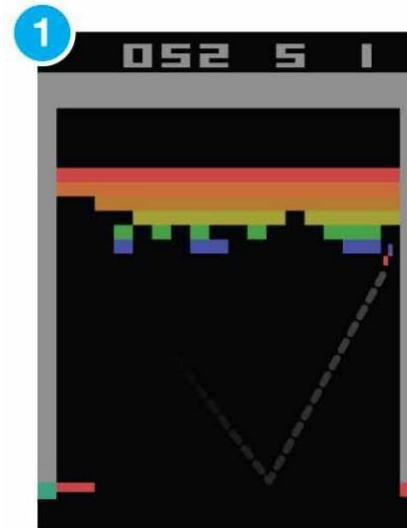
- In practice, Value Iteration is impractical
 - Very limited states/actions
 - Cannot generalize to unobserved states

- Think about the **Breakout** game

- State: screen pixels
 - Image size: **84 × 84** (resized)
 - Consecutive **4** images
 - Grayscale with **256** gray levels

{}

$256^{84 \times 84 \times 4}$ rows in the Q-table!



Philosophical Motivation for Deep Reinforcement Learning

Takeaway from Supervised Learning:

Neural networks are great at memorization and not (yet) great at reasoning.

Hope for Reinforcement Learning:

Brute-force propagation of outcomes to knowledge about states and actions. This is a kind of brute-force “reasoning”.

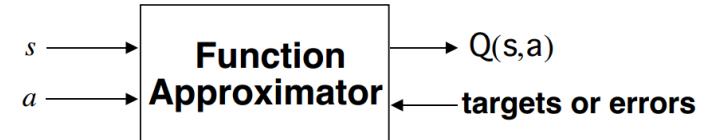
Hope for Deep Learning + Reinforcement Learning:

General purpose artificial intelligence through efficient generalizable learning of the optimal thing to do given a formalized set of actions and states (possibly huge).

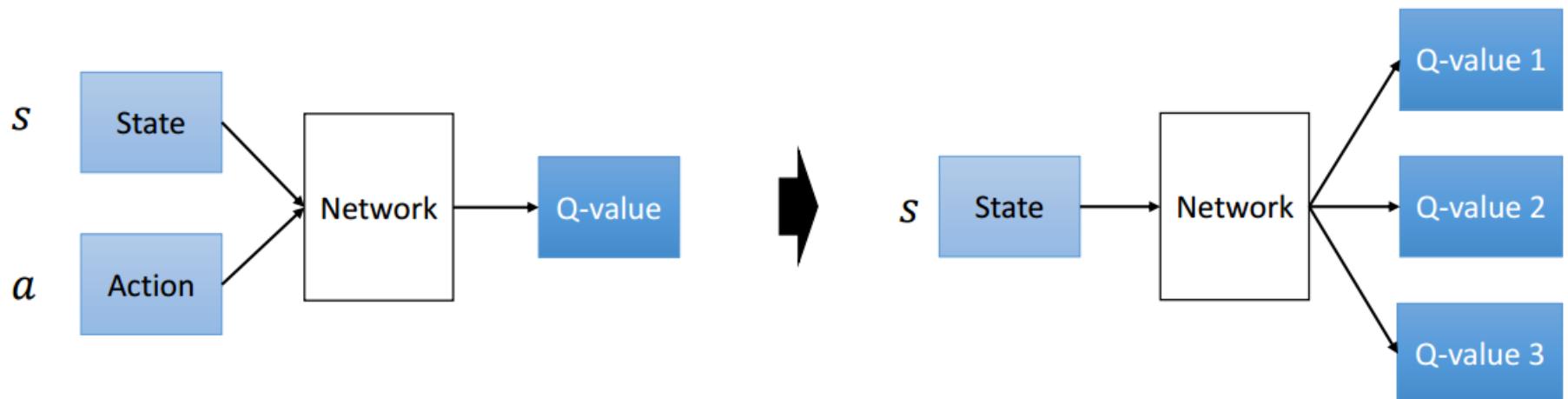
Deep Q-Learning

Use a function (with parameters) to approximate the Q-function

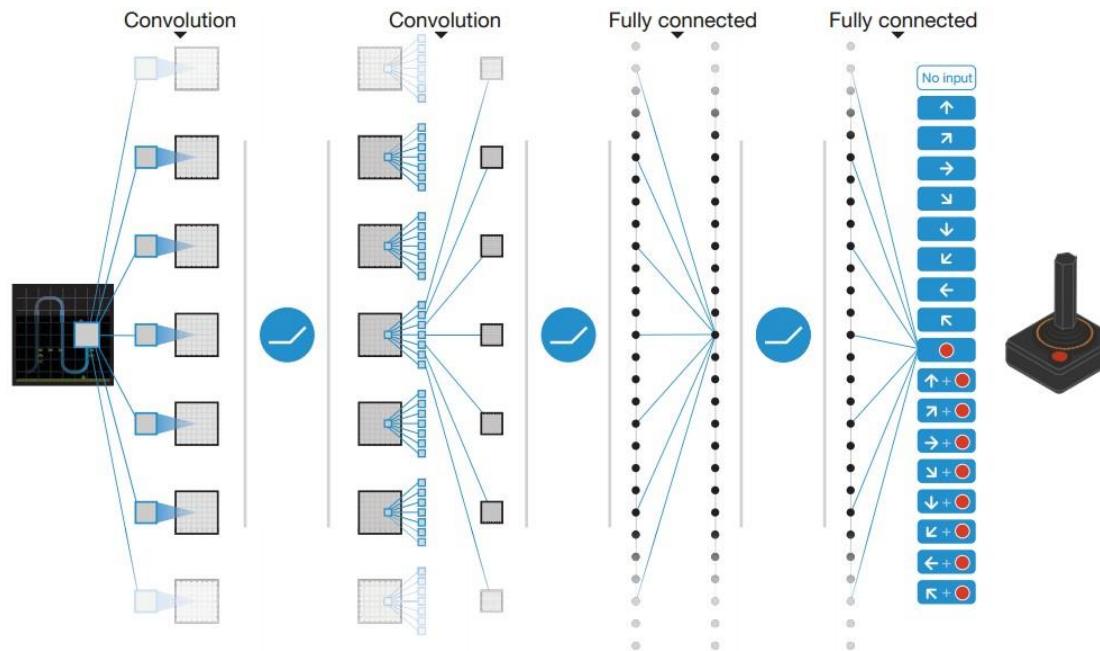
- Linear
- Non-linear: **Q-Network**



$$Q(s, a; \theta) \approx Q^*(s, a)$$



Deep Q-Network: Atari



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

Mnih et al. "Playing atari with deep reinforcement learning." 2013.

Deep Q-Network Training

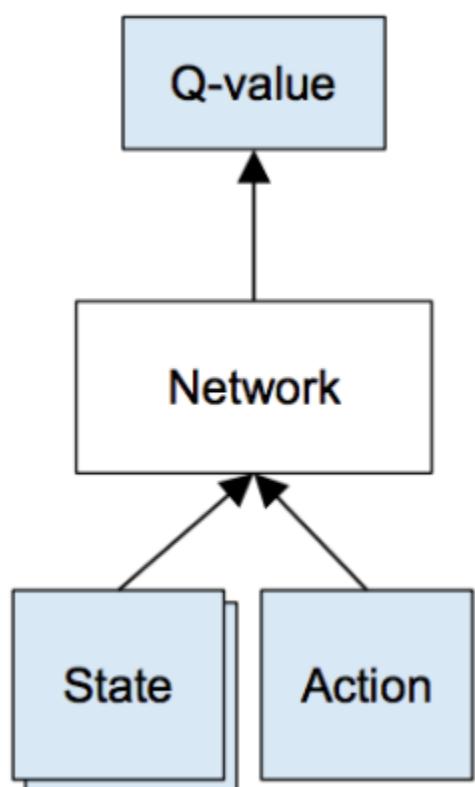
- Bellman Equation:

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

- Loss function (squared error):

$$L = \mathbb{E}[(\underbrace{r + \gamma \max_{a'} Q(s', a')}_{\text{target}} - Q(s, a))^2]$$

Deep Q-Network Training



Given a transition $\langle s, a, r, s' \rangle$, the Q-table update rule in the previous algorithm must be replaced with the following:

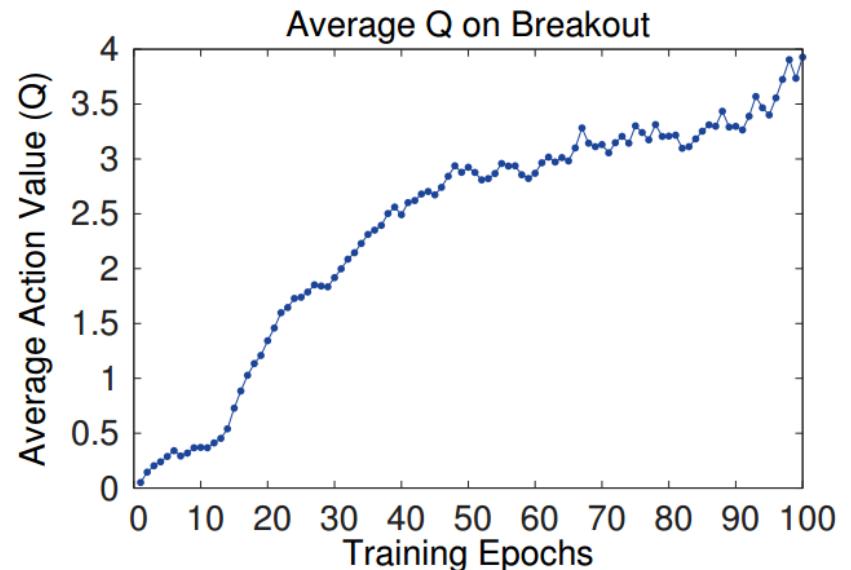
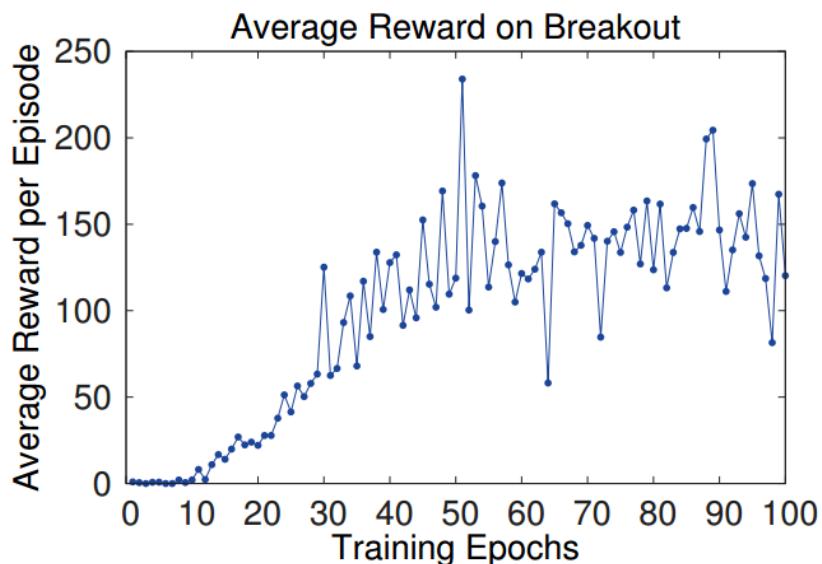
- Do a feedforward pass for the current state s to get **predicted Q-values for all actions**
- Do a feedforward pass for the next state s' and calculate maximum overall network outputs $\max_{a'} Q(s', a')$
- Set Q-value target for action to $r + \gamma \max_{a'} Q(s', a')$ (use the max calculated in 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.
- Update the weights using backpropagation.

Exploration vs Exploitation

- Key ingredient of Reinforcement Learning
- Deterministic/greedy policy won't explore all actions
 - Don't know anything about the environment at the beginning
 - Need to try all actions to find the optimal one
- Maintain exploration
 - Use *soft* policies instead: $\pi(s,a) > 0$ (for all s,a)
- ϵ -greedy policy
 - With probability $1-\epsilon$ perform the optimal/greedy action
 - With probability ϵ perform a random action
 - Will keep exploring the environment
 - Slowly move it towards greedy policy: $\epsilon \rightarrow 0$

Atari Breakout

- A few tricks needed, most importantly: experience replay



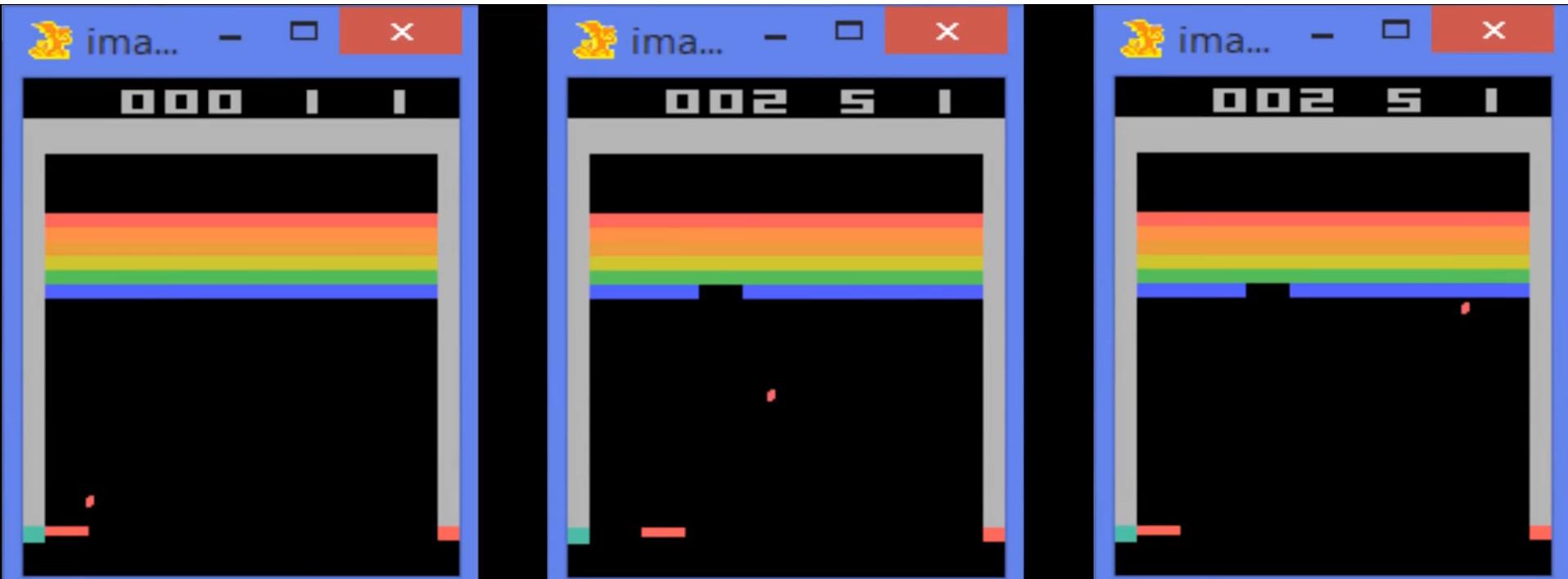
Deep Q-Learning Algorithm

```
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  $a = \text{argmax}_{a'} Q(s, a')$ 
    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  $\langle ss, aa, rr, ss' \rangle$  from replay memory  $D$ 
    calculate target for each minibatch transition
        if  $ss'$  is terminal state then  $tt = rr$ 
        otherwise  $tt = rr + \gamma \max_{a'} Q(ss', aa')$ 
    train the  $Q$  network using  $(tt - Q(ss, aa))^2$  as loss

     $s = s'$ 
until terminated
```

Atari Breakout

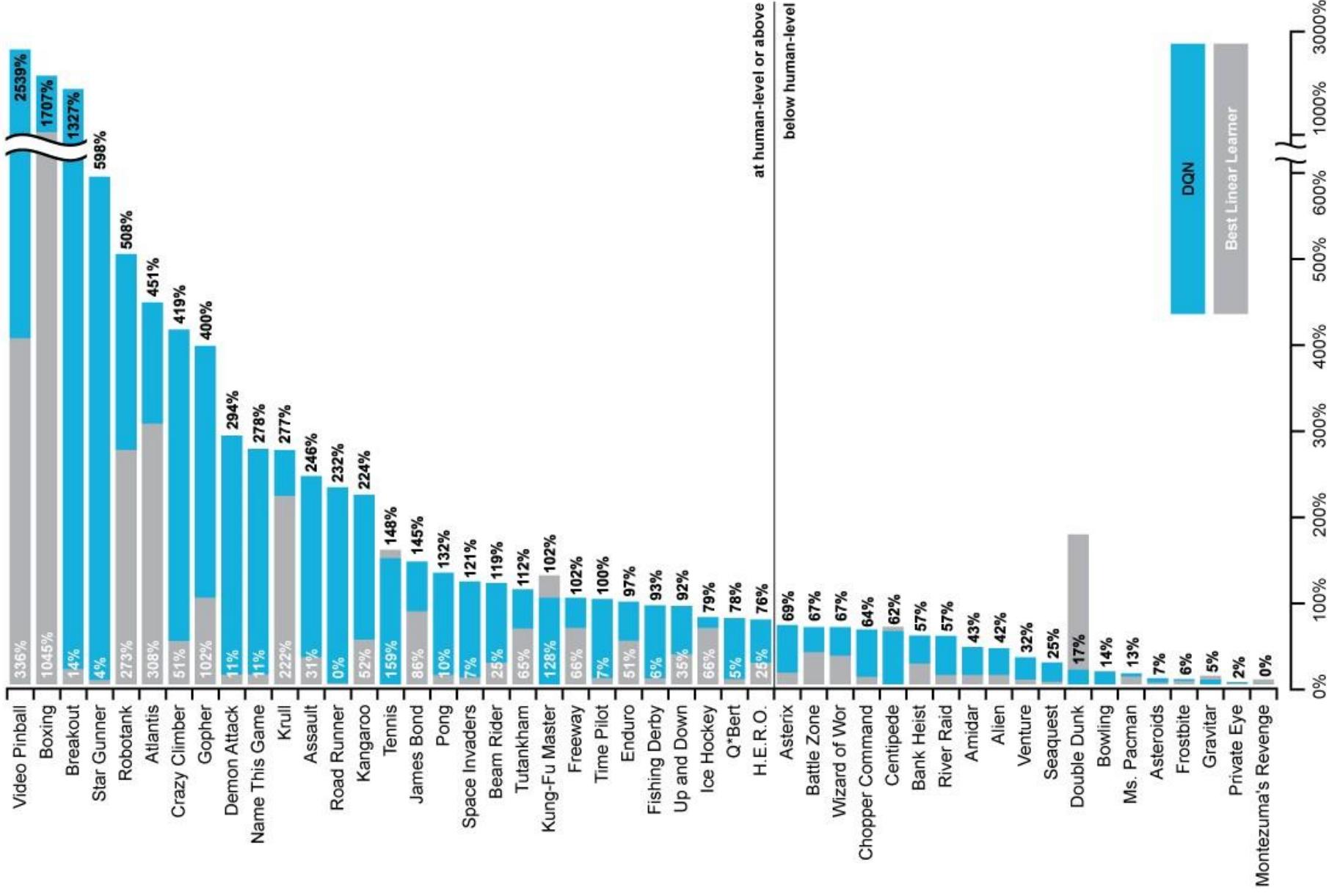


After
10 Minutes
of Training

After
120 Minutes
of Training

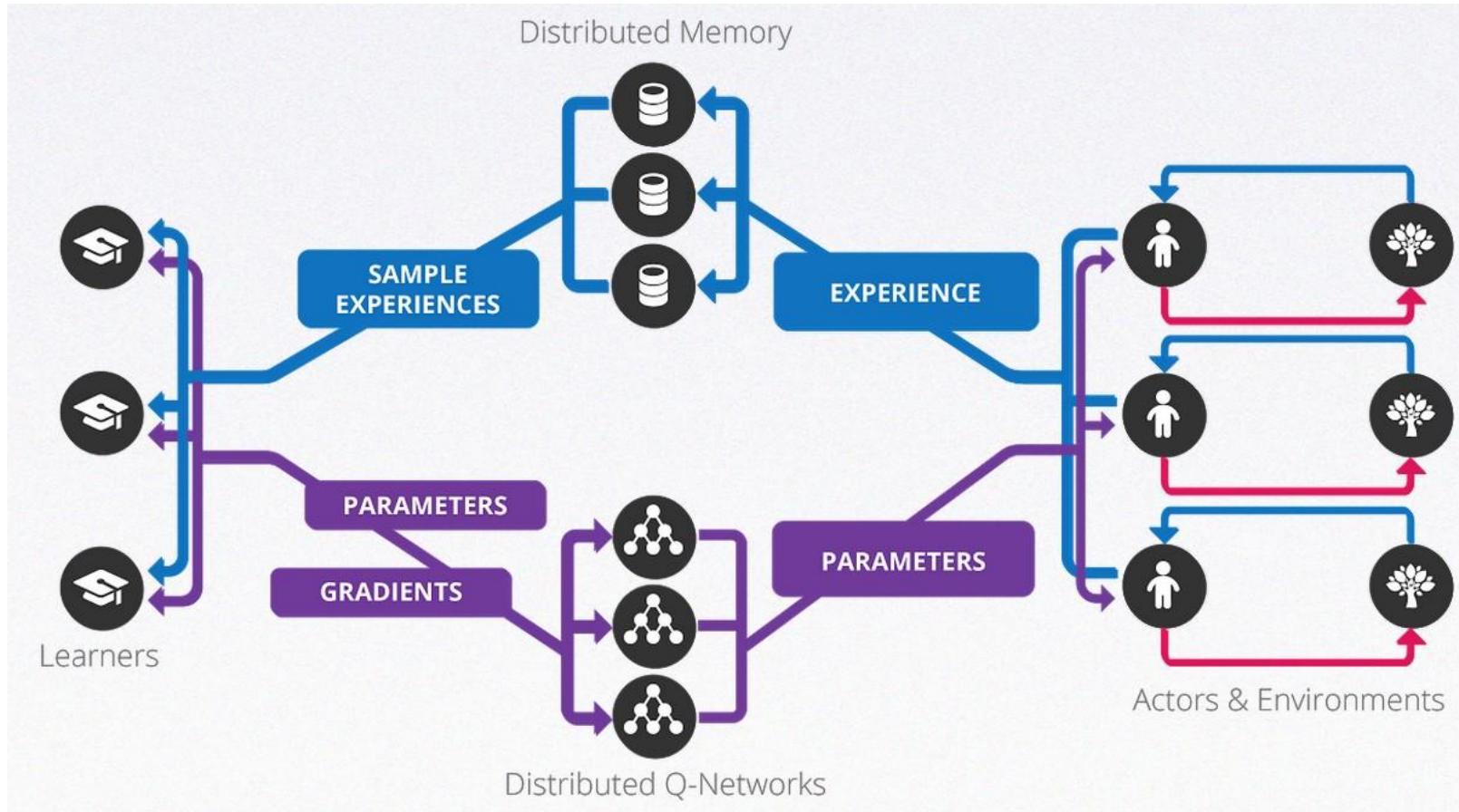
After
240 Minutes
of Training

DQN Results in Atari



Gorila

(General Reinforcement Learning Architecture)



- 10x faster than Nature DQN on 38 out of 49 Atari games
- Applied to recommender systems within Google

Nair et al. "Massively parallel methods for deep reinforcement learning." (2015).

The Game of Traffic

Open Question (Again):

Is driving closer to **chess** or to **everyday conversation?**



DeepTraffic: Solving Traffic with Deep Reinforcement Learning

The screenshot shows the DeepTraffic application interface. On the left, there is a simulation grid showing several cars on a road. A sidebar displays "Speed: 80 mph" and "Cars Passed: 290". Below the grid, there are dropdown menus for "Road Overlay:" (set to "None") and "Simulation Speed:" (set to "Normal"). In the center, there is a code editor window containing a snippet of JavaScript code. Below the code editor are four buttons: "Apply Code/Reset Net", "Save Code/Net to File", "Load Code/Net from File", and "Submit Model to Competition". Underneath these buttons is a line graph showing a performance metric over time, with a red line starting at ~-1.72 and rising to ~1.45. A blue button labeled "Start Evaluation Run" is positioned below the graph. At the bottom, there is a diagram titled "Value Function Approximating Neural Network:" showing the architecture: input(135) -> fc(10) -> relu(10) -> fc(5) -> regression(5).

```
1 //<![CDATA[
2 // a few things don't have var in front of them - they update already
3 // existing variables the game needs
4 lanesSide = 1; ///
5 patchesAhead = 10; //13;
6 patchesBehind = 0; //7;
7 trainIterations = 100000;
8
9 // begin from convnetjs example
10 var num_inputs = (lanesSide * 2 + 1) * (patchesAhead + patchesBehind);
11 var num_actions = 5;
12 var temporal_window = 3; //1 // amount of temporal memory. 0 = agent lives
in-the-moment :)
13 var network_size = num_inputs * temporal_window + num_actions *
```

Apply Code/Reset Net Save Code/Net to File Load Code/Net from File Submit Model to Competition

Start Evaluation Run

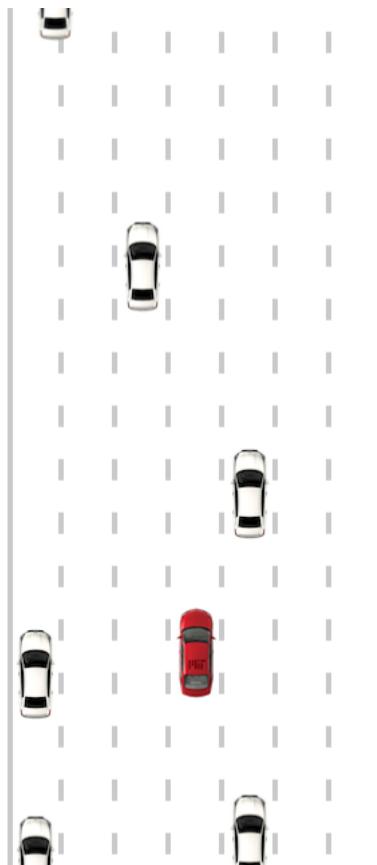
Value Function Approximating Neural Network:

input(135) fc(10) relu(10)fc(5) regression(5)

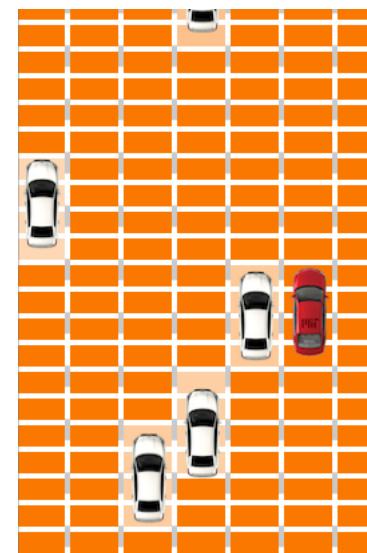
- **Goal:** Achieve the highest average speed over a long period of time.
- **Requirement for Students:** Follow tutorial to achieve a speed of 65mph

The Road, The Car, The Speed

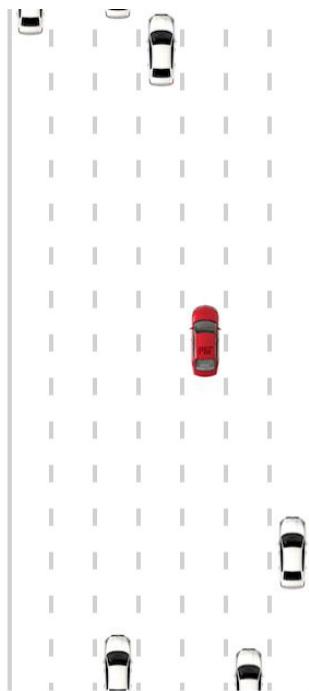
Speed:
47 mph
Cars Passed:
5



State Representation:



Simulation Speed



Road Overlay:

None

Simulation Speed:

Normal



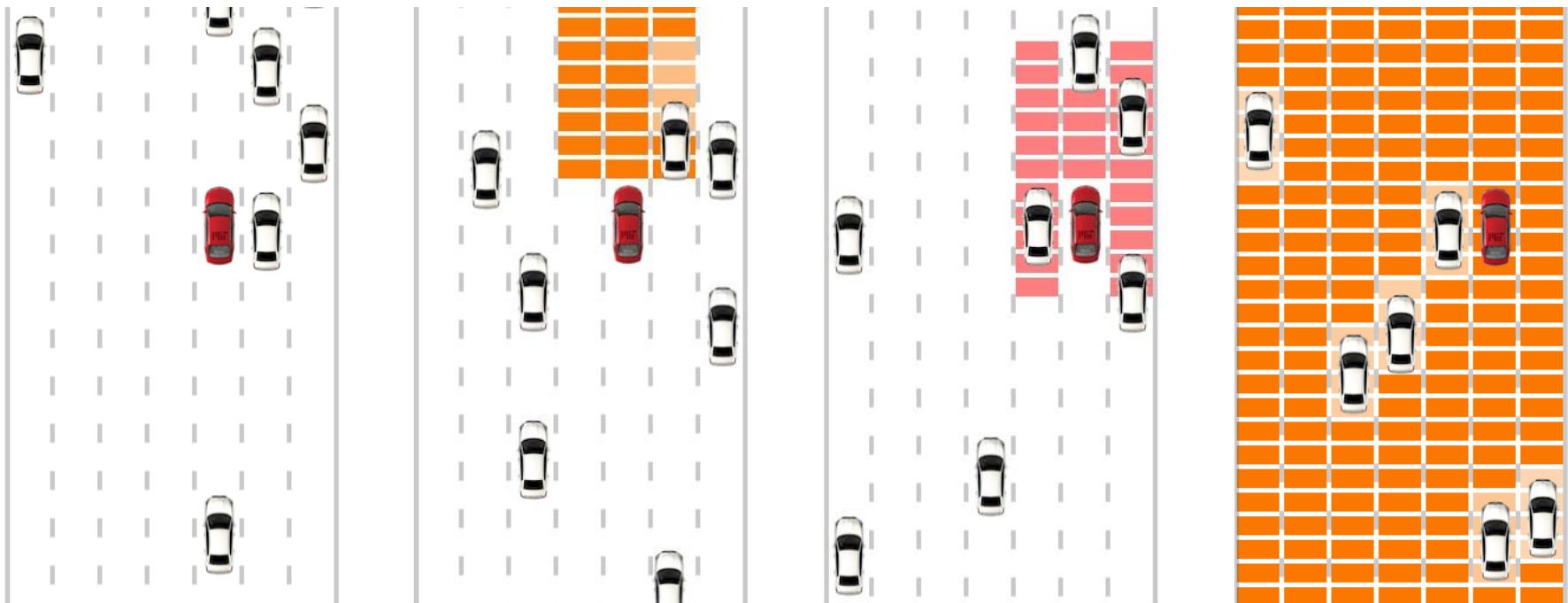
Road Overlay:

None

Simulation Speed:

Fast

Display Options



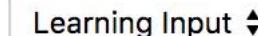
Road Overlay:

None



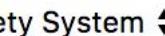
Road Overlay:

Learning Input



Road Overlay:

Safety System

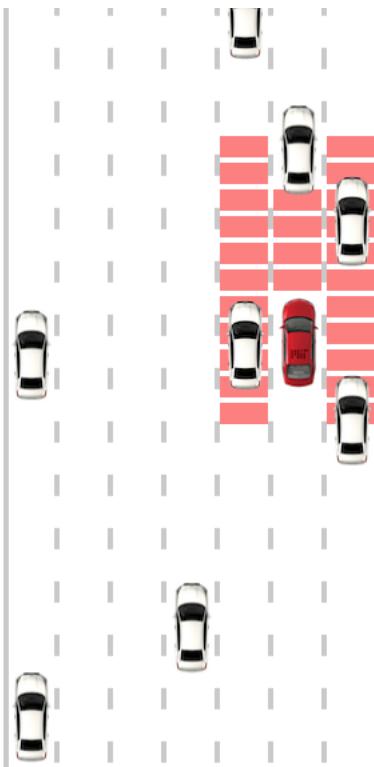


Road Overlay:

Full Map



Safety System



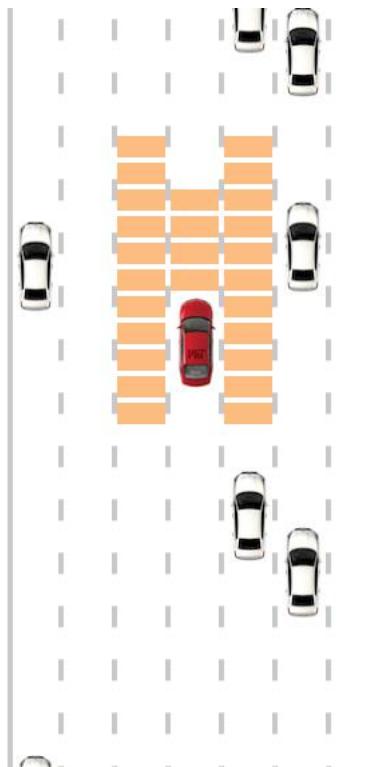
Road Overlay:

Safety System ▾



Road Overlay:

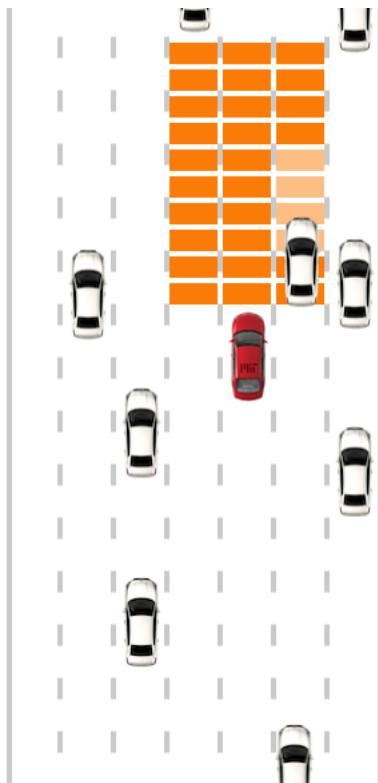
Safety System ▾



Road Overlay:

Safety System ▾

Driving / Learning

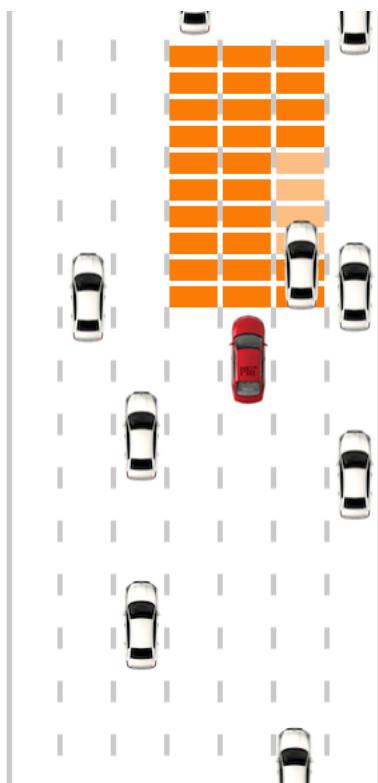


Road Overlay:

Learning Input ▾

```
learn = function (state, lastReward) {  
    brain.backward(lastReward);  
    var action = brain.forward(state);  
    return action;  
}  
  
var noAction = 0;  
var accelerateAction = 1;  
var decelerateAction = 2;  
var goLeftAction = 3;  
var goRightAction = 4;
```

Learning Input

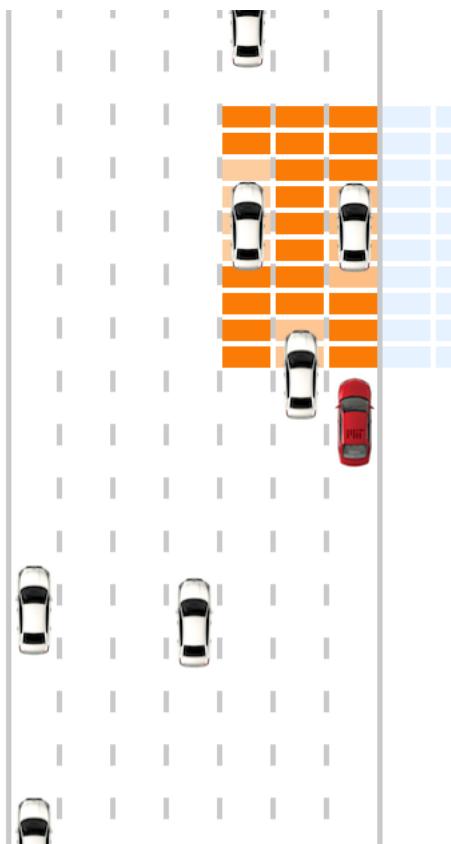


Road Overlay:

Learning Input ▾

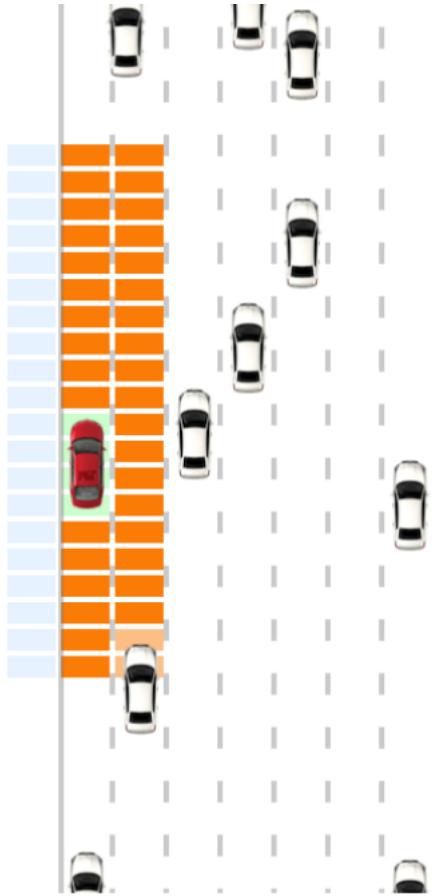
```
lanesSide = 1;  
patchesAhead = 10;  
patchesBehind = 0;
```

Learning Input



```
lanesSide = 2;  
patchesAhead = 10;  
patchesBehind = 0;
```

Learning Input



```
lanesSide = 1;  
patchesAhead = 10;  
patchesBehind = 10;
```

Evaluation

- Scoring: Average Speed
- Method:
 - Collect average speed
 - Ten runs, about 30 (simulated) minutes of game each
 - Result: median speed of the 10 runs
- Done server side after you submit
 - (no cheating possible! (we also look at the code ...))
- You can try it locally to get an estimate
 - Uses exactly the same evaluation procedure/code
 - But: some influence of randomness
 - Our number is what counts in the end!

Evaluation (Locally)

Start Evaluation Run

...



...

Average speed: 51 mph

Coding/Changing the Net Layout

```
1
2 //<![CDATA[
3 // a few things don't have var in front of them - they update already
existing variables the game needs
4 lanesSide = 1;
5 patchesAhead = 10;
6 patchesBehind = 10;
7 trainIterations = 100000;
8
9 // begin from convnetjs example
10 var num_inputs = (lanesSide * 2 + 1) * (patchesAhead + patchesBehind);
11 var num_actions = 5;
12 var temporal_window = 3; //1 // amount of temporal memory. 0 = agent lives
in-the-moment :)
13 var network_size = num_inputs * temporal_window + num_actions *</pre>
```

Apply Code/Reset Net

Watch out: kills trained state!

Training

- Done on separate thread (Web Workers, yay!)
 - Separate simulation, resets, state, etc.
 - A lot faster (1000 fps +)
- Net state gets shipped to the main simulation from time to time
 - You get to see the improvements/learning live



Training

```
trainIterations = 100000;
```

Run Training

...



Loading/Saving

Save Code/Net to File

- Danger: Overwrites all of your code and the trained net

Load Code/Net from File

Submitting

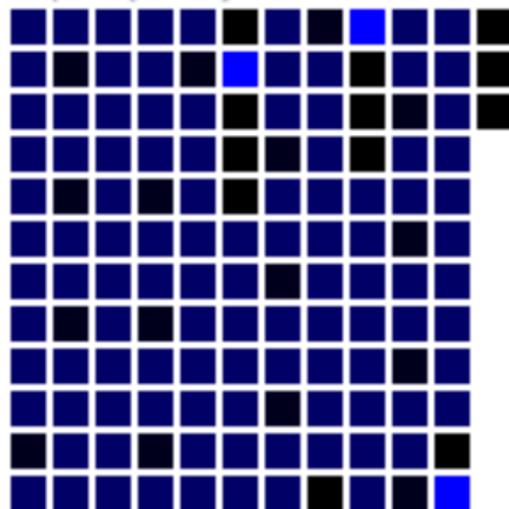
Submit Model to Competition

- Submits your code and the trained net state
 - **Make sure you ran training!**
- Adds your code to the end of a queue
 - Gets evaluated some time (no promises here)
- You can resubmit as often as you like
 - If your code wasn't evaluated yet it we still remove it from the queue (and move you to the end)
 - The highest/most recent???? score counts.

ConvNetJS / The Actual Deep Learning Part

Value Function Approximating Neural Network:

input(135)



fc(10)



relu(10)fc(5)



regression(5)

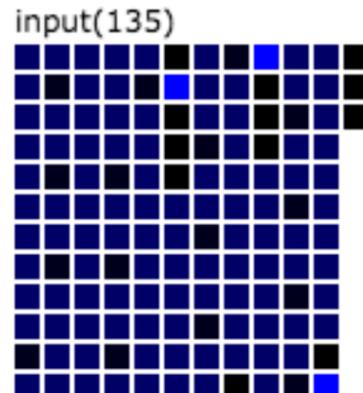


ConvNetJS: Settings

```
var num_inputs = (lanesSide * 2 + 1) * (patchesAhead + patchesBehind);
var num_actions = 5;
var temporal_window = 3;
var network_size = num_inputs * temporal_window + num_actions *
temporal_window + num_inputs;
```

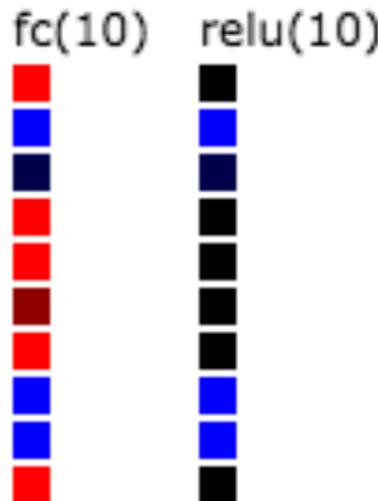
ConvNetJS: Input

```
var layer_defs = [];
layer_defs.push({
    type: 'input',
    out_sx: 1,
    out_sy: 1,
    out_depth: network_size
});
```



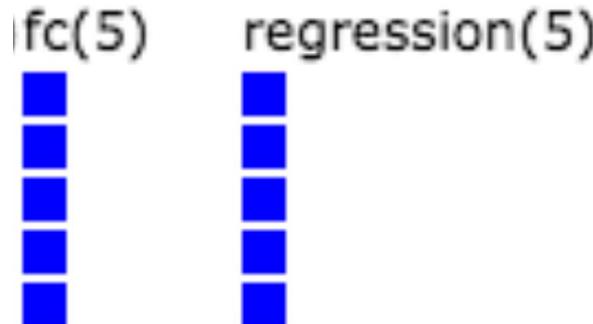
ConvNetJS: Hidden / Fully Connected Layers

```
layer_defs.push({  
    type: 'fc',  
    num_neurons: 10,  
    activation: 'relu'  
});
```



ConvNetJS: Output Layer

```
layer_defs.push({  
    type: 'regression',  
    num_neurons: num_actions  
});
```



ConvNetJS: Options

```
var opt = {};
opt.temporal_window = temporal_window;
opt.experience_size = 3000;
opt.start_learn_threshold = 500;
opt.gamma = 0.7;
opt.learning_steps_total = 10000;
opt.learning_steps_burnin = 1000;
opt.epsilon_min = 0.0;
opt.epsilon_test_time = 0.0;
opt.layer_defs = layer_defs;
opt.tdtrainer_options = {
    learning_rate: 0.001, momentum: 0.0, batch_size: 64, l2_decay: 0.01
};

brain = new deepqlearn.Brain(num_inputs, num_actions, opt);
```

ConvNetJS: Learning

```
learn = function (state, lastReward) {  
    brain.backward(lastReward);  
    var action = brain.forward(state);  
  
    draw_net();  
    draw_stats();  
  
    return action;  
}
```

Technical Details (How We Built The Game)

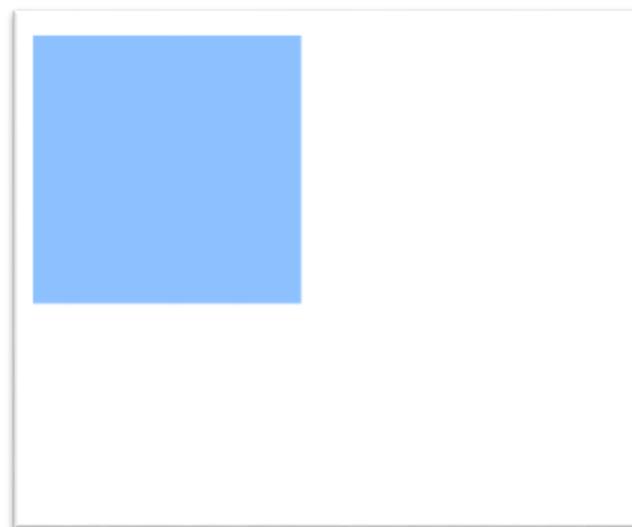
- Monaco Editor
- HTML5 Canvas
- Web Workers

Monaco Editor

```
<script src="monaco-editor/min/vs/loader.js"></script>
<script>
  require.config({
    paths: {
      'vs': 'monaco-editor/min/vs'
    }
  });
  require(['vs/editor/editor.main'], function () {
    editor = monaco.editor.create(document.getElementById('container'), {
      value: "some code ...",
      language: 'javascript',
      wrappingColumn: 75,
    });
  });
</script>
```

HTML5 Canvas

```
<canvas id="canvas" width="400" height="700"></canvas>
<script>
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.fillStyle = 'rgba(0,120,250,0.5)';
  ctx.fillRect(0, 0, 100, 100);
</script>
```



Web Workers

```
//main.js
if (window.Worker) {
    var myWorker = new Worker("worker.js");
    myWorker.onmessage = function (e) {
        console.log(e.data);
    };
}

//worker.js
postMessage("Hello world!");
```

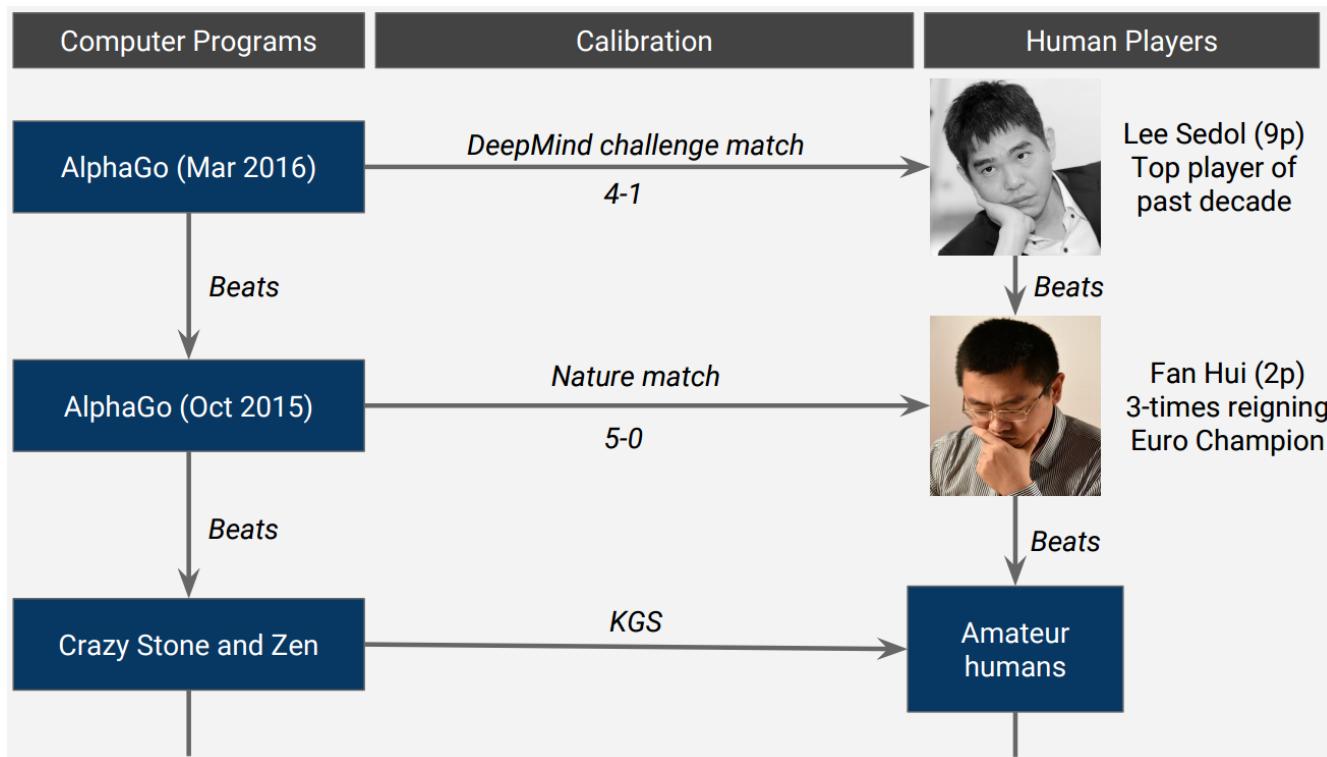
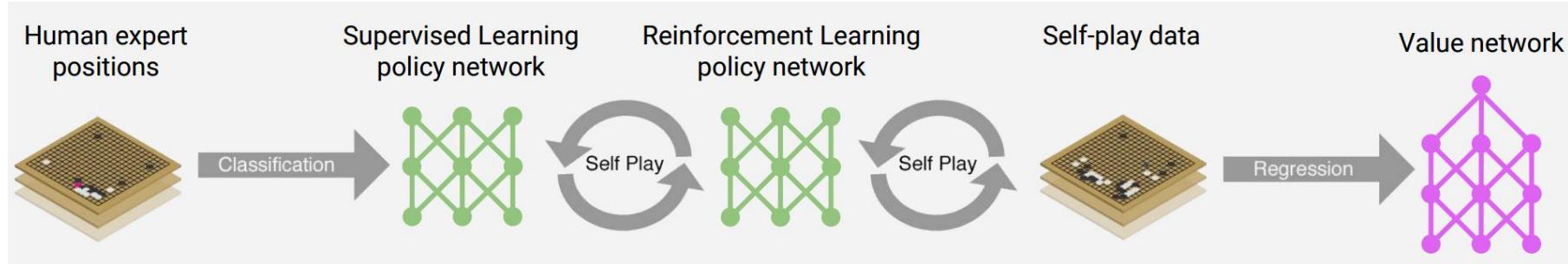
Tutorial:

<http://cars.mit.edu/deeptraffic>

Simulation:

<http://cars.mit.edu/deeptrafficjs>

Human-in-the-Loop Reinforcement Learning: Driving Ready?



Reminder: Unexpected Local Pockets of High Reward



References

All references cited in this presentation are listed in the following Google Sheets file:

<https://goo.gl/9Xhp2t>