

Lecture 6: CNNs and Deep Q Learning¹

Emma Brunskill

CS234 Reinforcement Learning.

Winter 2021

¹With many slides for DQN from David Silver and Ruslan Salakhutdinov and some vision slides from Gianni Di Caro and images from Stanford CS231n,
<http://cs231n.github.io/convolutional-networks/>

Lecture 6: Refresh Your Knowledge

- In TD learning with linear VFA (select all):
 - ① $\mathbf{w} = \mathbf{w} + \alpha(r(s_t) + \gamma \mathbf{x}(s_{t+1})^T \mathbf{w} - \mathbf{x}(s_t)^T \mathbf{w}) \mathbf{x}(s_t)$
 - ② $V(s) = \mathbf{w}(s) \mathbf{x}(s)$
 - ③ Asymptotic convergence to the true best minimum MSE linear representable $V(s)$ is guaranteed for $\alpha \in (0, 1)$, $\gamma < 1$.
 - ④ Not sure

Lecture 6: Refresh Your Knowledge **Solutions**

- In TD learning with linear VFA (select all):

- ① $\mathbf{w} = \mathbf{w} + \alpha(r(s_t) + \gamma \mathbf{x}(s_{t+1})^T \mathbf{w} - \mathbf{x}(s_t)^T \mathbf{w}) \mathbf{x}(s_t)$
- ② $V(s) = \mathbf{w}(s) \mathbf{x}(s)$
- ③ Asymptotic convergence to the true best minimum MSE linear representable $V(s)$ is guaranteed for $\alpha \in (0, 1)$, $\gamma < 1$.
- ④ Not sure

Answer: 1 is true. Convergence is not guaranteed to the best, the resulting one may still be worse than the best MSE solution by a factor of $\frac{1}{1-\gamma}$. The weights do not depend on the state.

Class Structure

- Last time: Value function approximation
- This time: RL with function approximation, deep RL
- Next time: Deep RL continued

Today

- Value function approximation
- Deep neural networks
- CNNs
- DQN

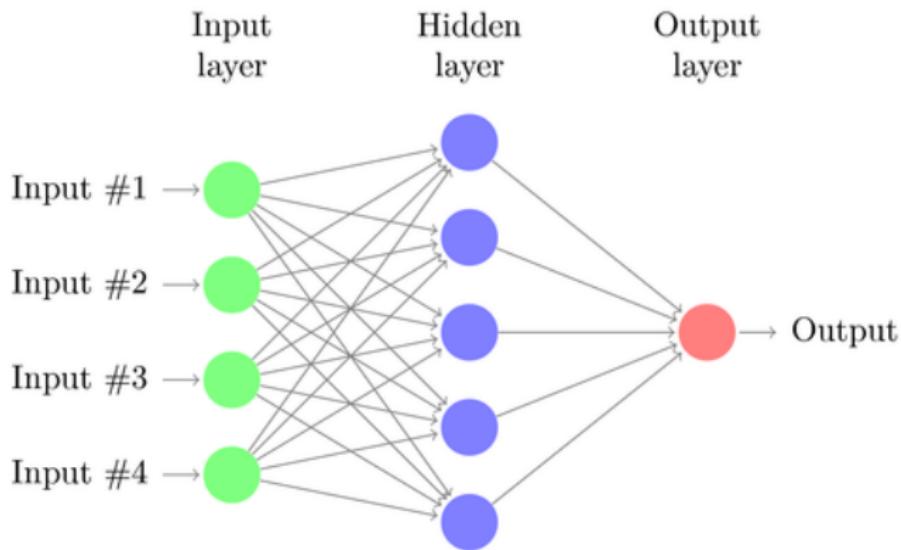
Control using Value Function Approximation

- Use value function approximation to represent state-action values
 $\hat{Q}^\pi(s, a; \mathbf{w}) \approx Q^\pi$
- Interleave
 - Approximate policy evaluation using value function approximation
 - Perform ϵ -greedy policy improvement
- Can be unstable. Generally involves intersection of the following:
 - Function approximation
 - Bootstrapping
 - **Off-policy learning**

RL with Function Approximation

- Linear value function approximators assume value function is a weighted combination of a set of features, where each feature a function of the state
- Linear VFA often work well given the right set of features
- But can require carefully hand designing that feature set
- An alternative is to use a much richer function approximation class that is able to directly go from states without requiring an explicit specification of features
- Local representations including Kernel based approaches have some appealing properties (including convergence results under certain cases) but can't typically scale well to enormous spaces and datasets

Neural Networks ¹



¹Figure by Kjell Magne Fauske

Deep Neural Networks (DNN)

- Composition of multiple functions
- Can use the chain rule to backpropagate the gradient
- Major innovation: tools to automatically compute gradients for a DNN

Deep Neural Networks (DNN) Specification and Fitting

- Generally combines both linear and non-linear transformations
 - Linear:
 - Non-linear:
- To fit the parameters, require a loss function (MSE, log likelihood etc)

The Benefit of Deep Neural Network Approximators

- Linear value function approximators assume value function is a weighted combination of a set of features, where each feature a function of the state
- Linear VFA often work well given the right set of features
- But can require carefully hand designing that feature set
- An alternative is to use a much richer function approximation class that is able to directly go from states without requiring an explicit specification of features
- Local representations including Kernel based approaches have some appealing properties (including convergence results under certain cases) but can't typically scale well to enormous spaces and datasets
- Alternative: Deep neural networks
 - Uses distributed representations instead of local representations
 - Universal function approximator
 - Can potentially need exponentially less nodes/parameters (compared to a shallow net) to represent the same function
 - Can learn the parameters using stochastic gradient descent

Today

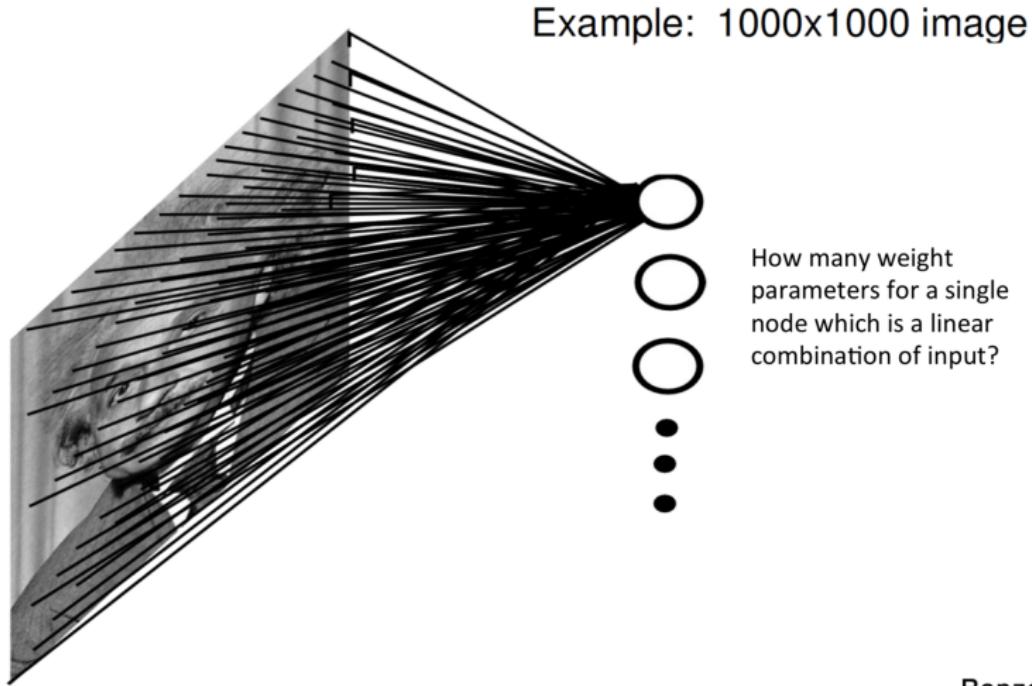
- Value function approximation
- Deep neural networks
- **CNNs**
- DQN

Why Do We Care About CNNs?

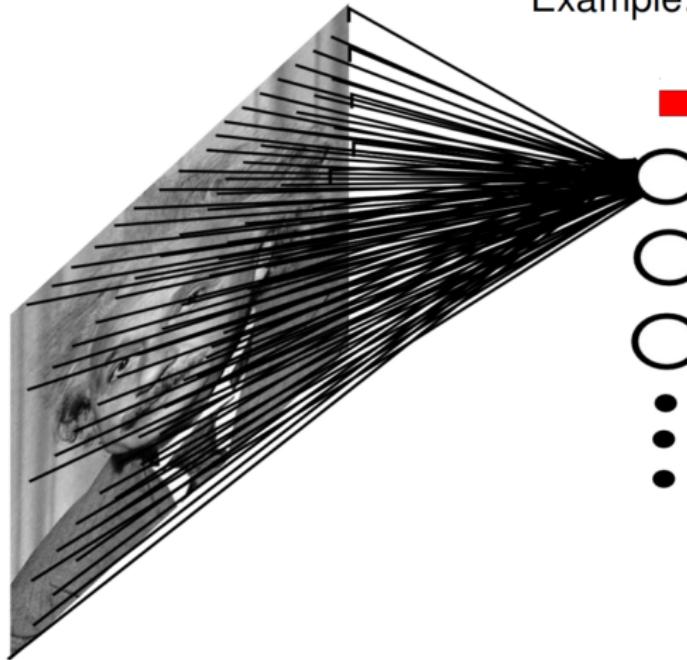
- CNNs extensively used in computer vision
- If we want to go from pixels to decisions, likely useful to leverage insights for visual input



Fully Connected Neural Net



Fully Connected Neural Net



Example: 1000x1000 image

1M hidden units

→ **10¹² parameters!!!**

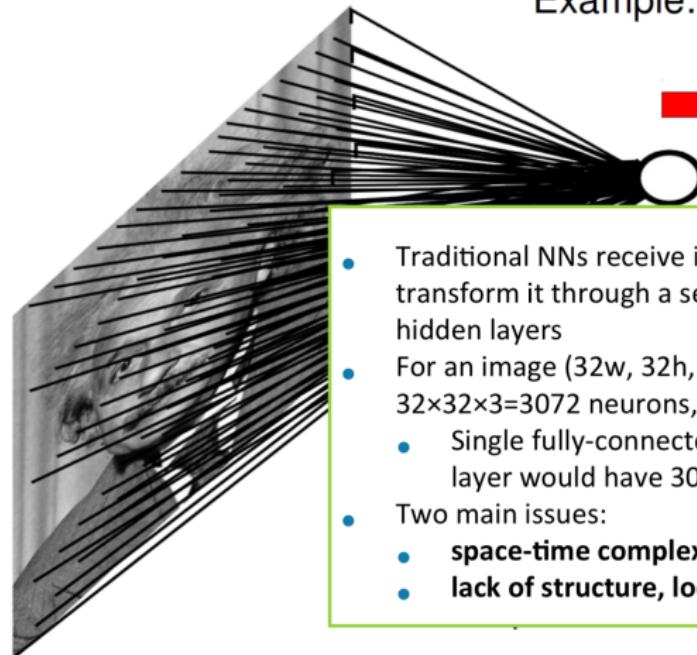


Fully Connected Neural Net

Example: 1000x1000 image

1M hidden units

→ **10¹² parameters!!!**



- Traditional NNs receive input as single vector & transform it through a series of (fully connected) hidden layers
- For an image (32w, 32h, 3c), the input layer has $32 \times 32 \times 3 = 3072$ neurons,
 - Single fully-connected neuron in the first hidden layer would have 3072 weights ...
- Two main issues:
 - **space-time complexity**
 - **lack of structure, locality of info**

Images Have Structure

- Have local structure and correlation
- Have distinctive features in space & frequency domains

Convolutional NN

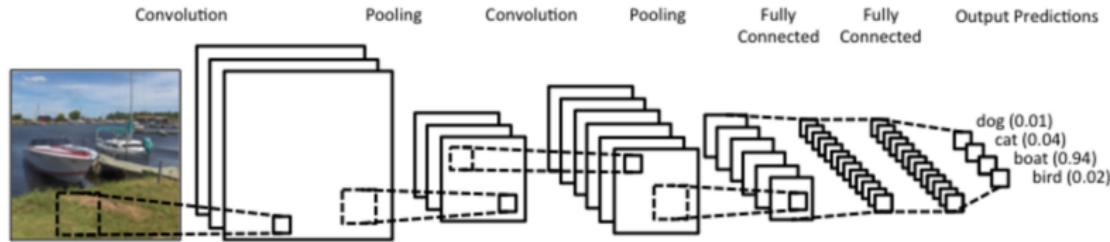
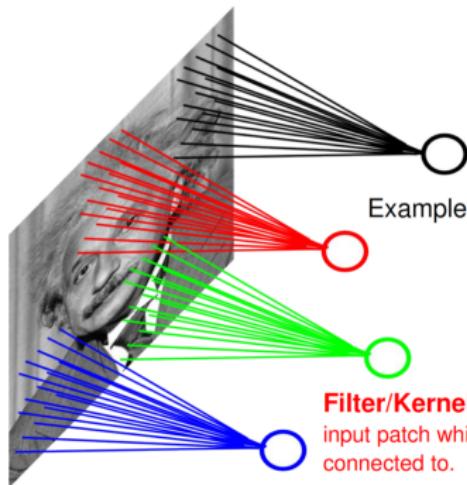


Image: <http://d3kbpzmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png>

- Consider local structure and common extraction of features
- Not fully connected
- Locality of processing
- Weight sharing for parameter reduction
- Learn the parameters of multiple convolutional filter banks
- Compress to extract salient features & favor generalization

Locality of Information: Receptive Fields



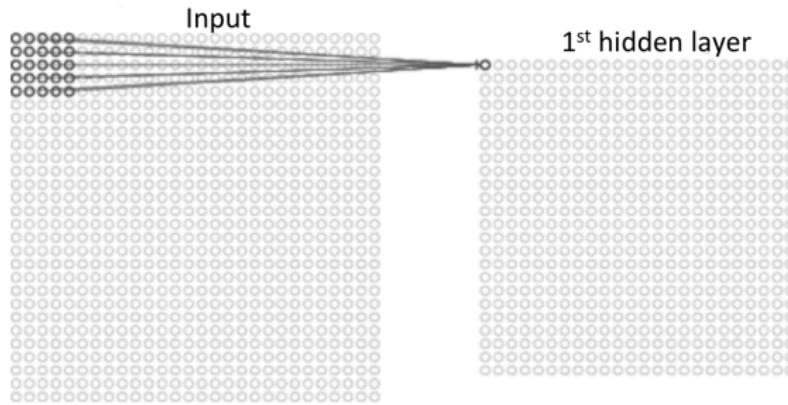
Example: 1000x1000 image
1M hidden units
Filter size: 10x10
100M parameters

Filter/Kernel/Receptive field:
input patch which the hidden unit is
connected to.



(Filter) Stride

- Slide the 5×5 mask over all the input pixels
- Stride length = 1
 - Can use other stride lengths
- Assume input is 28×28 , how many neurons in 1st hidden layer?



- Zero padding: how many 0s to add to either side of input layer

Shared Weights

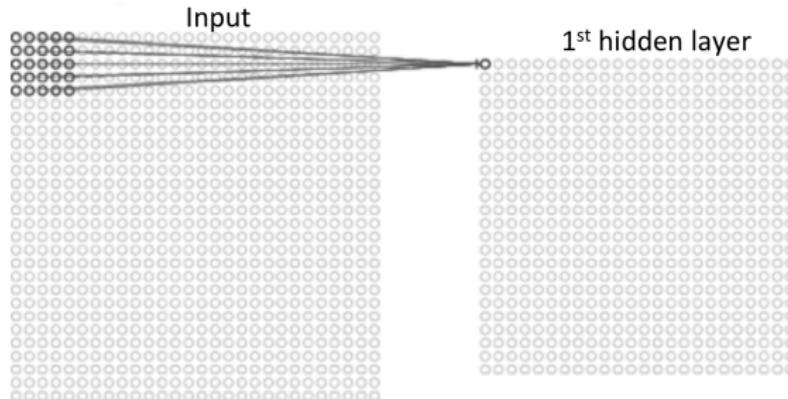
- What is the precise relationship between the neurons in the receptive field and that in the hidden layer?
- What is the *activation value* of the hidden layer neuron?

$$g(b + \sum_i w_i x_i)$$

- Sum over i is *only over the neurons in the receptive field* of the hidden layer neuron
- *The same weights w and bias b* are used for each of the hidden neurons
 - In this example, 24×24 hidden neurons

Ex. Shared Weights, Restricted Field

- Consider 28x28 input image
- 24x24 hidden layer

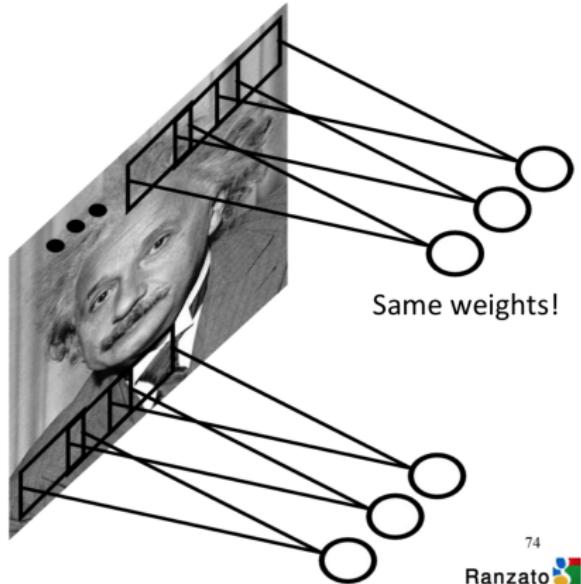


- Receptive field is 5x5

Feature Map

- All the neurons in the first hidden layer *detect exactly the same feature, just at different locations* in the input image.
- **Feature:** the kind of input pattern (e.g., a local edge) that makes the neuron produce a certain response level
- Why does this makes sense?
 - Suppose the weights and bias are (learned) such that the hidden neuron can pick out, a vertical edge in a particular local receptive field.
 - That ability is also likely to be useful at other places in the image.
 - Useful to apply the same feature detector everywhere in the image.
Yields translation (spatial) invariance (try to detect feature at any part of the image)
 - Inspired by visual system

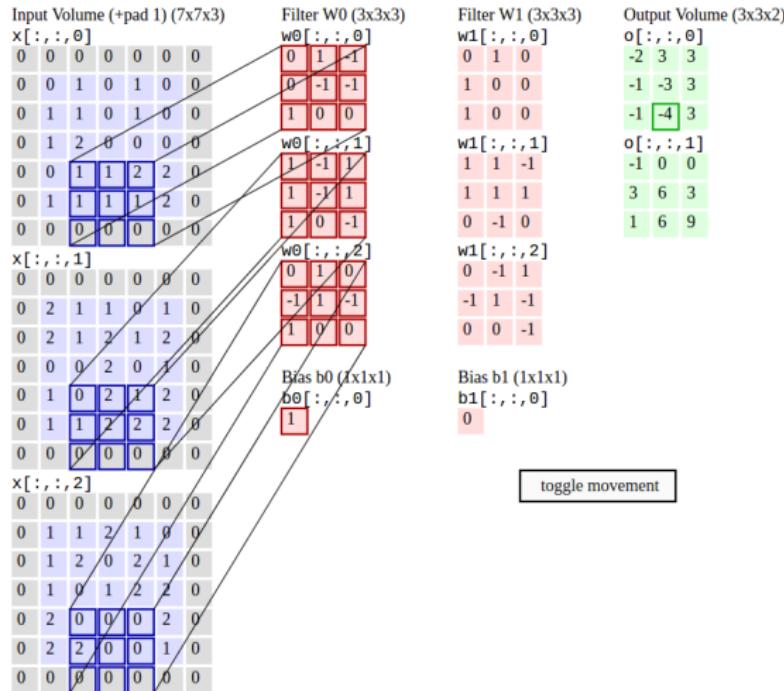
Feature Map



74
Ranzato

- The map from the input layer to the hidden layer is therefore a feature map: all nodes detect the same feature in different parts
- The map is defined by the shared weights and bias
- The shared map is the result of the application of a convolutional filter (defined by weights and bias), also known as convolution with learned kernels

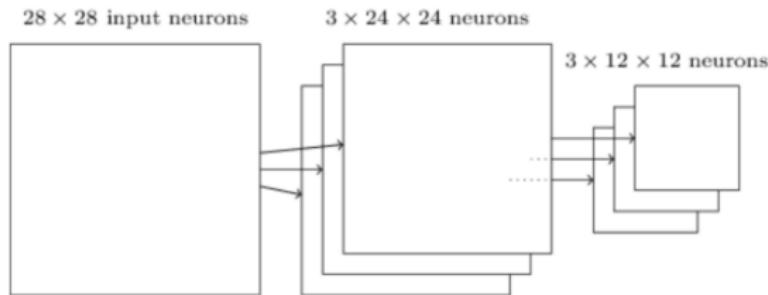
Convolutional Layer: Multiple Filters Ex.²



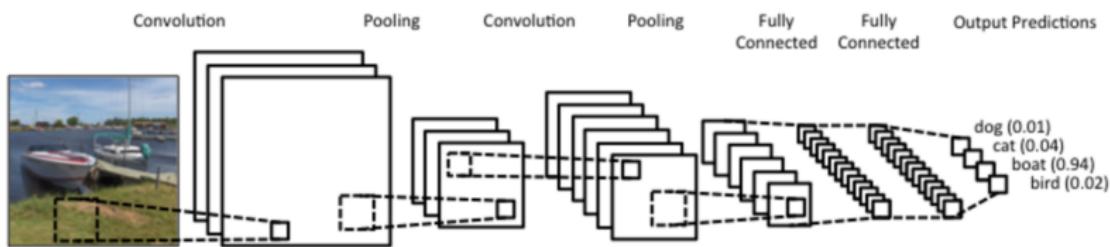
²<http://cs231n.github.io/convolutional-networks/>

Pooling Layers

- Pooling layers are usually used immediately after convolutional layers.
- Pooling layers simplify / subsample / compress the information in the output from convolutional layer
- A pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map



Final Layer Typically Fully Connected



Today

- Value function approximation
- Deep neural networks
- CNNs
- **DQN**

Generalization

- Using function approximation to help scale up to making decisions in really large domains



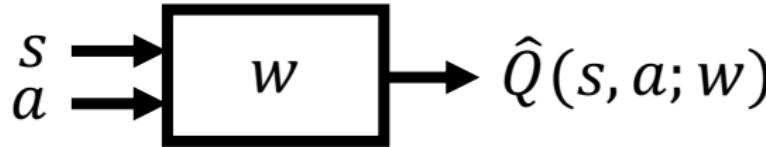
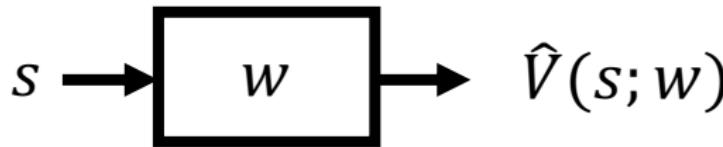
Deep Reinforcement Learning

- Use deep neural networks to represent
 - Value, Q function
 - Policy
 - Model
- Optimize loss function by stochastic gradient descent (SGD)

Deep Q-Networks (DQNs)

- Represent state-action value function by Q-network with weights w

$$\hat{Q}(s, a; w) \approx Q(s, a)$$



Recall: Incremental Model-Free Control Approaches

- Similar to policy evaluation, true state-action value function for a state is unknown and so substitute a target value
- In Monte Carlo methods, use a return G_t as a substitute target

$$\Delta \mathbf{w} = \alpha(G_t - \hat{Q}(s_t, a_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

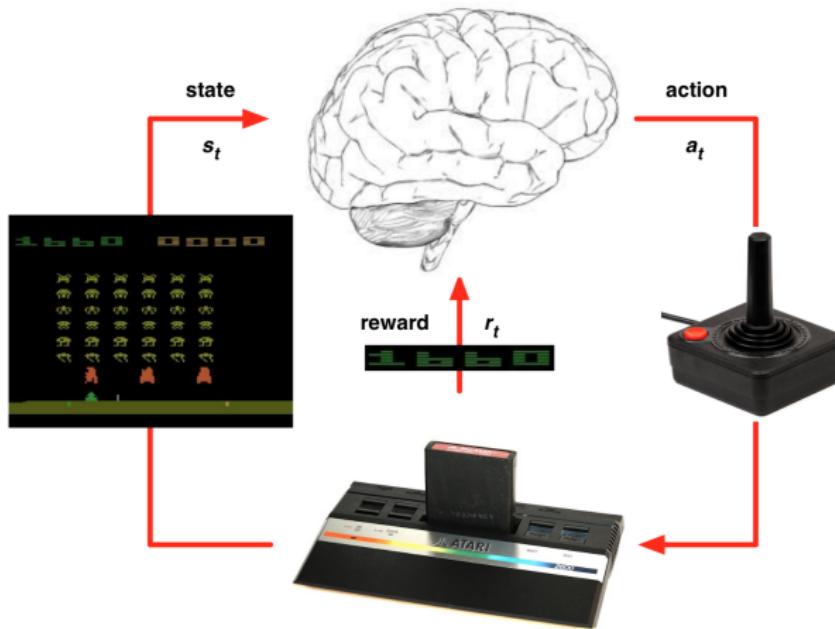
- For SARSA instead use a TD target $r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w})$ which leverages the current function approximation value

$$\Delta \mathbf{w} = \alpha(r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

- For Q-learning instead use a TD target $r + \gamma \max_a \hat{Q}(s_{t+1}, a; \mathbf{w})$ which leverages the max of the current function approximation value

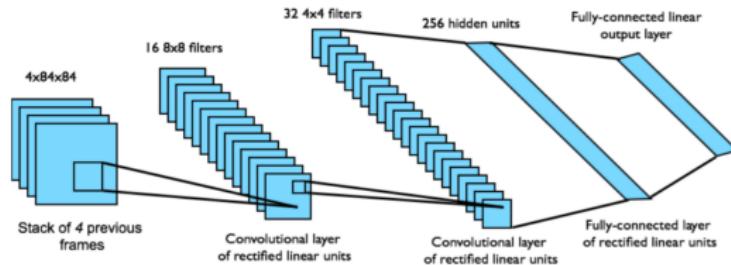
$$\Delta \mathbf{w} = \alpha(r + \gamma \max_a \hat{Q}(s_{t+1}, a; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

Using these ideas to do Deep RL in Atari



DQNs in Atari

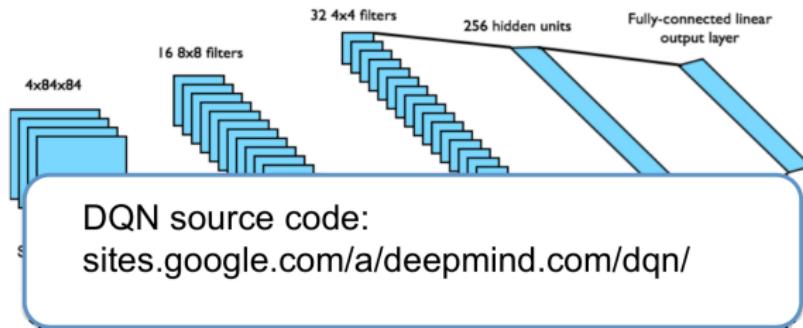
- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



- Network architecture and hyperparameters fixed across all games

DQNs in Atari

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



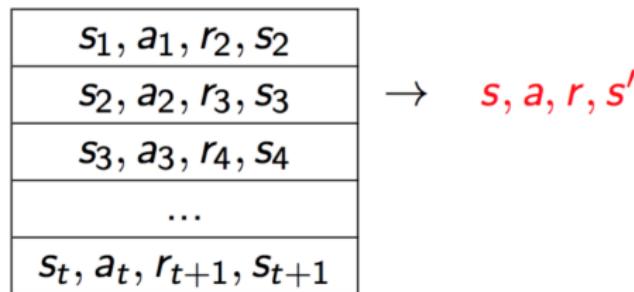
- Network architecture and hyperparameters fixed across all games

Q-Learning with Value Function Approximation

- Q-learning converges to the optimal $Q^*(s, a)$ using table lookup representation
- In value function approximation Q-learning we can minimize MSE loss by stochastic gradient descent using a target Q estimate instead of true Q (as we saw with linear VFA)
- But Q-learning with VFA can diverge
- Two of the issues causing problems:
 - Correlations between samples
 - Non-stationary targets
- Deep Q-learning (DQN) addresses these challenges by
 - Experience replay
 - Fixed Q-targets

DQNs: Experience Replay

- To help remove correlations, store dataset (called a **replay buffer**) \mathcal{D} from prior experience



- To perform experience replay, repeat the following:
 - $(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
 - Compute the target value for the sampled s : $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w})$
 - Use stochastic gradient descent to update the network weights

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}) - \hat{Q}(s, a; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a; \mathbf{w})$$

DQNs: Experience Replay

- To help remove correlations, store dataset \mathcal{D} from prior experience

| |
|------------------------------|
| s_1, a_1, r_2, s_2 |
| s_2, a_2, r_3, s_3 |
| s_3, a_3, r_4, s_4 |
| \dots |
| $s_t, a_t, r_{t+1}, s_{t+1}$ |

→ s, a, r, s'

- To perform experience replay, repeat the following:
 - $(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
 - Compute the target value for the sampled s : $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w})$
 - Use stochastic gradient descent to update the network weights

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}) - \hat{Q}(s, a; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a; \mathbf{w})$$

- Can treat the target as a scalar, but the weights will get updated on the next round, changing the target value**

DQNs: Fixed Q-Targets

- To help improve stability, fix the **target weights** used in the target calculation for multiple updates
- Target network uses a different set of weights than the weights being updated
- Let parameters \mathbf{w}^- be the set of weights used in the target, and \mathbf{w} be the weights that are being updated
- Slight change to computation of target value:
 - $(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
 - Compute the target value for the sampled s : $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^-)$
 - Use stochastic gradient descent to update the network weights

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^-) - \hat{Q}(s, a; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a; \mathbf{w})$$

DQN Pseudocode

```
1: Input  $C, \alpha$ ,  $D = \{\}$ , Initialize  $\mathbf{w}$ ,  $\mathbf{w}^- = \mathbf{w}$ ,  $t = 0$ 
2: Get initial state  $s_0$ 
3: loop
4:   Sample action  $a_t$  given  $\epsilon$ -greedy policy for current  $\hat{Q}(s_t, a; \mathbf{w})$ 
5:   Observe reward  $r_t$  and next state  $s_{t+1}$ 
6:   Store transition  $(s_t, a_t, r_t, s_{t+1})$  in replay buffer  $D$ 
7:   Sample random minibatch of tuples  $(s_i, a_i, r_i, s_{i+1})$  from  $D$ 
8:   for  $j$  in minibatch do
9:     if episode terminated at step  $i + 1$  then
10:       $y_i = r_i$ 
11:    else
12:       $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}^-)$ 
13:    end if
14:    Do gradient descent step on  $(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))^2$  for parameters  $\mathbf{w}$ :  $\Delta \mathbf{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_i, a_i; \mathbf{w})$ 
15:   end for
16:    $t = t + 1$ 
17:   if mod( $t, C$ ) == 0 then
18:      $\mathbf{w}^- \leftarrow \mathbf{w}$ 
19:   end if
20: end loop
```

DQN Pseudocode Hyperparameters

```
1: Input  $C, \alpha$ ,  $D = \{\}$ , Initialize  $\mathbf{w}$ ,  $\mathbf{w}^- = \mathbf{w}$ ,  $t = 0$ 
2: Get initial state  $s_0$ 
3: loop
4:   Sample action  $a_t$  given  $\epsilon$ -greedy policy for current  $\hat{Q}(s_t, a; \mathbf{w})$ 
5:   Observe reward  $r_t$  and next state  $s_{t+1}$ 
6:   Store transition  $(s_t, a_t, r_t, s_{t+1})$  in replay buffer  $D$ 
7:   Sample random minibatch of tuples  $(s_i, a_i, r_i, s_{i+1})$  from  $D$ 
8:   for  $j$  in minibatch do
9:     if episode terminated at step  $i + 1$  then
10:        $y_i = r_i$ 
11:     else
12:        $y_i = r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'; \mathbf{w}^-)$ 
13:     end if
14:     Do gradient descent step on  $(y_i - \hat{Q}(s_i, a_i; \mathbf{w}))^2$  for parameters  $\mathbf{w}$ :  $\Delta \mathbf{w} = \alpha(y_i - \hat{Q}(s_i, a_i; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_i, a_i; \mathbf{w})$ 
15:   end for
16:    $t = t + 1$ 
17:   if mod( $t, C$ ) == 0 then
18:      $\mathbf{w}^- \leftarrow \mathbf{w}$ 
19:   end if
20: end loop
```

Note there are several hyperparameters and algorithm choices. One needs to choose the neural network architecture, the learning rate, and how often to update the target network. Often a fixed size replay buffer is used for experience replay, which introduces a parameter to control the size, and the need to decide how to populate it.

Check Your Understanding: Fixed Targets

- In DQN we compute the target value for the sampled (s, a, r, s') using a separate set of target weights: $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^-)$
- Select all that are true
- If the target network is trained on other data, this might help with the maximization bias
- This doubles the computation time compared to a method that does not have a separate set of weights
- This doubles the memory requirements compared to a method that does not have a separate set of weights
- Not sure

Check Your Understanding: Fixed Targets **Solutions**

- In DQN we compute the target value for the sampled (s, a, r, s') using a separate set of target weights: $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}^-)$
- Select all that are true
- If the target network is trained on other data, this might help with the maximization bias
- This doubles the computation time compared to a method that does not have a separate set of weights
- This doubles the memory requirements compared to a method that does not have a separate set of weights
- Not sure

Answer: It doubles the memory requirements. In the maximization bias, we use a separate function to choose the action and to evaluate the value of it.

DQNs Summary

- DQN uses experience replay and fixed Q-targets
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory \mathcal{D}
- Sample random mini-batch of transitions (s, a, r, s') from \mathcal{D}
- Compute Q-learning targets w.r.t. old, fixed parameters \mathbf{w}^-
- Optimizes MSE between Q-network and Q-learning targets
- Uses stochastic gradient descent

DQN

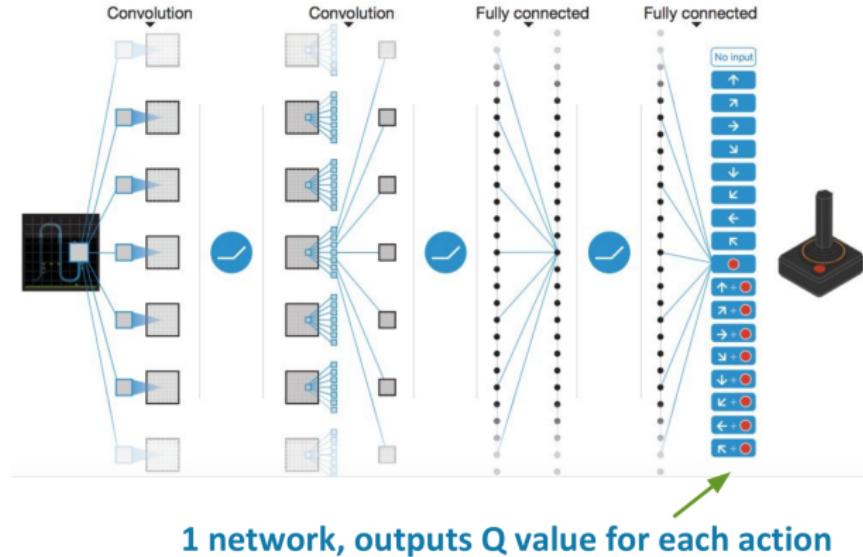


Figure: Human-level control through deep reinforcement learning, Mnih et al, 2015

Demo

DQN Results in Atari

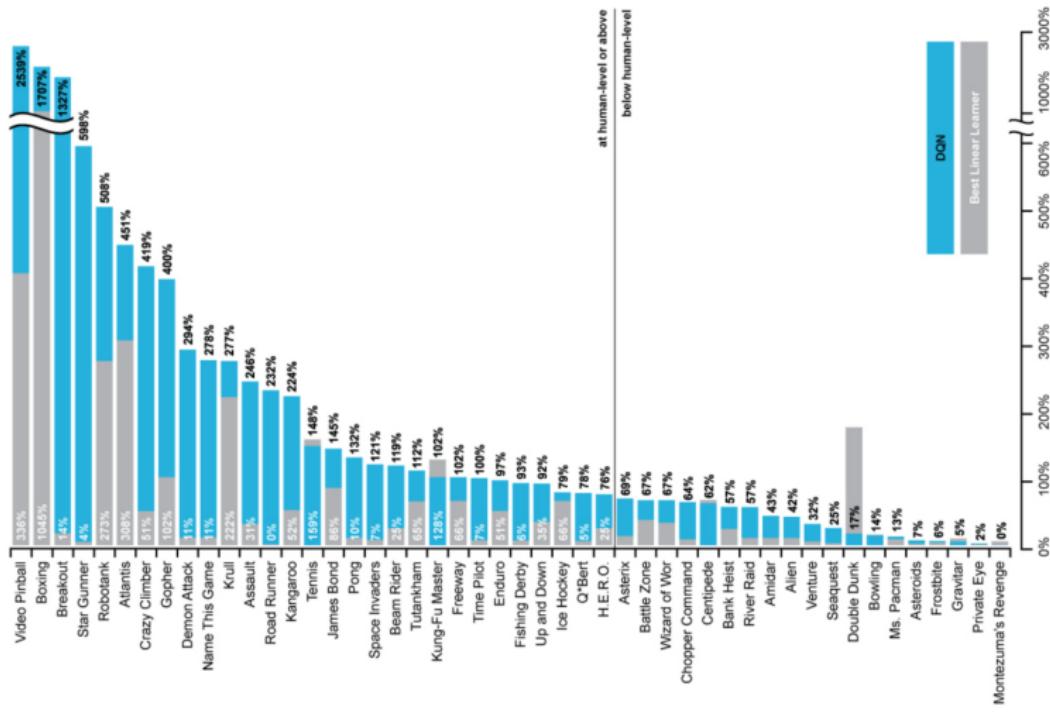


Figure: Human-level control through deep reinforcement learning, Mnih et al, 2015

Which Aspects of DQN were Important for Success?

| Game | Linear | Deep Network |
|----------------|--------|--------------|
| Breakout | 3 | 3 |
| Enduro | 62 | 29 |
| River Raid | 2345 | 1453 |
| Seaquest | 656 | 275 |
| Space Invaders | 301 | 302 |

Note: just using a deep NN actually hurt performance sometimes!

Which Aspects of DQN were Important for Success?

| Game | Linear | Deep Network | DQN w/ fixed Q |
|----------------|--------|--------------|-------------------|
| Breakout | 3 | 3 | 10 |
| Enduro | 62 | 29 | 141 |
| River Raid | 2345 | 1453 | 2868 |
| Seaquest | 656 | 275 | 1003 |
| Space Invaders | 301 | 302 | 373 |

Which Aspects of DQN were Important for Success?

| Game | Linear | Deep Network | DQN w/ fixed Q | DQN w/ replay | DQN w/replay and fixed Q |
|----------------|--------|--------------|----------------|---------------|--------------------------|
| Breakout | 3 | 3 | 10 | 241 | 317 |
| Enduro | 62 | 29 | 141 | 831 | 1006 |
| River Raid | 2345 | 1453 | 2868 | 4102 | 7447 |
| Seaquest | 656 | 275 | 1003 | 823 | 2894 |
| Space Invaders | 301 | 302 | 373 | 826 | 1089 |

- Replay is **hugely** important
- Why? Beyond helping with correlation between samples, what does replaying do?

- Success in Atari has led to huge excitement in using deep neural networks to do value function approximation in RL
- Some immediate improvements (many others!)
 - **Double DQN** (Deep Reinforcement Learning with Double Q-Learning, Van Hasselt et al, AAAI 2016)
 - Prioritized Replay (Prioritized Experience Replay, Schaul et al, ICLR 2016)
 - Dueling DQN (best paper ICML 2016) (Dueling Network Architectures for Deep Reinforcement Learning, Wang et al, ICML 2016)

Class Structure

- Last time and start of this time: Control (making decisions) without a model of how the world works
- Rest of today: Deep reinforcement learning
- Next time: Deep RL continued