

Lecture 6: CNNs and Deep Q Learning ¹

Emma Brunskill

CS234 Reinforcement Learning.

Winter 2020

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/>

Refresh Your Knowledge 5

- In TD learning with linear VFA (select all):
 - 1 $w = w + \alpha(r(s_t) + \gamma x(s_{t+1})^T w - x(s_t)^T w)x(s_t)$
 - 2 $V(s) = w(s)x(s)$
 - 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

Class Structure

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

Table of Contents

1 Control using Value Function Approximation

2 Convolutional Neural Nets (CNNs)

3 Deep Q Learning

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**

Linear State Action Value Function Approximation

- Use features to represent both the state and action

$$\mathbf{x}(s, a) = \begin{pmatrix} x_1(s, a) \\ x_2(s, a) \\ \dots \\ x_n(s, a) \end{pmatrix}$$

- Represent state-action value function with a weighted linear combination of features

$$\hat{Q}(s, a; \mathbf{w}) = \mathbf{x}(s, a)^T \mathbf{w} = \sum_{j=1}^d x_j(s, a) w_j$$

- Gradient descent update:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \nabla_{\mathbf{w}} \mathbb{E}_{\pi} [(Q^{\pi}(s, a) - \hat{Q}^{\pi}(s, a; \mathbf{w}))^2]$$

Incremental Model-Free Control Approaches w/Linear VFA

- 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 $r + \gamma \hat{Q}(s', a'; \mathbf{w})$ which leverages the current function approximation value

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(r + \gamma \hat{Q}(s', a'; \mathbf{w}) - \hat{Q}(s, a; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a; \mathbf{w}) \\ &= \alpha(r + \gamma \mathbf{x}(s', a')^T \mathbf{w} - \mathbf{x}(s, a)^T \mathbf{w}) \mathbf{x}(s, a) \end{aligned}$$

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 $r + \gamma \hat{Q}(s', a'; \mathbf{w})$ which leverages the current function approximation value

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(r + \gamma \hat{Q}(s', a'; \mathbf{w}) - \hat{Q}(s, a; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a; \mathbf{w}) \\ &= \alpha(r + \gamma \mathbf{x}(s', a')^T \mathbf{w} - \mathbf{x}(s, a)^T \mathbf{w}) \mathbf{x}(s, a) \end{aligned}$$

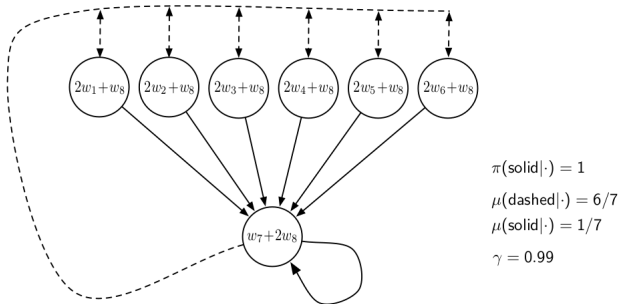
- For Q-learning instead use a TD target $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w})$ which leverages the max of the current function approximation value

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w}) - \hat{Q}(s, a; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s, a; \mathbf{w}) \\ &= \alpha(r + \gamma \max_{a'} \mathbf{x}(s', a')^T \mathbf{w} - \mathbf{x}(s, a)^T \mathbf{w}) \mathbf{x}(s, a) \end{aligned}$$

Convergence of TD Methods with VFA

- Informally, updates involve doing an (approximate) Bellman backup followed by best trying to fit underlying value function to a particular feature representation
- Bellman operators are contractions, but value function approximation fitting can be an expansion

Challenges of Off Policy Control: Baird Example ¹



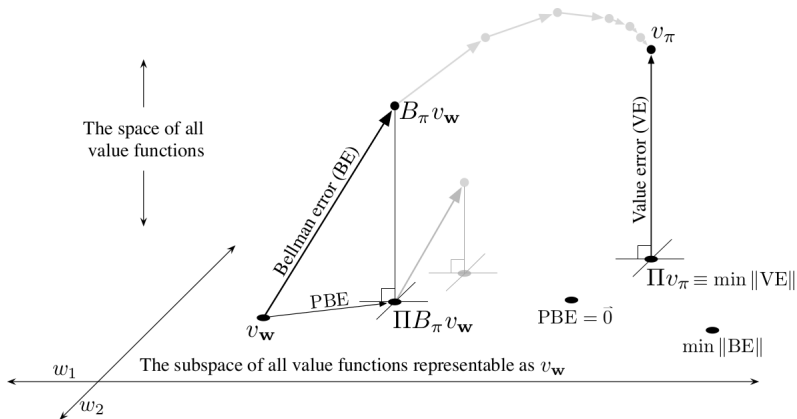
- Behavior policy and target policy are not identical
- Value can diverge

Convergence of Control Methods with VFA

Algorithm	Tabular	Linear VFA	Nonlinear VFA
Monte-Carlo Control			
Sarsa			
Q-learning			

- This is an active area of research
- An important issue is not just whether the algorithm converges, but **what** solution it converges to
- Critical choices: **objective function and feature representation**
- Chp 11 SB has a good discussion of some efforts in this direction

Linear Value Function Approximation¹



¹Figure from Sutton and Barto 2018

What You Should Understand

- Be able to implement TD(0) and MC on policy evaluation with linear value function approximation
- Be able to define what TD(0) and MC on policy evaluation with linear VFA are converging to and when this solution has 0 error and non-zero error.
- Be able to implement Q-learning and SARSA and MC control algorithms
- List the 3 issues that can cause instability and describe the problems qualitatively: function approximation, bootstrapping and off policy learning

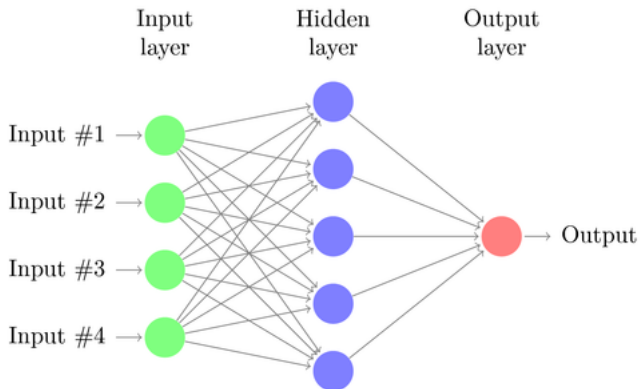
Class Structure

- 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

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 ²



²Figure 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

Table of Contents

- 1 Control using Value Function Approximation
- 2 Convolutional Neural Nets (CNNs)
- 3 Deep Q Learning

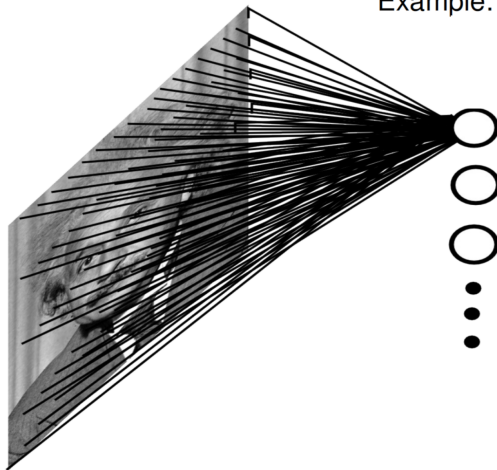
Why Do We Care About CNNs?

- 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

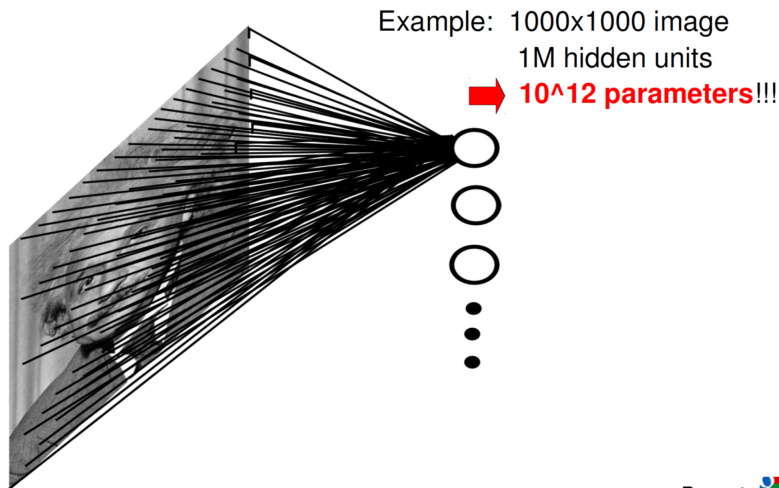
Example: 1000x1000 image



How many weight parameters for a single node which is a linear combination of input?

Ranzato 

Fully Connected Neural Net

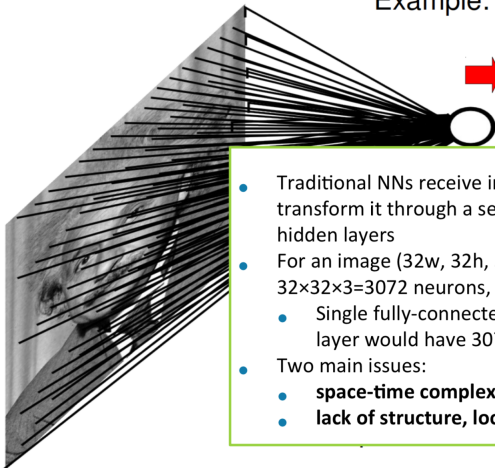


Ranzato 

Fully Connected Neural Net

Example: 1000x1000 image
1M hidden units

➡ **10^{12} 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**

Ranzato 

Images Have Structure

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

Convolutional NN

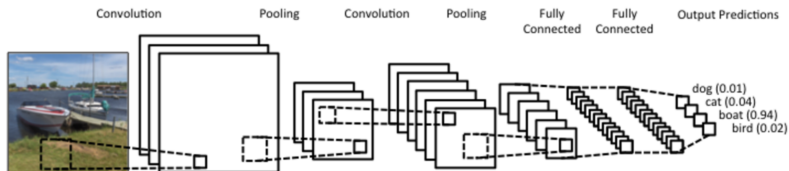
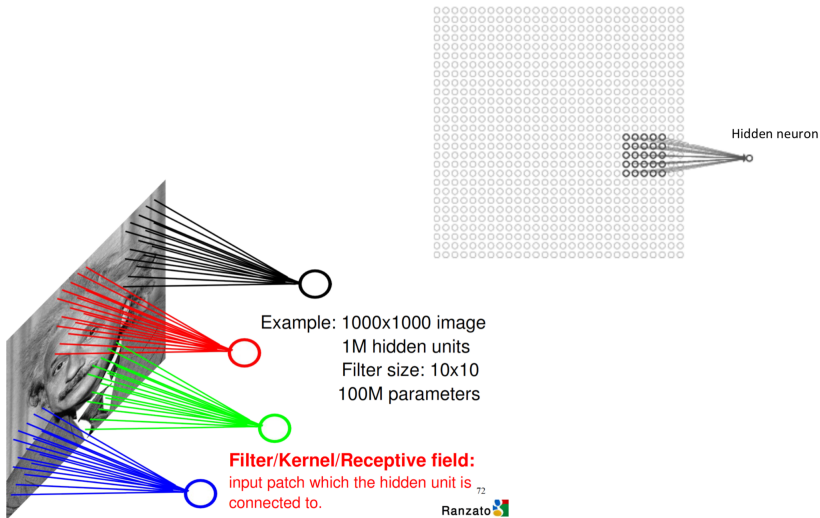


Image: <http://d3kbpzmcynnm.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png>

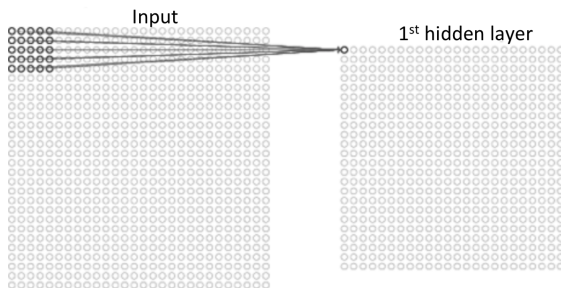
- Consider local structure and common extraction of features
- Not fully connected
- Locality of processing
- Weight sharing for parameter reduction
- Learn the parameters of multiple convolutional filter banks
- Compress to extract salient features & favor generalization

Locality of Information: Receptive Fields



(Filter) Stride

- Slide the 5x5 mask over all the input pixels
- Stride length = 1
 - Can use other stride lengths
- Assume input is 28x28, how many neurons in 1st hidden layer?



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

Shared Weights

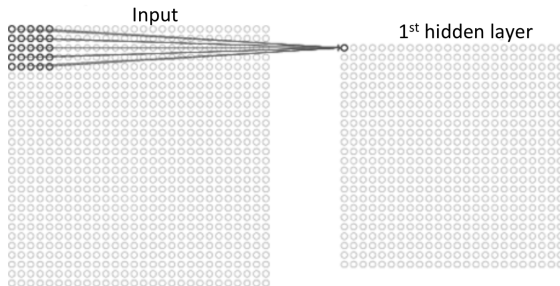
- What is the precise relationship between the neurons in the receptive field and that in the hidden layer?
- What is the *activation value* of the hidden layer neuron?

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

- 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

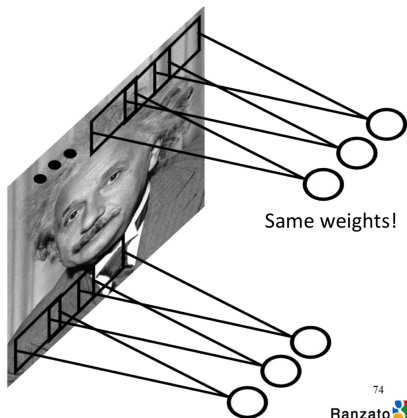


- Receptive field is 5x5

Feature Map

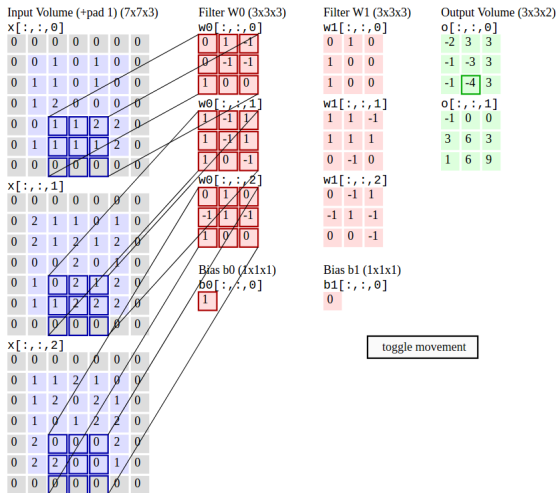
- All the neurons in the first hidden layer *detect exactly the same feature, just at different locations* in the input image.
- **Feature:** the kind of input pattern (e.g., a local edge) that makes the neuron produce a certain response level
- Why does this makes sense?
 - Suppose the weights and bias 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



- 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

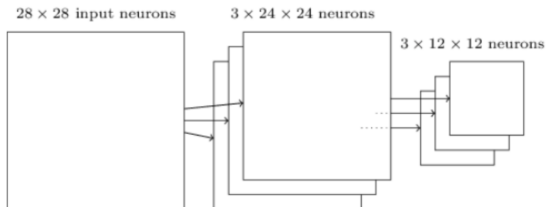
Convolutional Layer: Multiple Filters Ex.³



³<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

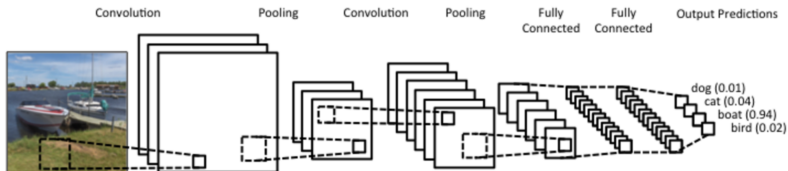


Image: <http://d3kbpzbcynnm.x.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png>

Table of Contents

- 1 Control using Value Function Approximation
- 2 Convolutional Neural Nets (CNNs)
- 3 Deep Q Learning

Generalization

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



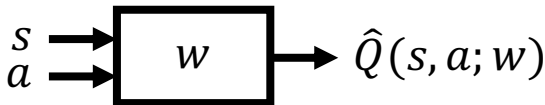
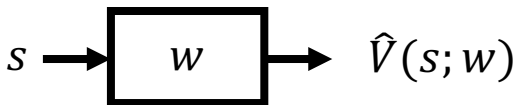
Deep Reinforcement Learning

- 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 $\mathbf{w}$

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



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

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

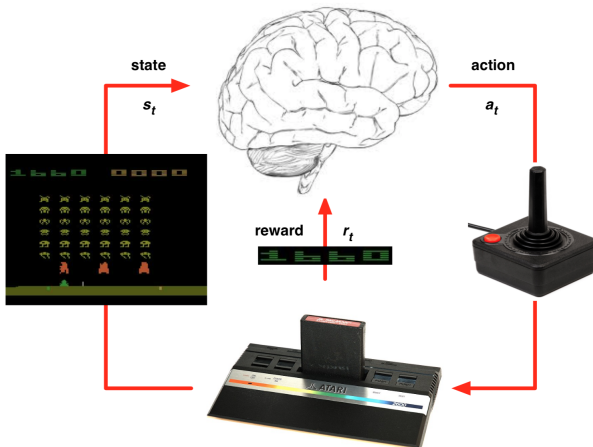
- For SARSA instead use a TD target $r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w})$ which leverages the current function approximation value

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

- 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

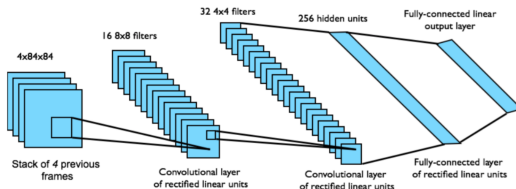
$$\Delta \mathbf{w} = \alpha(r + \gamma \max_a \hat{Q}(s_{t+1}, a; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{Q}(s_t, a_t; \mathbf{w})$$

Using these ideas to do Deep RL in Atari



DQNs in Atari

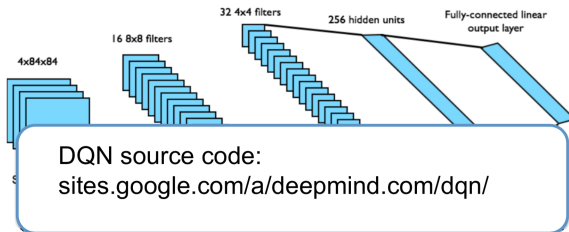
- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



- 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



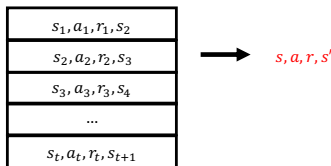
- Network architecture and hyperparameters fixed across all games

Q-Learning with Value Function Approximation

- Minimize MSE loss by stochastic gradient descent
- Converges to the optimal $Q^*(s, a)$ using table lookup representation
- But Q-learning with VFA can diverge
- Two of the issues causing problems:
 - Correlations between samples
 - Non-stationary targets
- Deep Q-learning (DQN) addresses both of 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

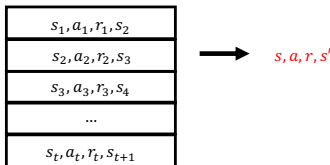


- To perform experience replay, repeat the following:
 - $(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
 - Compute the target value for the sampled s : $r + \gamma \max_{a'} \hat{Q}(s', a'; \mathbf{w})$
 - Use stochastic gradient descent to update the network weights

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

DQNs: Experience Replay

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



- 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

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

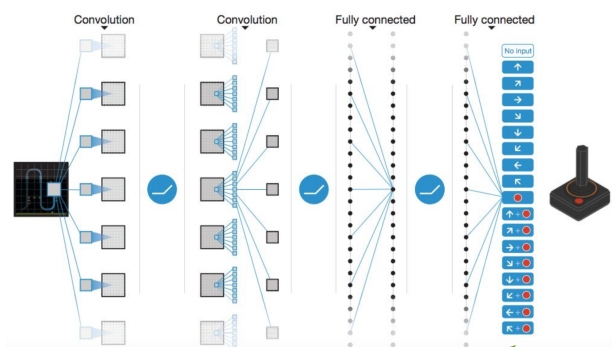
Check Your Understanding: Fixed Targets

- In DQN we compute the target value for the sampled 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

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



1 network, outputs Q value for each action

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

DQN Results in Atari

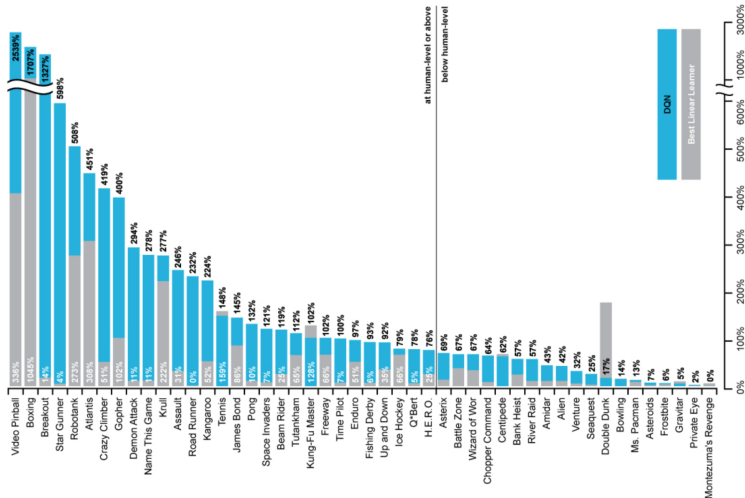


Figure: Human-level control through deep reinforcement learning, Mnih et al, 2015

Which Aspects of DQN were Important for Success?

Game	Linear	Deep Network	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