Lecture 6: CNNs and Deep Q Learning [1]

Emma Brunskill

CS234 Reinforcement Learning.

Winter 2019

[1] With 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/

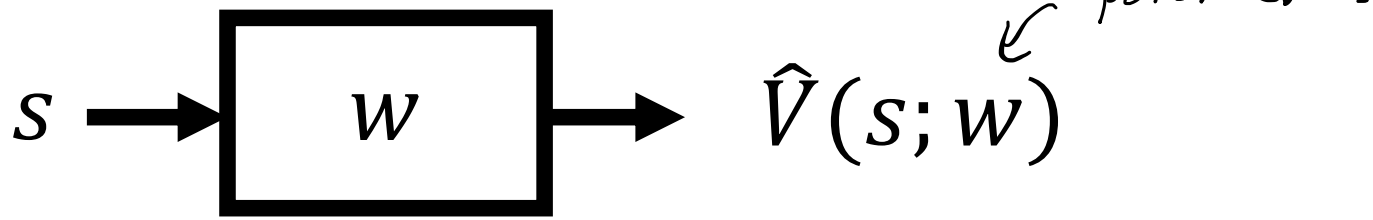
Table of Contents

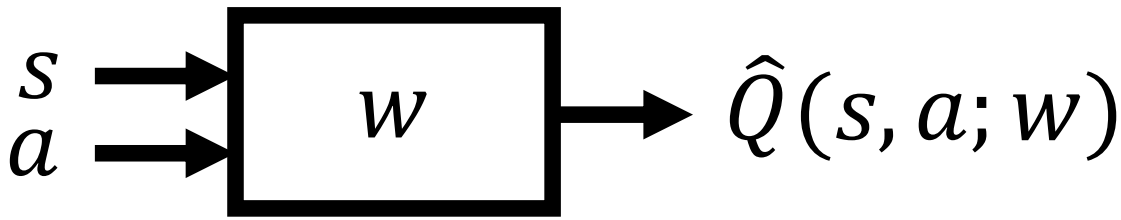
Class Structure

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

Generalization

- Want to be able to use reinforcement learning to tackle self-driving cars, Atari, consumer marketing, healthcare, education, ...
- Most of these domains have enormous state and/or action spaces
- Requires representations (of models / state-action values / values / policies) that can generalize across states and/or actions
- Represent a (state-action/state) value function with a parameterized function instead of a table





Recall: Stochastic Gradient Descent

- Goal: Find the parameter vector $\boldsymbol{w}$ that minimizes the loss between a true value function $V^{\pi}(s)$ and its approximation $\hat{V}^{\pi}(s; \boldsymbol{w})$ as represented with a particular function class parameterized by $\boldsymbol{w}$.
- Generally use mean squared error and define the loss as

$$J(\boldsymbol{w}) = \mathbb{E}_{\pi}[(V^{\pi}(s) - \hat{V}^{\pi}(s; \boldsymbol{w}))^2]$$

- Can use gradient descent to find a local minimum

$$\Delta \boldsymbol{w} = -\frac{1}{2}\alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w})$$

(s, a, r, s')

- Stochastic gradient descent (SGD) samples the gradient:

$$-\frac{1}{2}\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = \mathbb{E}_{\pi}[(V^{\pi}(s) - \hat{V}^{\pi}(s; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{V}^{\pi}(s; \boldsymbol{w})]$$

$$\Delta \boldsymbol{w} = \alpha(V^{\pi}(s) - \hat{V}^{\pi}(s; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{V}^{\pi}(s; \boldsymbol{w})$$

- Expected SGD is the same as the full gradient update

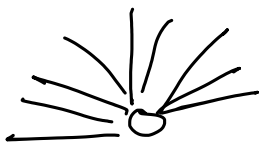
Last Time: Linear Value Function Approximation for Prediction With An Oracle

- Represent a value function (or state-action value function) for a particular policy with a weighted linear combination of features

$$\hat{V}(s; \boldsymbol{w}) = \sum_{j=1}^{n} x_j(s) w_j = \boldsymbol{x}(s)^T \boldsymbol{w}$$

features

state



- Objective function is

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[(V^\pi(s) - \hat{V}(s; \boldsymbol{w}))^2]$$

true value

- Recall weight update is

$$\Delta \boldsymbol{w} = -\frac{1}{2} \alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w})$$

Last Time: Linear Value Function Approximation for Prediction With An Oracle

- Represent a value function (or state-action value function) for a particular policy with a weighted linear combination of features

$$\hat{V}(s; \boldsymbol{w}) = \sum_{j=1}^{n} x_j(s) w_j = \boldsymbol{x}(s)^T \boldsymbol{w}$$

- Objective function is $J(\boldsymbol{w}) = \mathbb{E}_\pi[(V^\pi(s) - \hat{V}^\pi(s; \boldsymbol{w}))^2]$
- Recall weight update is $\Delta \boldsymbol{w} = -\frac{1}{2}\alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w})$
- For MC policy evaluation

$$\Delta \boldsymbol{w} = \alpha(G_t - \boldsymbol{x}(s_t)^T \boldsymbol{w})\boldsymbol{x}(s_t)$$

return from a full episode; features; pred. error

- For TD policy evaluation

$$\Delta \boldsymbol{w} = \alpha(r_t + \gamma \boldsymbol{x}(s_{t+1})^T \boldsymbol{w} - \boldsymbol{x}(s_t)^T \boldsymbol{w})\boldsymbol{x}(s_t)$$

bootstrapping

RL with Function Approximator

- 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

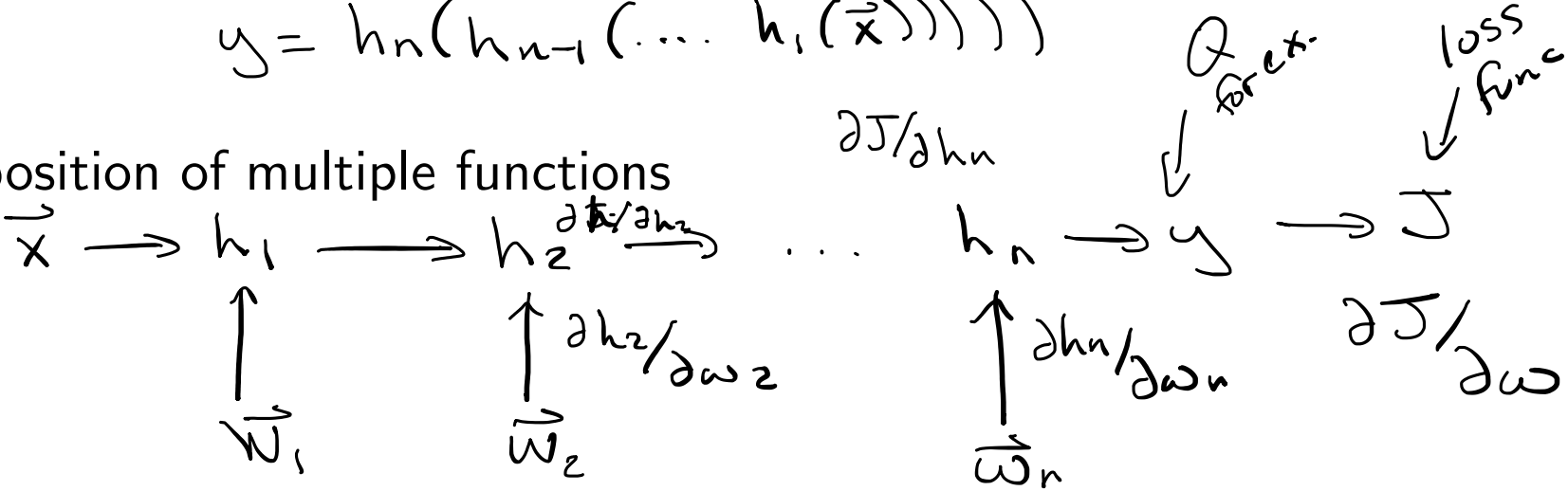
k-nearest neighbor type ?

You can think of a deep neural network is just a really complicated way to get out features, plus the last layer being a linear combination of those features.

Deep Neural Networks (DNN)

$$y = h_n(h_{n-1}(\dots h_1(\vec{x})))$$

- Composition of multiple functions



- Can use the chain rule to backpropagate the gradient

h need to be differentiable

- 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: $h_n = W h_{n-1}$
  - Non-linear: $h_n = f(h_{n-1})$ activation function (sigmoid, relu)
- To fit the parameters, require a loss function (MSE, log likelihood etc)

The Benefit of Deep Neural Network Approximators

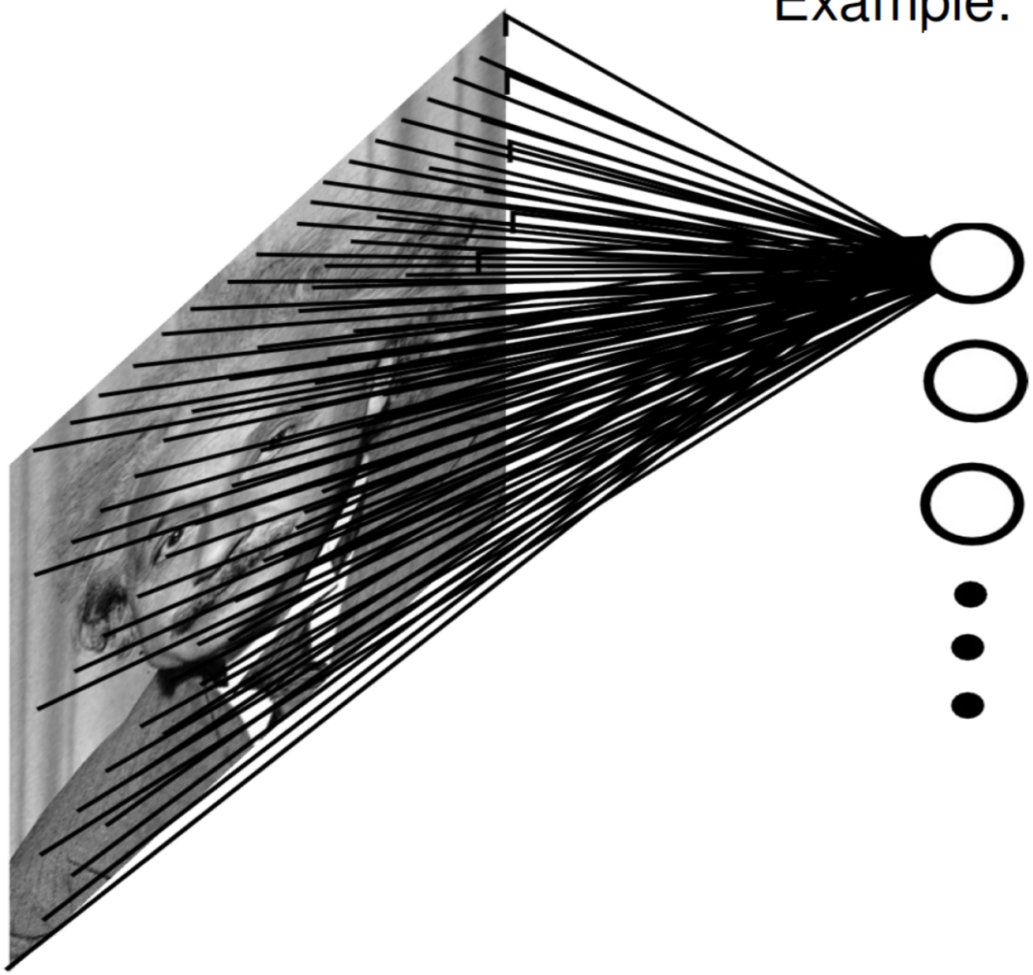
- Linear value function approximators assume value function is a weighted combination of a set of features, where each feature a function of the state
- Linear VFA often work well given the right set of features
- But can require carefully hand designing that feature set
- An alternative is to use a 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

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

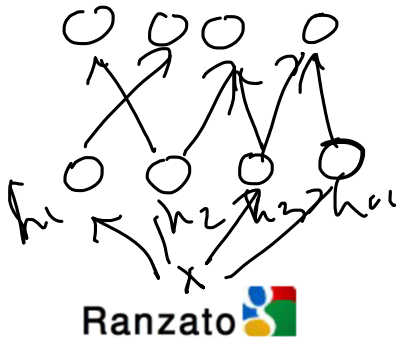


Example: 1000x1000 image

10^6

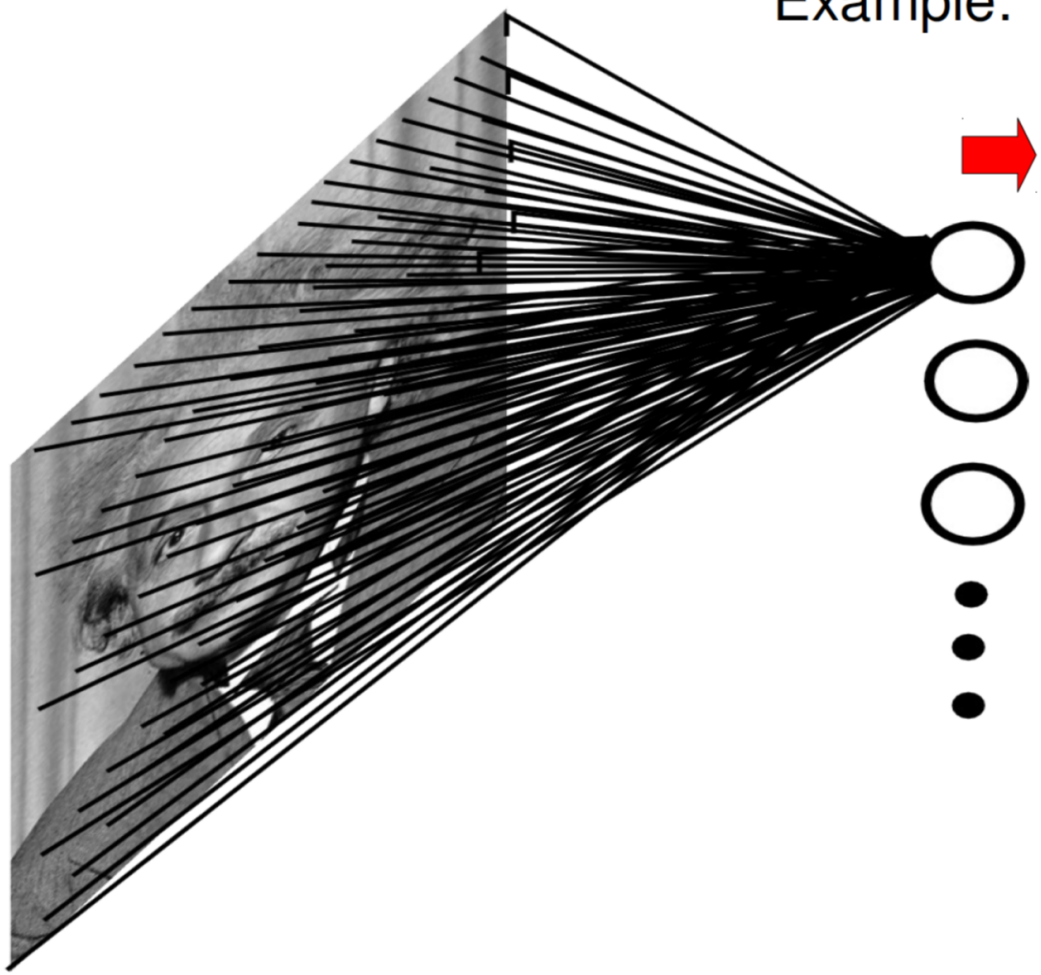
10^6 $\quad$ W $\quad$ X $\quad$ $\swarrow 10^6$

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



Ranzato

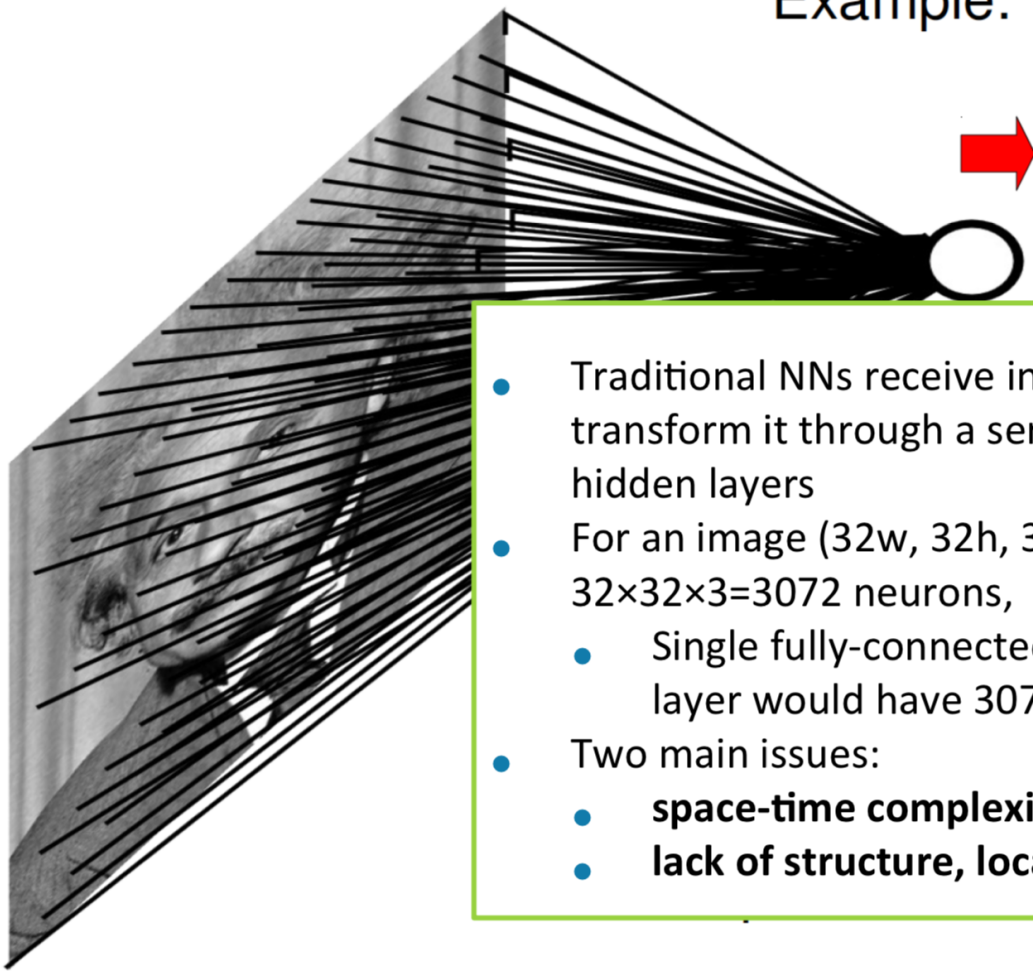
Fully Connected Neural Net



Example: 1000x1000 image
1M hidden units
10^12 parameters!!!

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 32×32×3=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

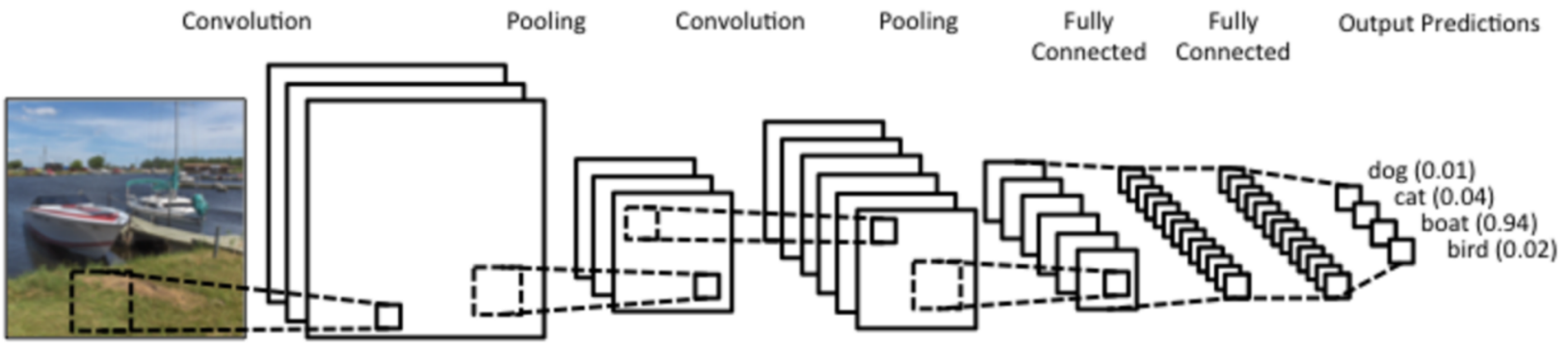
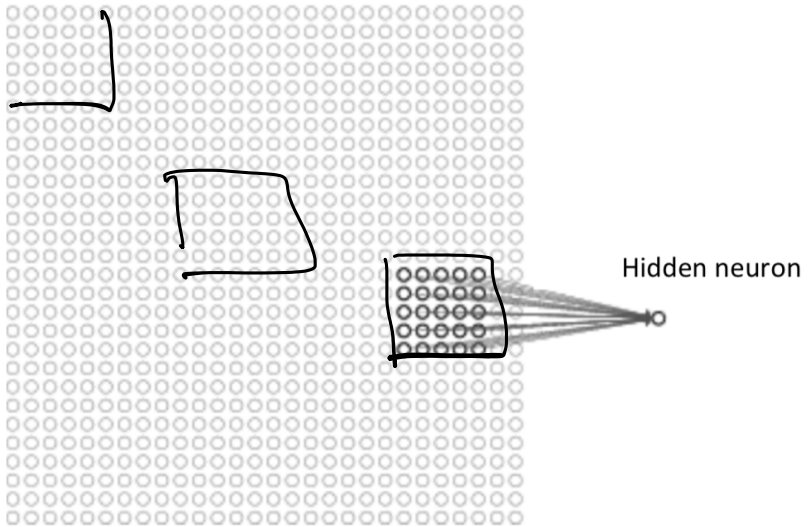
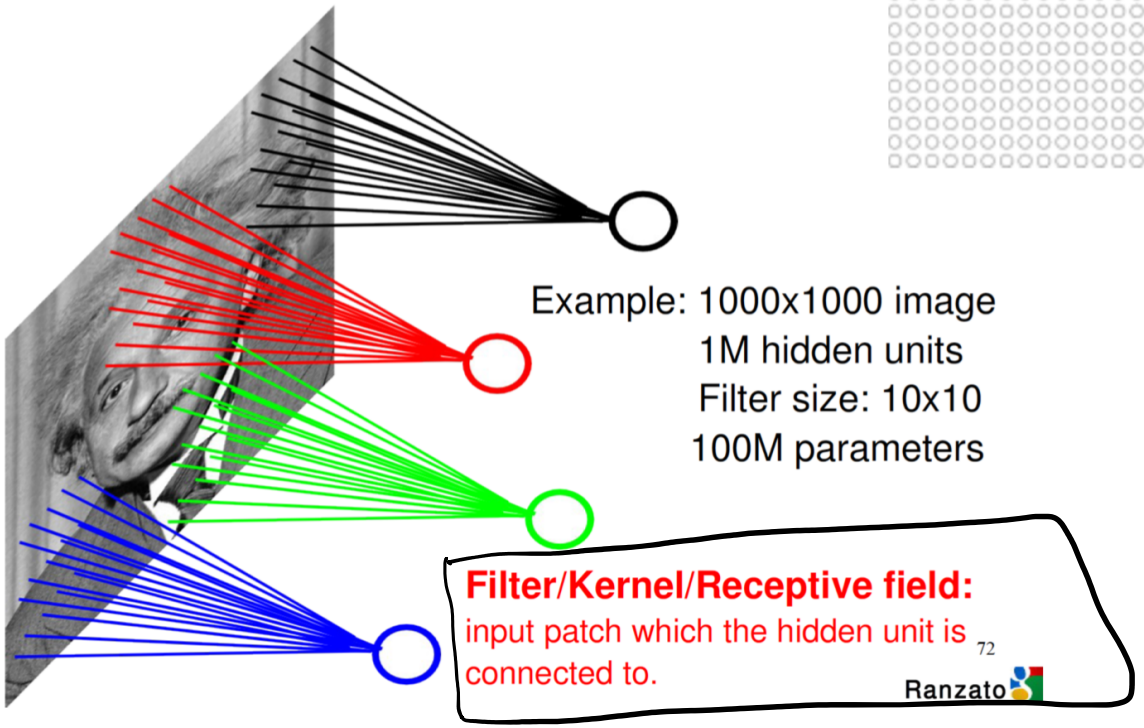


Image: http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png

- 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

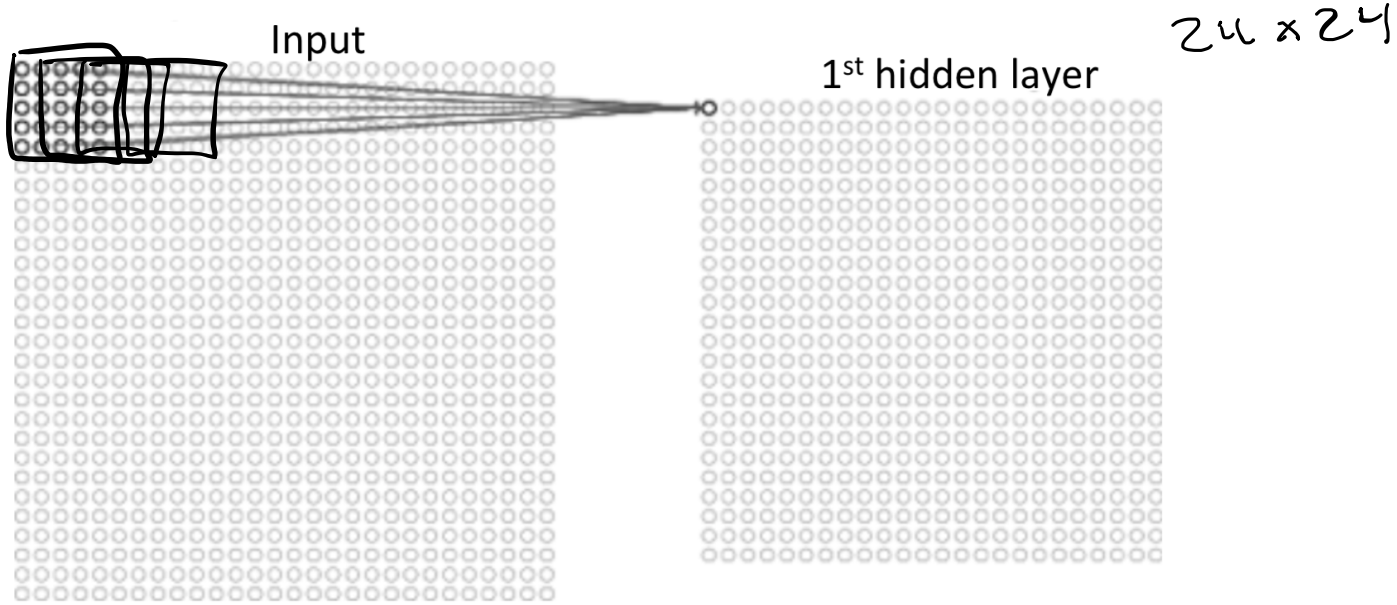




(Filter) Stride

↙ 25 weights

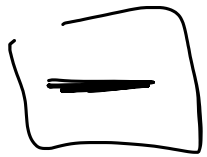
- Slide the 5x5 mask over all the input pixels
- Stride length $= 1$
  - Can use other stride lengths
- Assume input is 28x28, how many neurons in 1st hidden layer?



- 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?

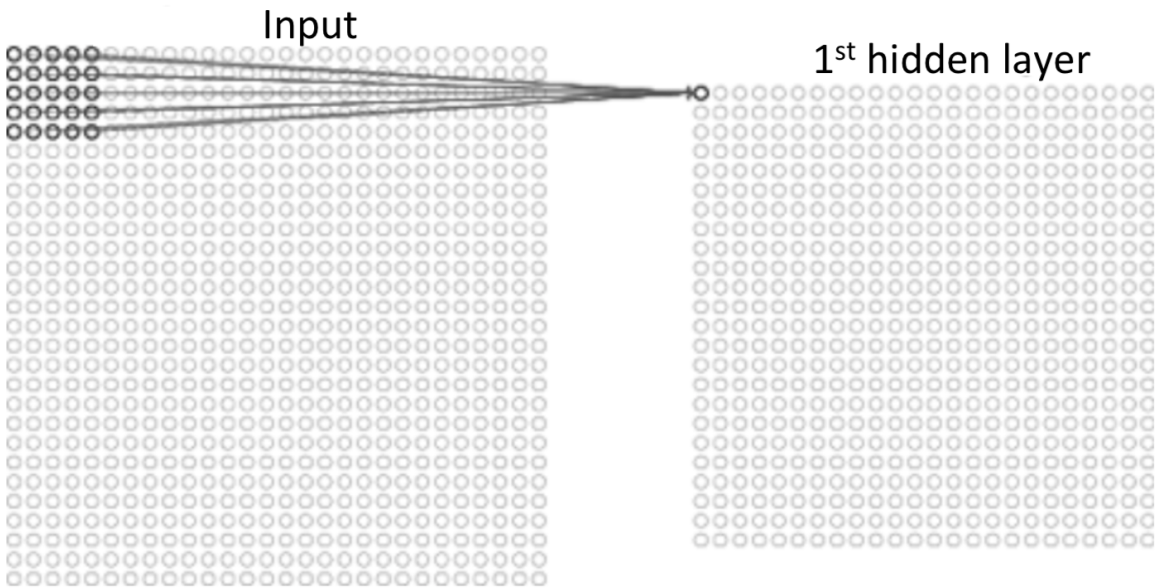


$$g(b + \sum_i w_i x_i)$$

- 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

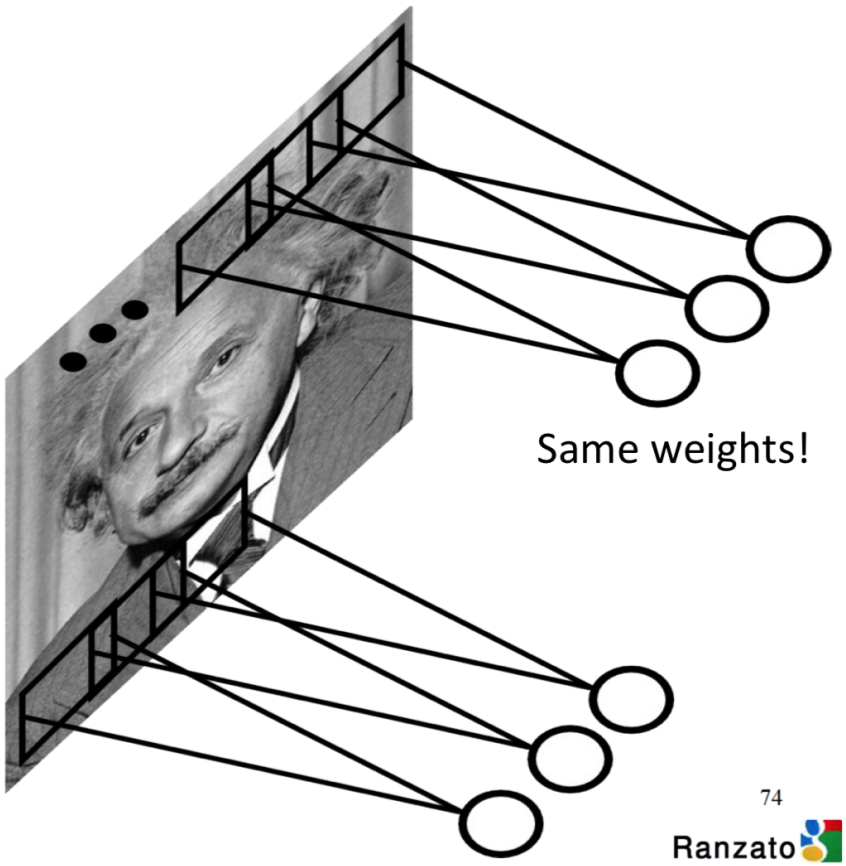


- Receptive field is 5x5

Feature Map

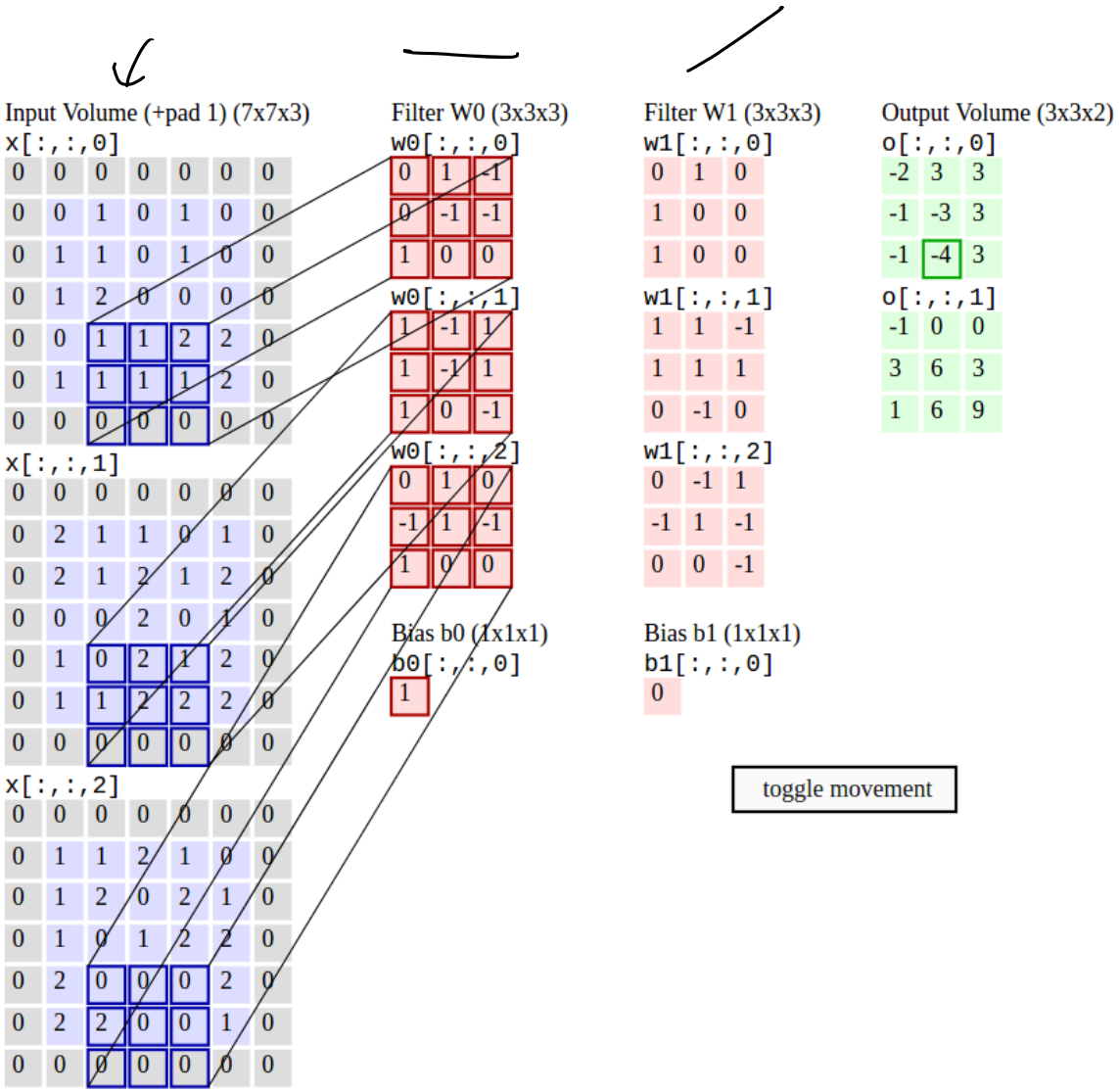
- All the neurons in the first hidden layer *detect exactly the same feature, just at different locations* in the input image.
- **Feature**: the kind of input pattern (e.g., a local edge) that makes the neuron produce a certain response level
- Why does this makes sense?
  - Suppose the weights and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

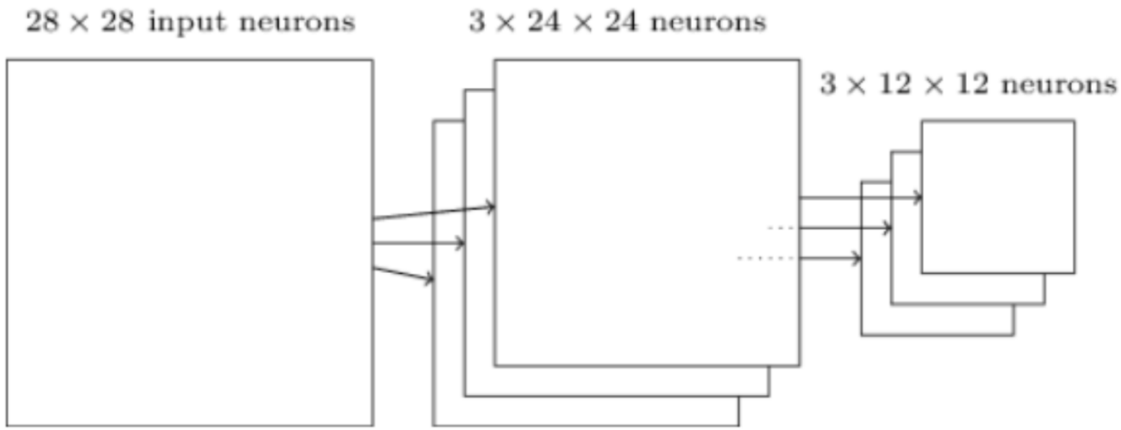
Convolutional Layer: Multiple Filters Ex.[1]



[1] 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

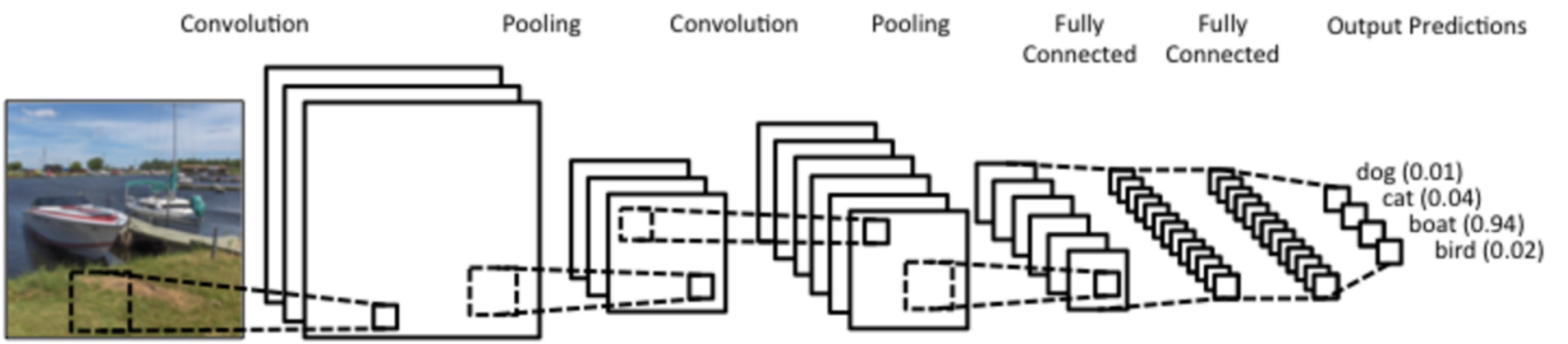


Image: http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png

Table of Contents

Generalization

~1994 TD backgammon
neural nets

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

⇒ 1995 ~ 1998 func approx
to off policy control
+ bootstrapping
can fail to converge
simple cases
that went wrong

mid 2000s - now
DNN ++

~2014 deep mind

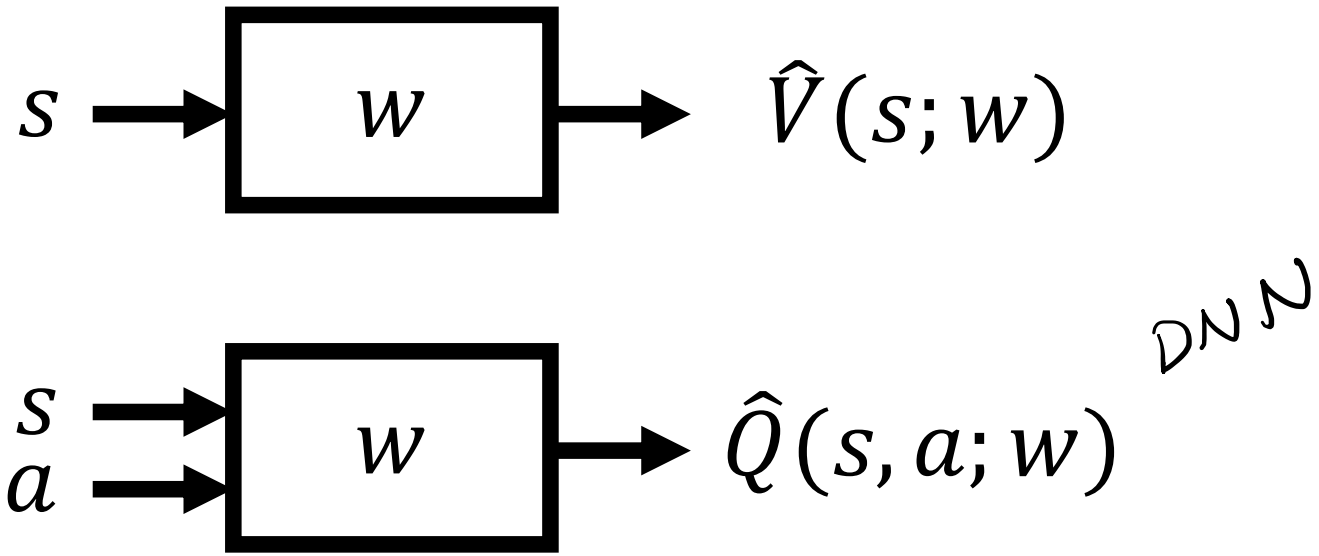
Deep Reinforcement Learning

- Use deep neural networks to represent
  - Value 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 $\boldsymbol{w}$

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



Recall: Action-Value Function Approximation with an Oracle

- $\hat{Q}^{\pi}(s, a; \boldsymbol{w}) \approx Q^{\pi}$ control
- Minimize the mean-squared error between the true action-value function $Q^{\pi}(s, a)$ and the approximate action-value function:

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

- Use stochastic gradient descent to find a local minimum

$$\begin{aligned} -\frac{1}{2}\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &= \mathbb{E}_{\pi}\left[(Q^{\pi}(s, a) - \hat{Q}^{\pi}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}^{\pi}(s, a; \boldsymbol{w})\right] \\ \Delta(\boldsymbol{w}) &= -\frac{1}{2}\alpha \nabla_{\boldsymbol{w}} J(\boldsymbol{w}) \end{aligned}$$

- Stochastic gradient descent (SGD) samples the gradient

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

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

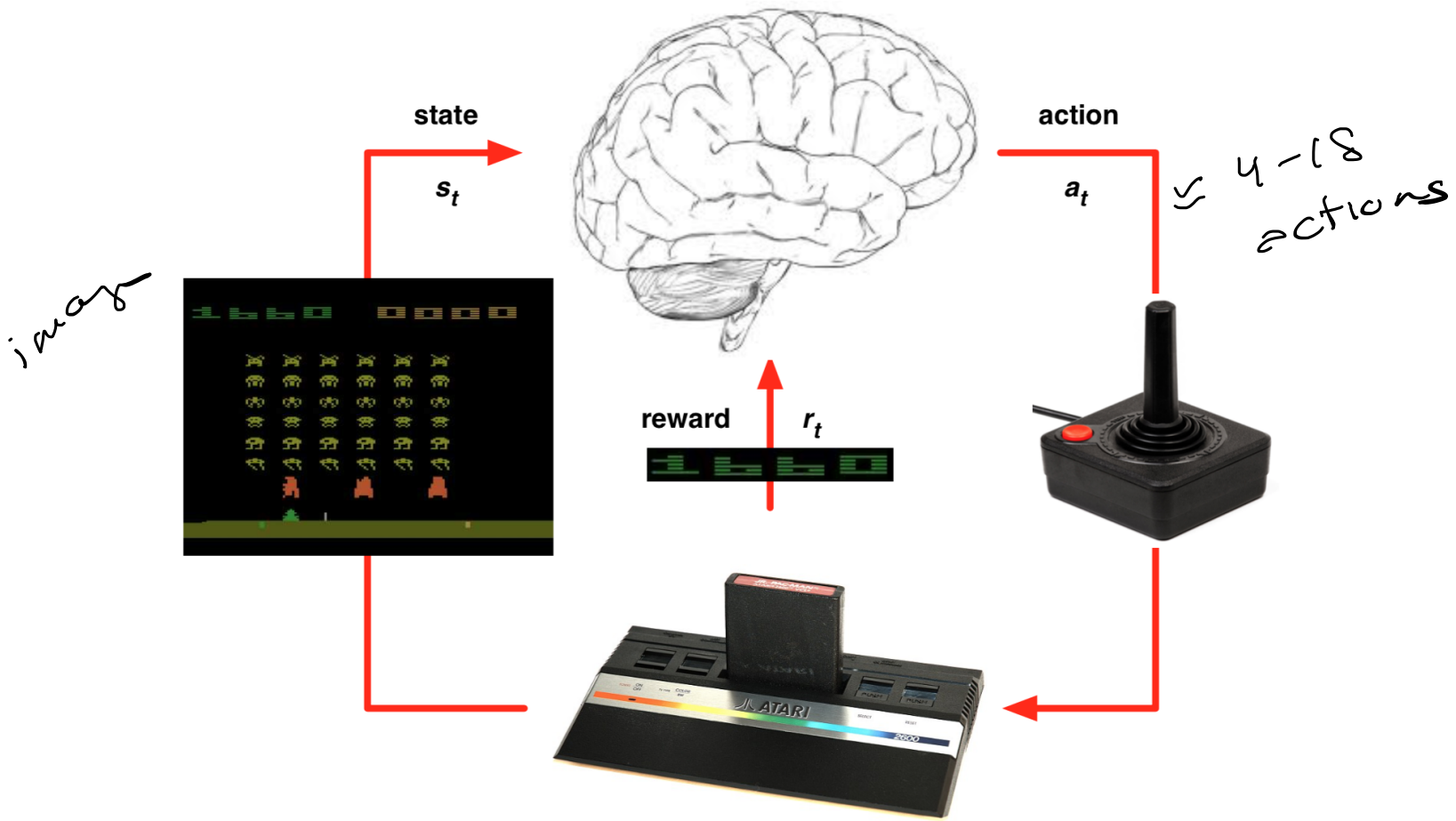
- For SARSA instead use a TD target $r + \gamma \hat{Q}(s_{t+1}, a_{t+1}; \boldsymbol{w})$ which leverages the current function approximation value

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

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

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

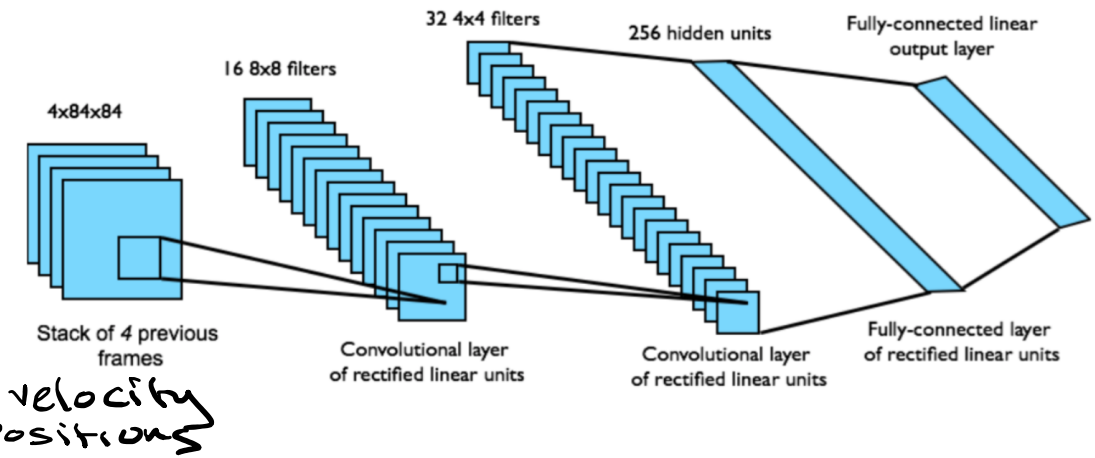
Using these ideas to do Deep RL in Atari



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

you need velocity & position



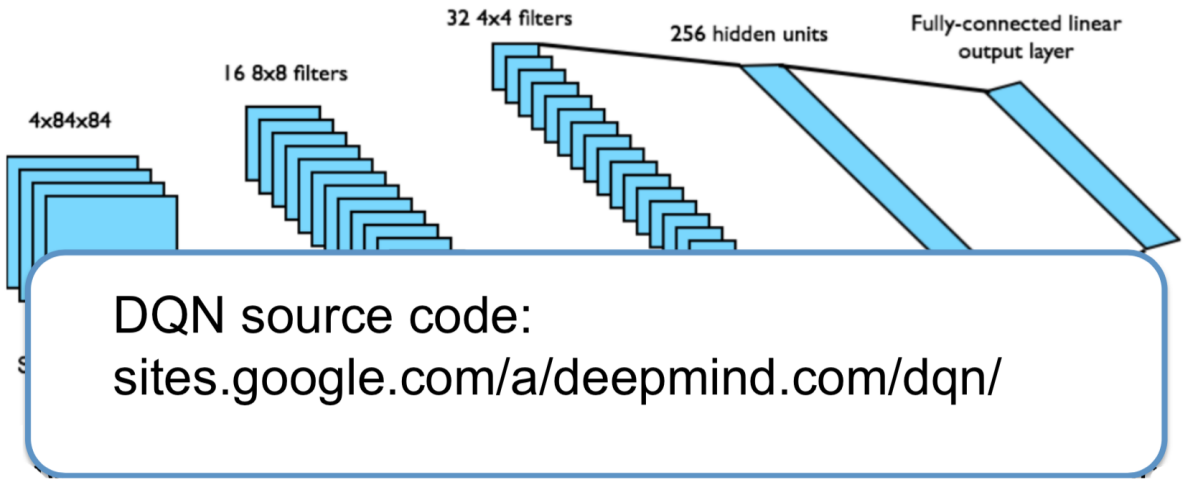
but, where is the input a ?

image Mnih et al Nature 2015

- Network architecture and hyperparameters fixed across all games

DQNs in Atari

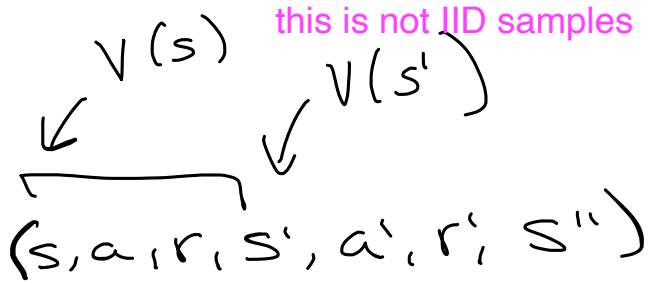
- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



- Network architecture and hyperparameters fixed across all games

Q-Learning with Value Function Approximation

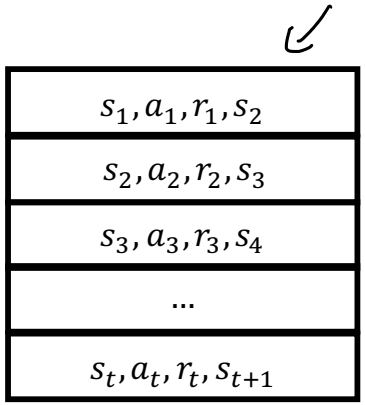
- Minimize MSE loss by stochastic gradient descent
- Converges to the optimal $Q^*(s, a)$ using table lookup representation
- But Q-learning with VFA can diverge
- Two of the issues causing problems:
  - Correlations between samples V(s),V(s')
  - Non-stationary targets
- Deep Q-learning (DQN) addresses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

Q: Is the size of buffer fixed? What is the strategy of inserting & deleting ?
A: different people do different choices.



s, a, r, s'

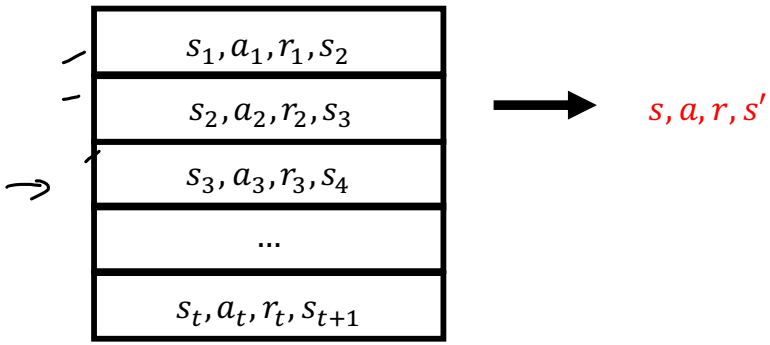
Instead of just using each data point once, you can reuse it and that can be helpful.

- 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: $r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w})$
  - Use stochastic gradient descent to update the network weights

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

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: $r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w})$
  - Use stochastic gradient descent to update the network weights

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

- **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

This is just reducing the noise. If you think of this as a supervised learning problem, we have an input x and output y, the challenge in RL is that our y is changing, if you make it not changing, it's much easier to fit.

- To help improve stability, fix the **target weights** used in the target calculation for multiple updates
- Use a different set of weights to compute target than is being updated
- Let parameters $\boldsymbol{w}^-$ be the set of weights used in the target, and $\boldsymbol{w}$ 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: $r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}^-)$
  - Use stochastic gradient descent to update the network weights

target, fix weight w^-

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

every n $\quad w^- \leftarrow w$

We can periodically update w^- as well, in a fixed schedule, say every n episodes, or every n steps. It's a hyper parameter choice, it's a trade-off between propagating faster and stable. if n=1, then back to TD learning, if n = Inf, never update 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 $\boldsymbol{w}^-$
- Optimizes MSE between Q-network and Q-learning targets
- Uses stochastic gradient descent

· We're typically doing ε-greedy exploration.

ϵ-greedy exploration

DQN

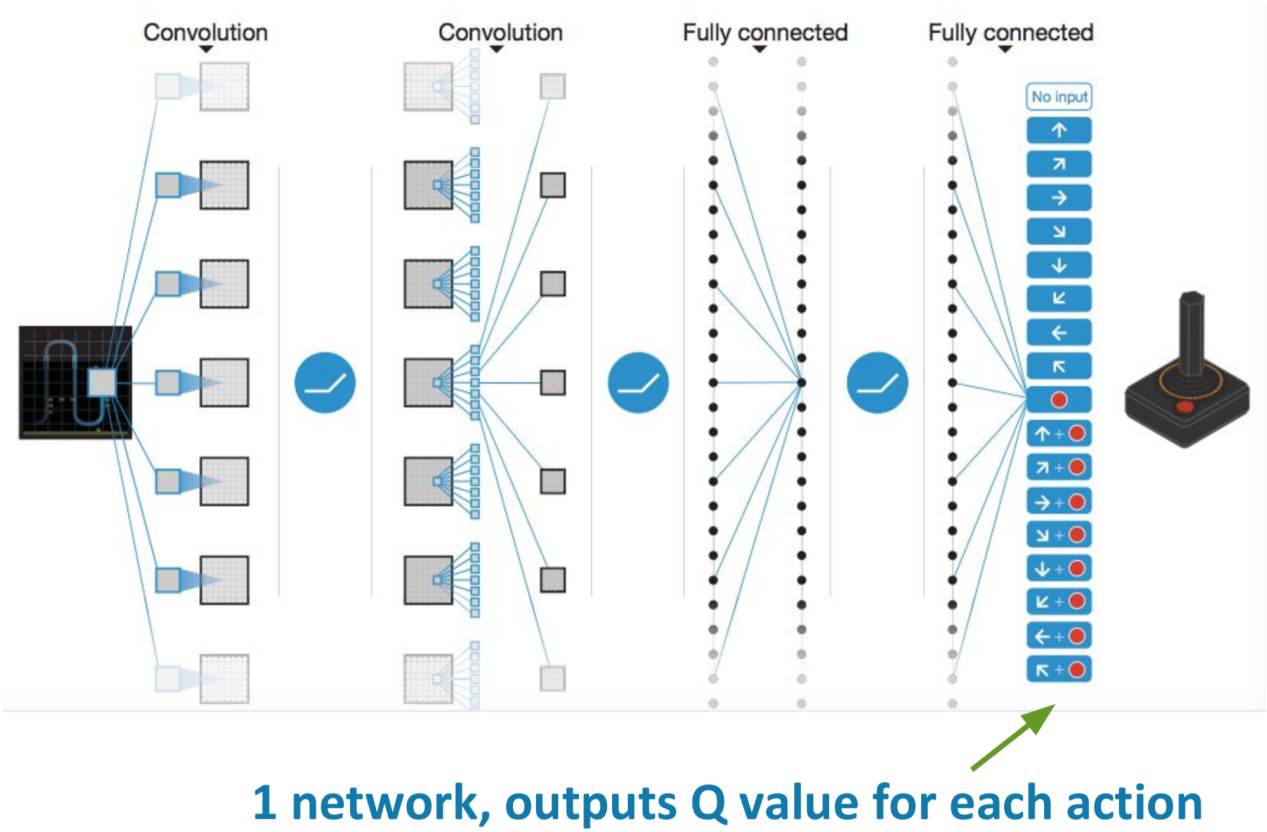


Figure: Human-level control through deep reinforcement learning, Mnih et al, 2015

Demo

DQN Results in Atari

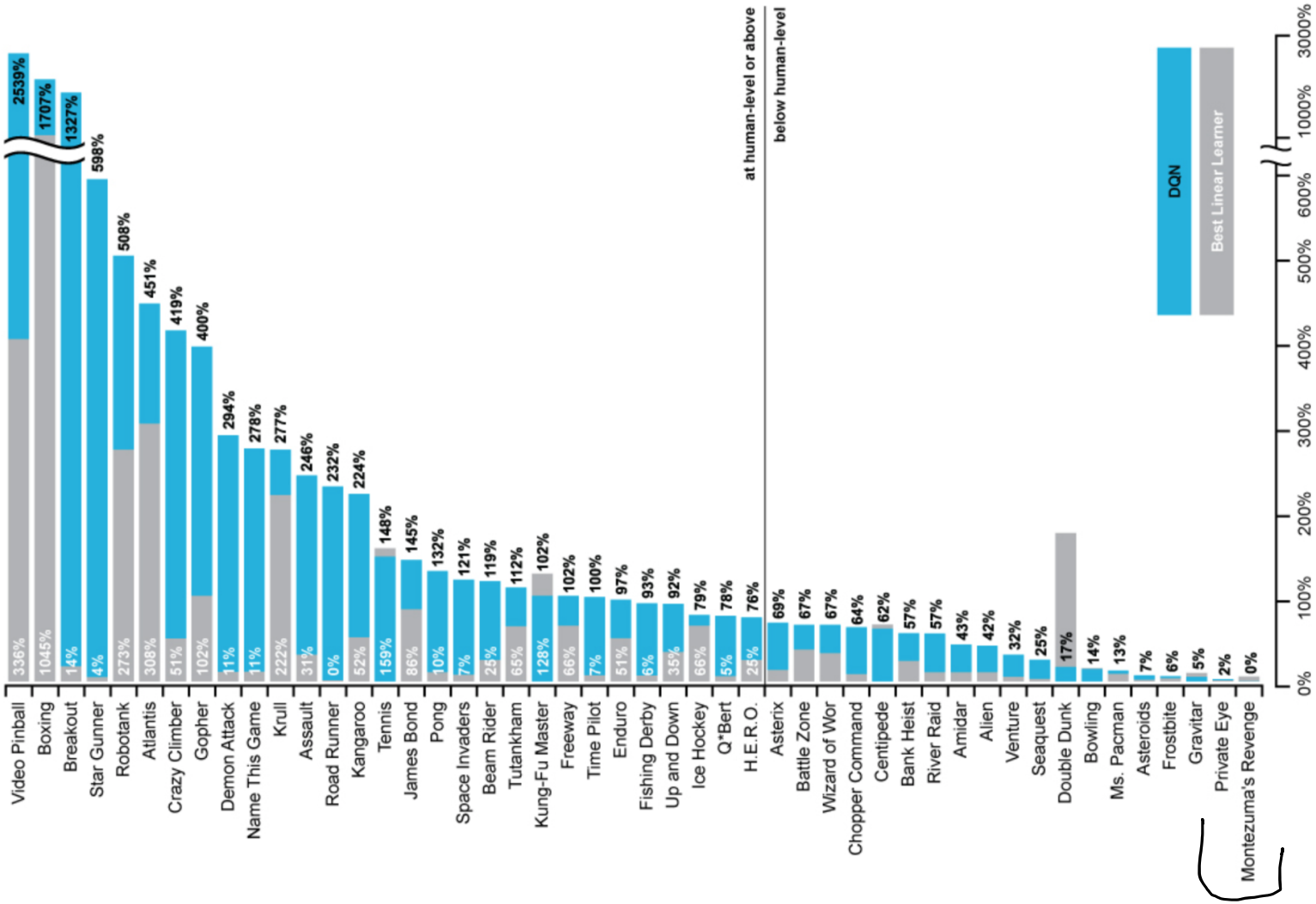


Figure: Human-level control through deep reinforcement learning, Mnih et al, 2015

Which Aspects of DQN were Important for Success?

Game	Linear	Deep Network	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?

Deep RL

- 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)

Double DQN

- Recall maximization bias challenge
  - Max of the estimated state-action values can be a biased estimate of the max
- Double Q-learning

Recall: Double Q-Learning

Initialize $Q_1(s,a)$ and $Q_2(s,a), \forall s \in S, a \in A$ $t = 0$, initial state $s_t = s_0$
loop
  Select a_t using ϵ-greedy $\pi(s) = \arg\max_a Q_1(s_t,a) + Q_2(s_t,a)$
  Observe (r_t, s_{t+1})
  if (with 0.5 probability True) **then**

$$Q_1(s_t,a_t) \leftarrow Q_1(s_t,a_t) + \alpha(r_t + Q_1(s_{t+1}, \arg\max_{a'} Q_2(s_{t+1},a')) - Q_1(s_t,a_t))$$

other network

  else

$$Q_2(s_t,a_t) \leftarrow Q_2(s_t,a_t) + \alpha(r_t + Q_2(s_{t+1}, \arg\max_{a'} Q_1(s_{t+1},a')) - Q_2(s_t,a_t))$$

  end if
  $t = t + 1$
end loop

Double DQN

- Extend this idea to DQN
- Current Q-network $\boldsymbol{w}$ is used to select actions fixed target
- Older Q-network $\boldsymbol{w}^-$ is used to evaluate actions

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \overbrace{\hat{Q}(\underbrace{\arg\max_{a'} \hat{Q}(s', a'; \boldsymbol{w})}_{\text{Action selection: } \boldsymbol{w}}; \underline{\boldsymbol{w}^-})}^{\text{Action evaluation: } \boldsymbol{w}^-} - \hat{Q}(s, a; \boldsymbol{w}))$$

Double DQN

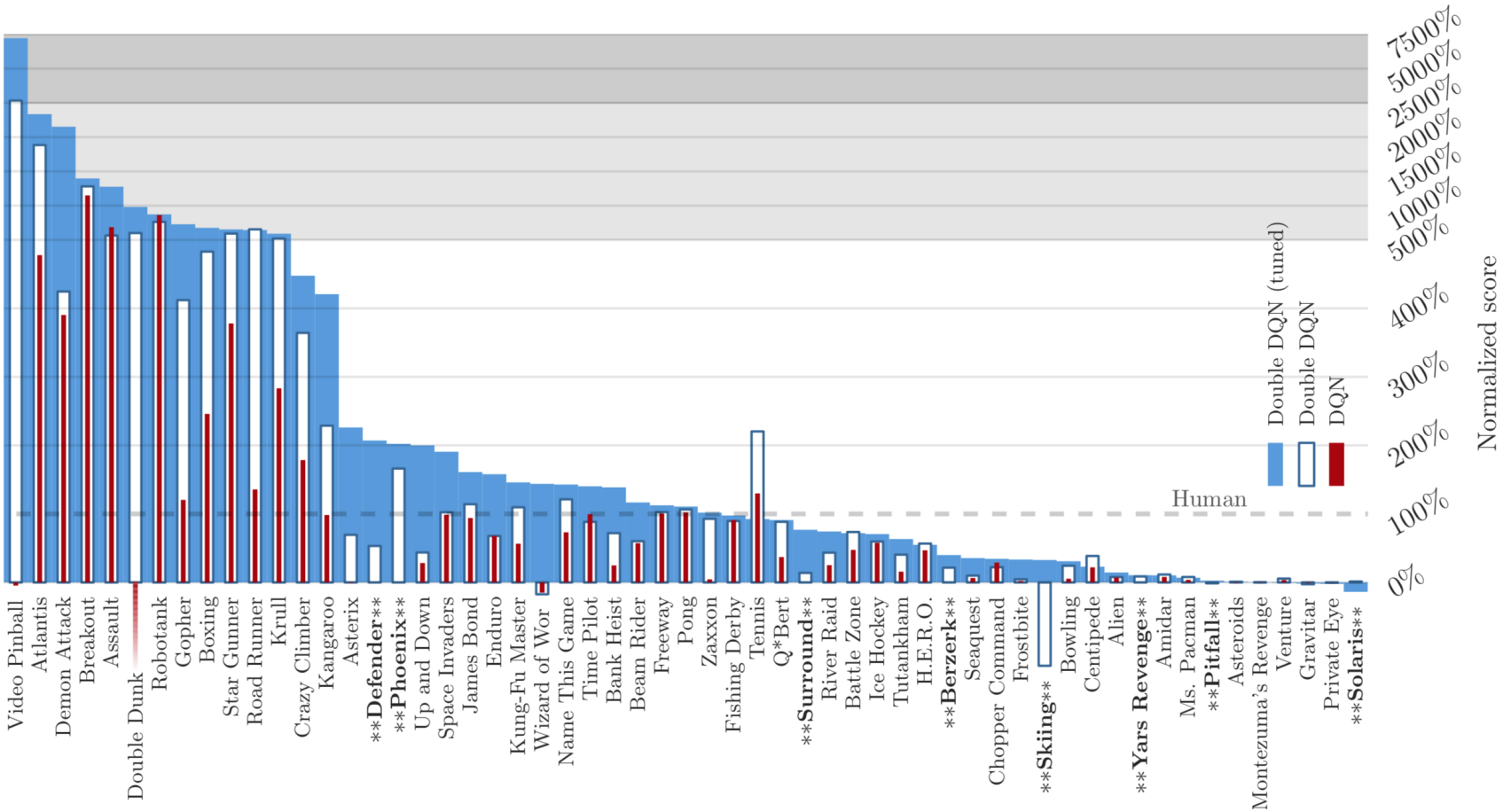


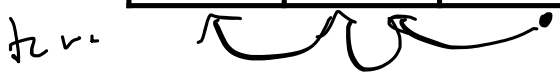
Figure: van Hasselt, Guez, Silver, 2015

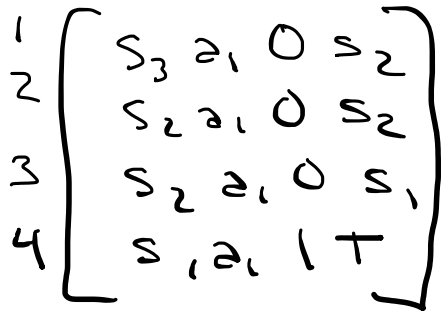
Deep RL

- 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!)
  - 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)

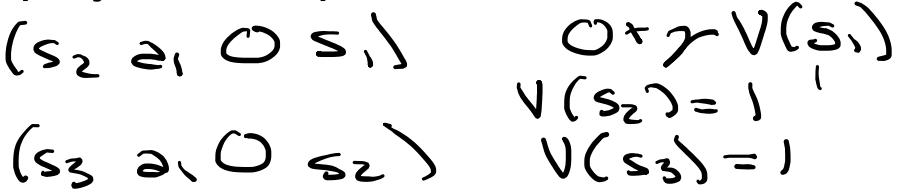
Refresher: Mars Rover Model-Free Policy Evaluation

s_1	s_2	s_3	s_4	s_5	s_6	s_7
$R(s_1) = +1$ *Okay Field Site*	$R(s_2) = 0$	$R(s_3) = 0$	$R(s_4) = 0$	$R(s_5) = 0$	$R(s_6) = 0$	$R(s_7) = +10$ *Fantastic Field Site*





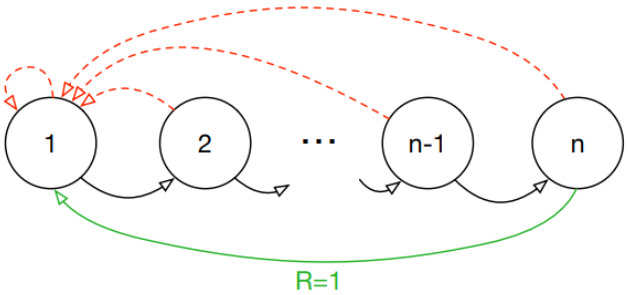
- Mars rover: R = [1 0 0 0 0 0 +10] for any action
- $\pi(s) = a_1 \ \forall s, \ \gamma = 1$. any action from s_1 and s_7 terminates episode
- Trajectory = (s_3, a_1, 0, s_2, a_1, 0, s_2, a_1, 0, s_1, a_1, 1, terminal)
- First visit MC estimate of V of each state? [1 1 1 0 0 0 0]
- Every visit MC estimate of V of s_2? 1
- TD estimate of all states (init at 0) with $\alpha = 1$ is [1 0 0 0 0 0 0]
- Now get to chose 2 "replay" backups to do. Which should we pick to get best estimate?



Impact of Replay?

- In tabular TD-learning, **order** of replaying updates could help speed learning
- Repeating some updates seem to better propagate info than others
- Systematic ways to prioritize updates?

Potential Impact of Ordering Episodic Replay Updates



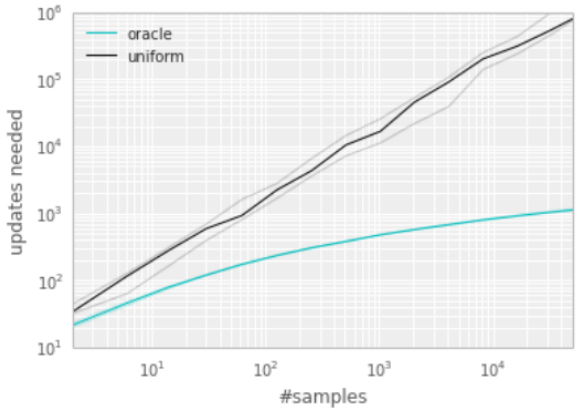


Figure: Schaul, Quan, Antonoglou, Silver ICLR 2016

- Schaul, Quan, Antonoglou, Silver ICLR 2016
- Oracle: picks (s, a, r, s') tuple to replay that will minimize global loss
- Exponential improvement in convergence
  - Number of updates needed to converge
- Oracle is not a practical method but illustrates impact of ordering

Prioritized Experience Replay

- Let i be the index of the i-the tuple of experience (s_i, a_i, r_i, s_{i+1})
- Sample tuples for update using priority function
- Priority of a tuple i is proportional to DQN error

$$p_i = \left| r + \gamma \max_{a'} Q(s_{i+1}, a'; \boldsymbol{w}^-) - Q(s_i, a_i; \boldsymbol{w}) \right|$$

TD error

current

- Update p_i every update
- p_i for new tuples is set to 0
- One method[1]: proportional (stochastic prioritization)

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

[1] See paper for details and an alternative

Check Your Understanding

- Let i be the index of the i-the tuple of experience (s_i, a_i, r_i, s_{i+1})
- Sample tuples for update using priority function
- Priority of a tuple i is proportional to DQN error

$$p_i = \left| r + \gamma \max_{a'} Q(s_{i+1}, a'; \boldsymbol{w}^-) - Q(s_i, a_i; \boldsymbol{w}) \right|$$

- Update p_i every update
- p_i for new tuples is set to 0
- One method[1]: proportional (stochastic prioritization)

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

- $\alpha = 0$ yields what rule for selecting among existing tuples? uniform

[1] See paper for details and an alternative

Performance of Prioritized Replay vs Double DQN

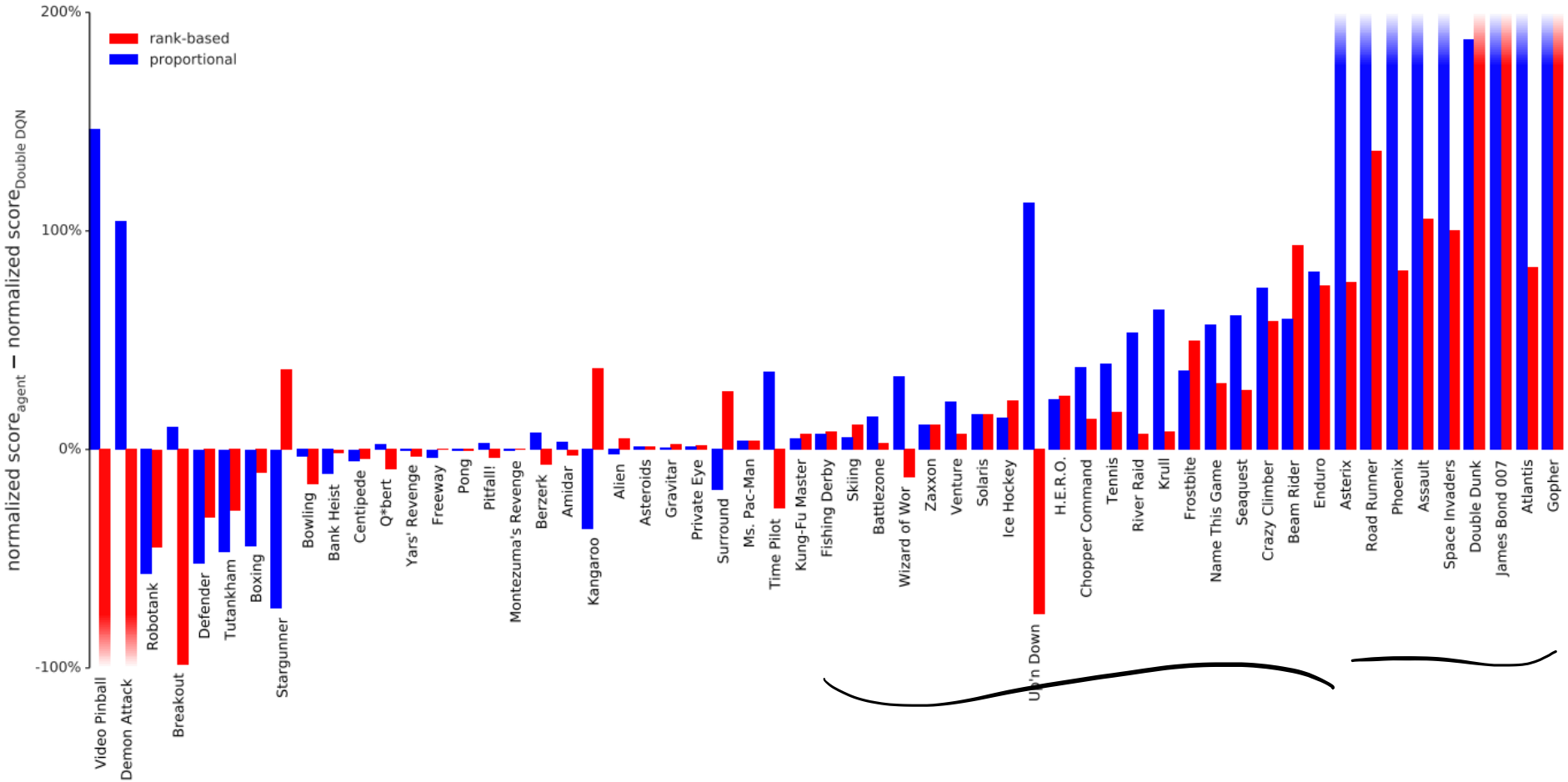


Figure: Schaul, Quan, Antonoglou, Silver ICLR 2016

Deep RL

- 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!)
  - 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)

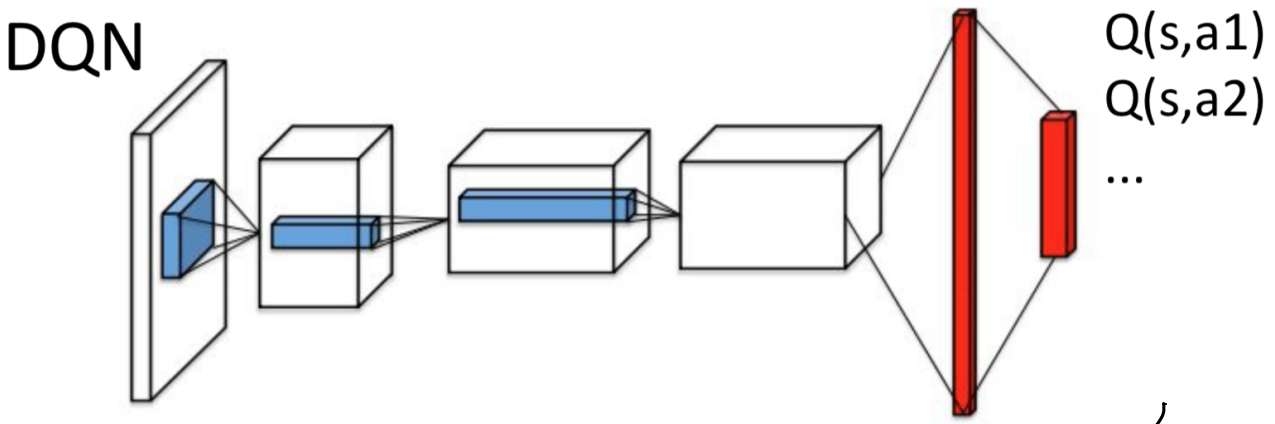
Value & Advantage Function

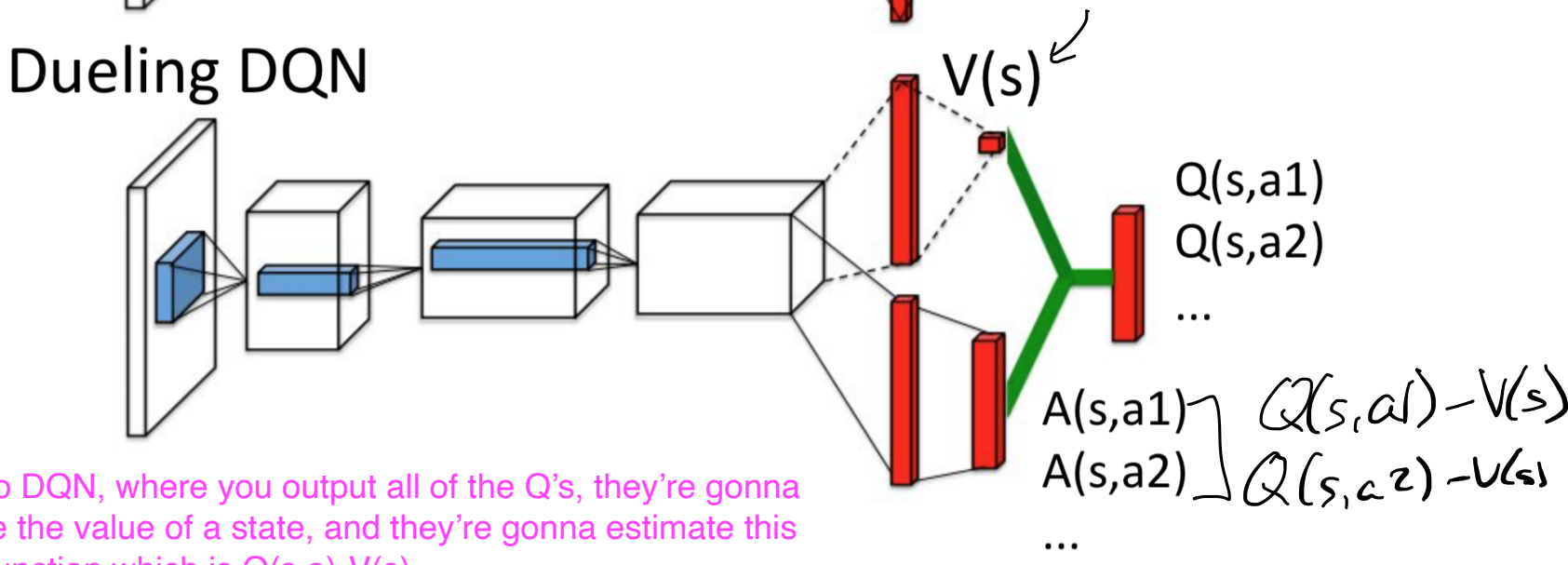
- Intuition: Features need to pay attention to determine value may be different than those need to determine action benefit
- E.g.
  - Game score may be relevant to predicting $V(s)$
  - But not necessarily in indicating relative action values
- Advantage function (Baird 1993)

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

You wanna know , how much better or worse taking a particular action is versus following the current policy.
That really like I don't care about estimating the value of a state, I care about being able to understand which of the actions has the better value.

Dueling DQN





In contrast to DQN, where you output all of the Q's, they're gonna first estimate the value of a state, and they're gonna estimate this advantage function which is Q(s,a)-V(s).

Wang et.al., ICML, 2016

Identifiability

- Advantage function

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

- Identifiable?

No.

Identifiability

- Advantage function

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

- Unidentifiable
- Option 1: Force $A(s,a) = 0$ if a is action taken

$$\hat{Q}(s,a;\boldsymbol{w}) = \hat{V}(s;\boldsymbol{w}) + \left(\hat{A}(s,a;\boldsymbol{w}) - \max_{a' \in \mathcal{A}} \hat{A}(s,a';\boldsymbol{w})\right)$$

- Option 2: Use mean as baseline (more stable)

$$\hat{Q}(s,a;\boldsymbol{w}) = \hat{V}(s;\boldsymbol{w}) + \left(\hat{A}(s,a;\boldsymbol{w}) - \frac{1}{|\mathcal{A}|}\sum_{a'} \hat{A}(s,a';\boldsymbol{w})\right)$$

Dueling DQN V.S. Double DQN with Prioritized Replay

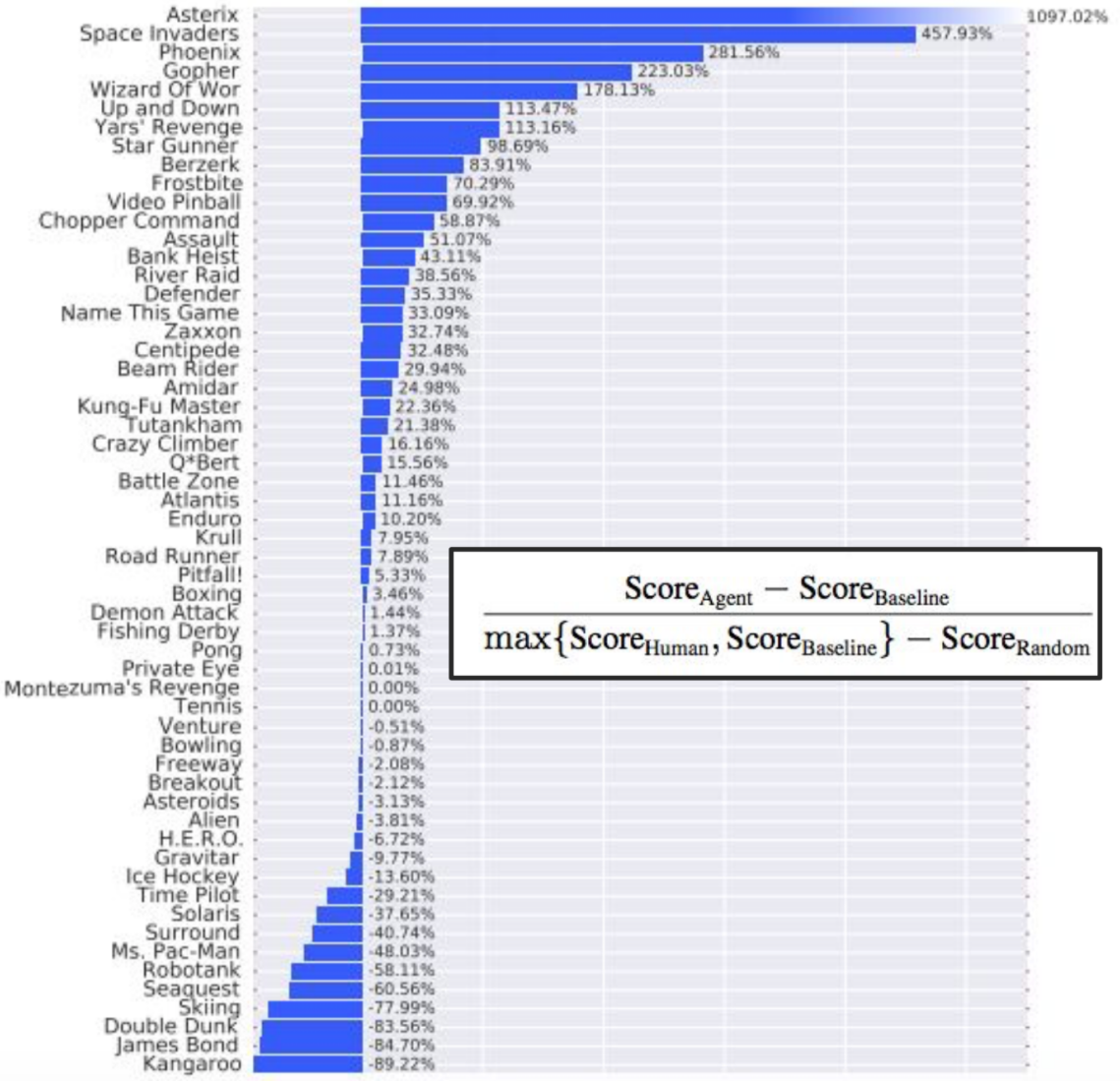


Figure: Wang et al, ICML 2016

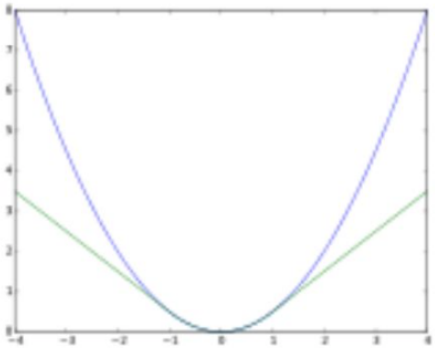
Practical Tips for DQN on Atari (from J. Schulman)

- DQN is more reliable on some Atari tasks than others. Pong is a reliable task: if it doesn't achieve good scores, something is wrong
- Large replay buffers improve robustness of DQN, and memory efficiency is key
  - Use uint8 images, don't duplicate data
- Be patient. DQN converges slowly—for ATARI it's often necessary to wait for 10-40M frames (couple of hours to a day of training on GPU) to see results significantly better than random policy
- In our Stanford class: Debug implementation on small test environment

Practical Tips for DQN on Atari (from J. Schulman) cont.

- Try Huber loss on Bellman error

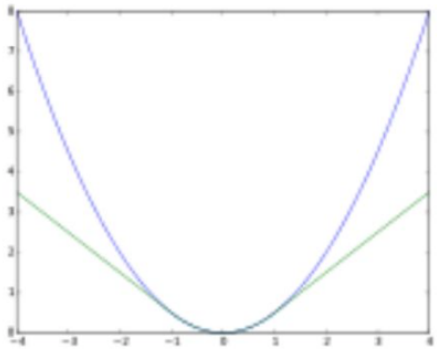
$$L(x) = \begin{cases} \frac{x^2}{2} & \text{if } |x| \leq \delta \\ \delta|x| - \frac{\delta^2}{2} & \text{otherwise} \end{cases}$$



Practical Tips for DQN on Atari (from J. Schulman) cont.

- Try Huber loss on Bellman error

$$L(x) = \begin{cases} \frac{x^2}{2} & \text{if } |x| \leq \delta \\ \delta|x| - \frac{\delta^2}{2} & \text{otherwise} \end{cases}$$



- Consider trying Double DQN—significant improvement from small code change in Tensorflow.
- To test out your data pre-processing, try your own skills at navigating the environment based on processed frames
- Always run at least two different seeds when experimenting
- Learning rate scheduling is beneficial. Try high learning rates in initial exploration period
- Try non-standard exploration schedules

Table of Contents

Class Structure

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