

Lecture 6: CNNs and Deep Q Learning²

Emma Brunskill

CS234 Reinforcement Learning.

Winter 2018

²With 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

1 Convolutional Neural Nets (CNNs)

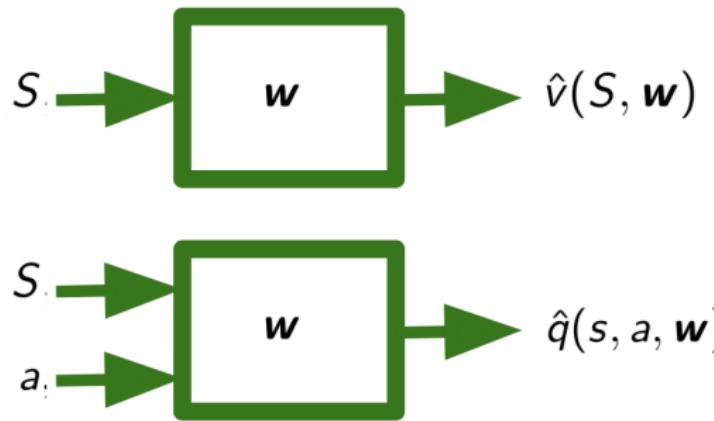
2 Deep Q Learning

Class Structure

- Last time: Value function approximation and deep learning
- This time: Convolutional neural networks and deep RL
- Next time: Imitation learning

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: 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 $S \rightarrow f(S) \rightarrow V$
- 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: use 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
- Last time discussed basic feedforward deep networks

Table of Contents

1 Convolutional Neural Nets (CNNs)

2 Deep Q Learning

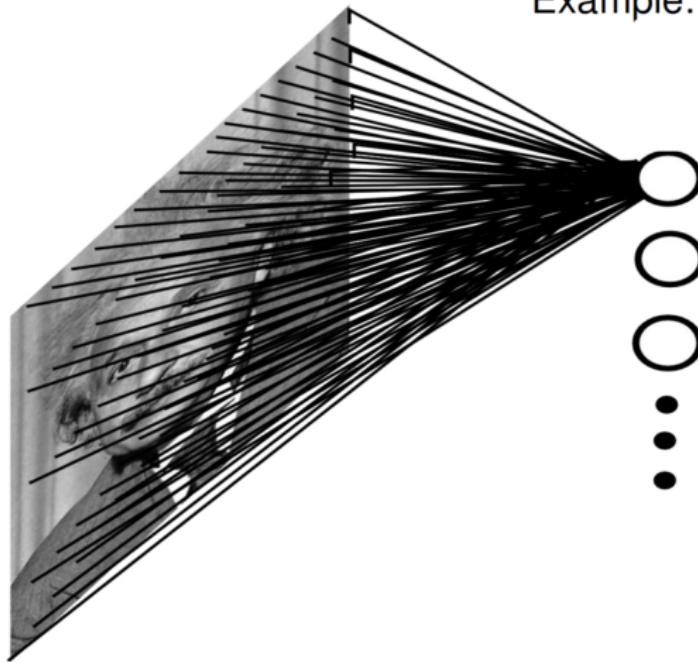
Why Do We Care About CNNs?

- 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

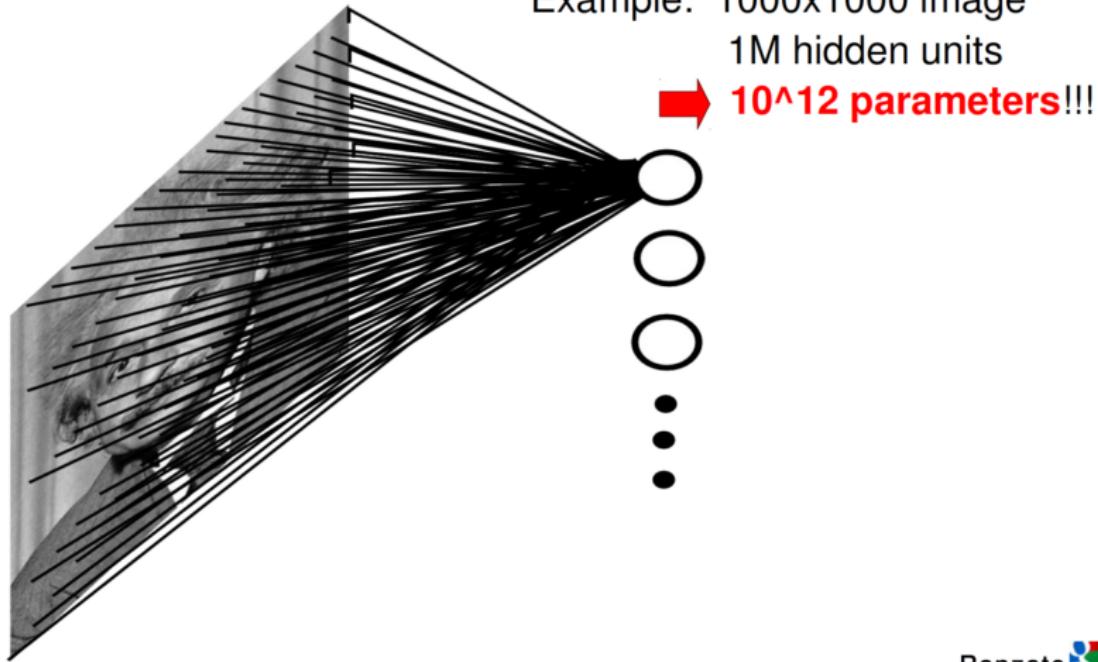
Example: 1000x1000 image



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

10^6
parameters
+ 1
bias

Fully Connected Neural Net



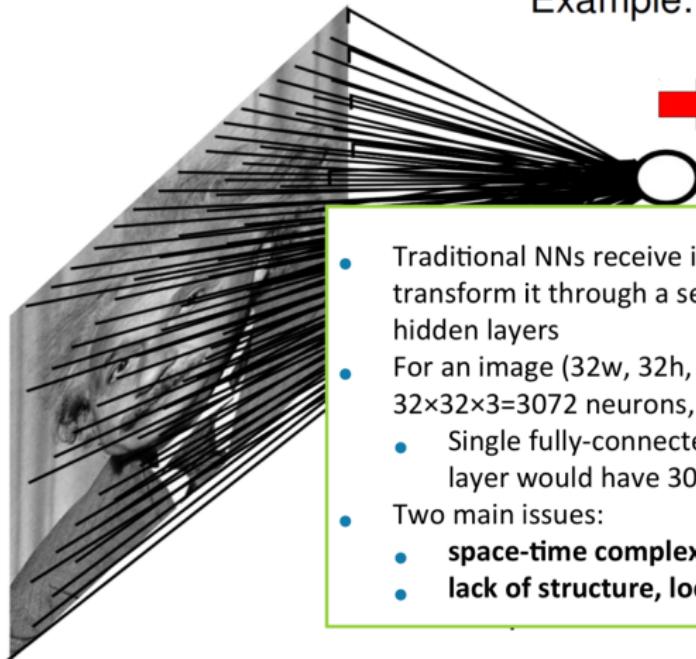
Fully Connected Neural Net

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

Image Features

- Want uniqueness
- Want invariance
- Geometric invariance: translation, rotation, scale
- Photometric invariance: brightness, exposure, ...
- Leads to unambiguous matches in other images or w.r.t. to known entities of interest
- Look for “interest points”: image regions that are unusual
- Coming up with these is nontrivial

Convolutional NN

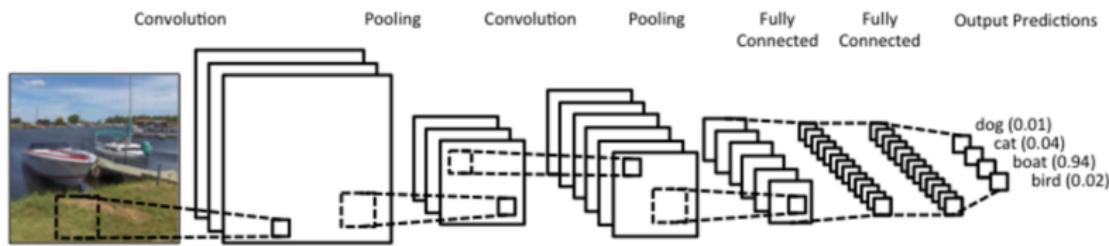
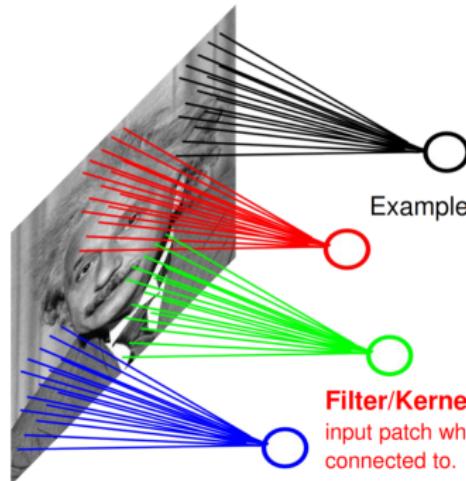


Image: <http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png>

- 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



Example: 1000x1000 image

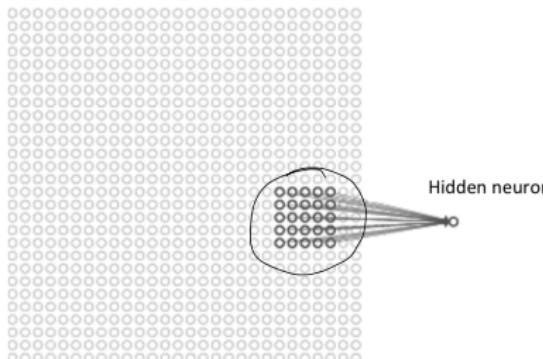
1M hidden units

Filter size: 10x10

100M parameters

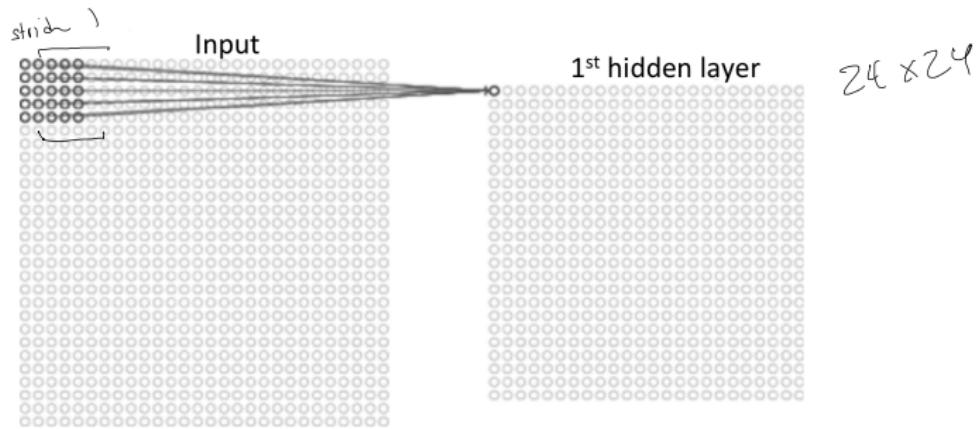
100 inputs

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



(Filter) Stride

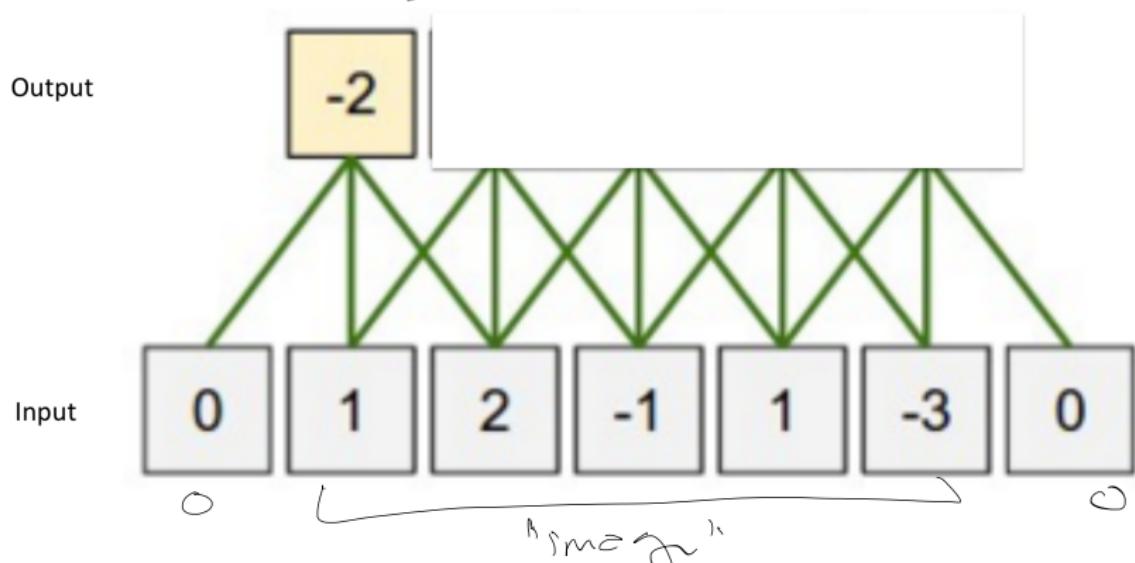
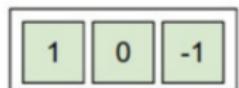
- Slide the 5×5 mask over all the input pixels
- Stride length = 1
 - Can use other stride lengths
- Assume input is 28×28 , how many neurons in 1st hidden layer?



Stride and Zero Padding

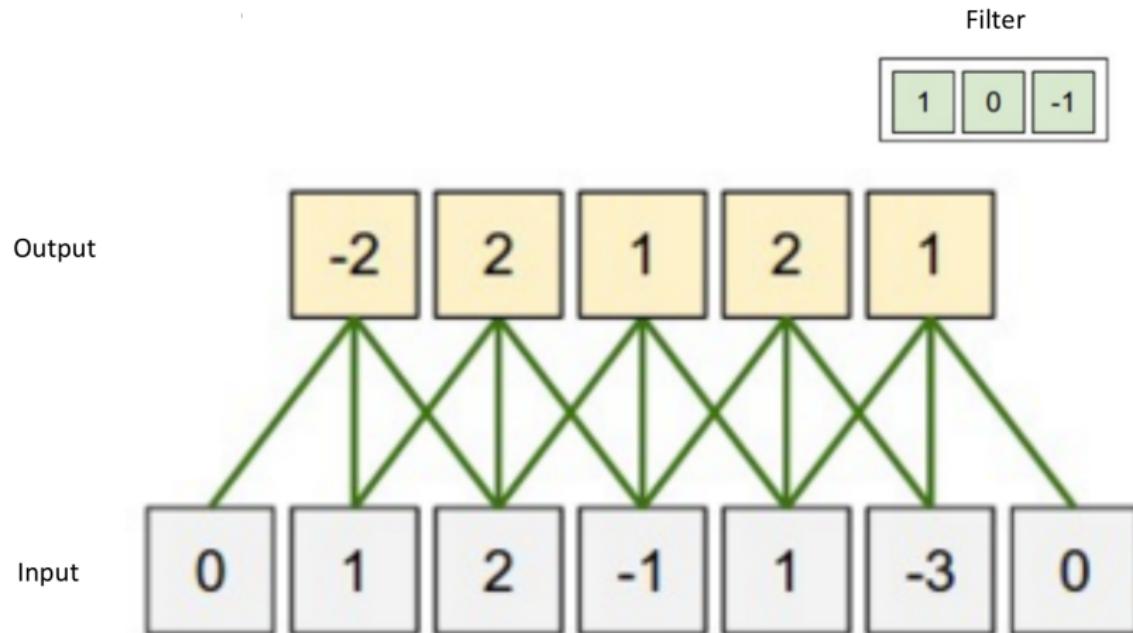
$$0 \cdot 1 + 1 \cdot 0 + 2 \cdot -1 = -2$$

Filter



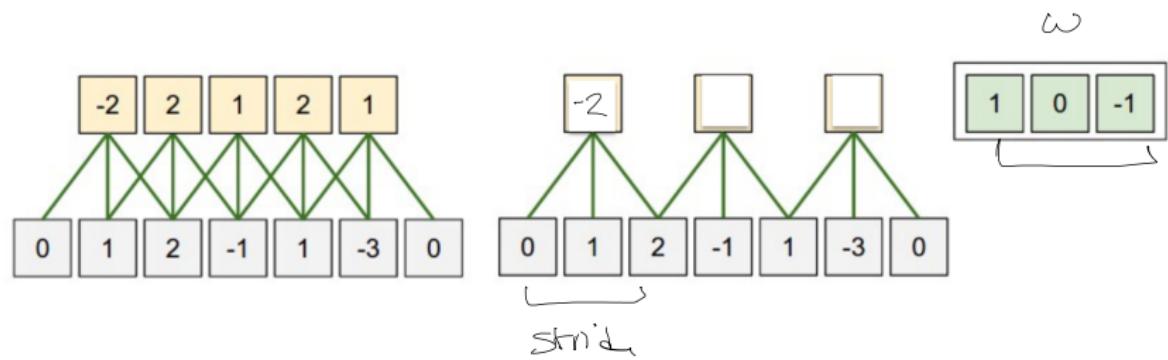
- Stride: how far (spatially) move over filter
- Zero padding: how many 0s to add to either side of input layer

Stride and Zero Padding

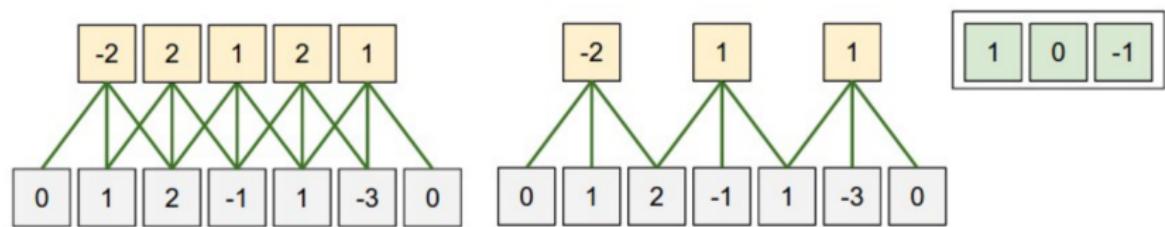


- Stride: how far (spatially) move over filter
- Zero padding: how many 0s to add to either side of input layer

What is the Stride and the Values in the Second Example?



Stride is 2



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?

$$g(b + \sum_i^{\text{bias}} w_i x_i)$$

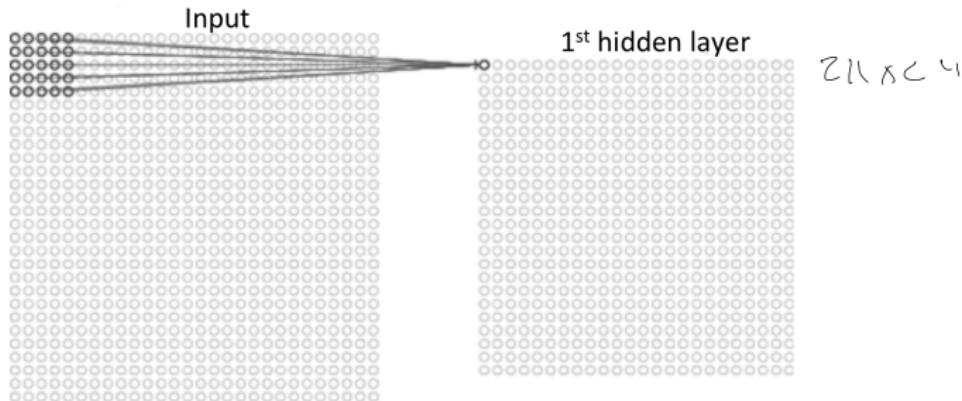
bias *input*
i *receptive*

- Sum over i is *only over the neurons in the receptive field* of the hidden layer neuron (5×5 input ex.)
- *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

5x5 receptive field



- Receptive field is 5x5
- Number of parameters for 1st hidden neuron?
- Number of parameters for entire layer of hidden neurons?

$$b + \sum_i w_i x_i$$

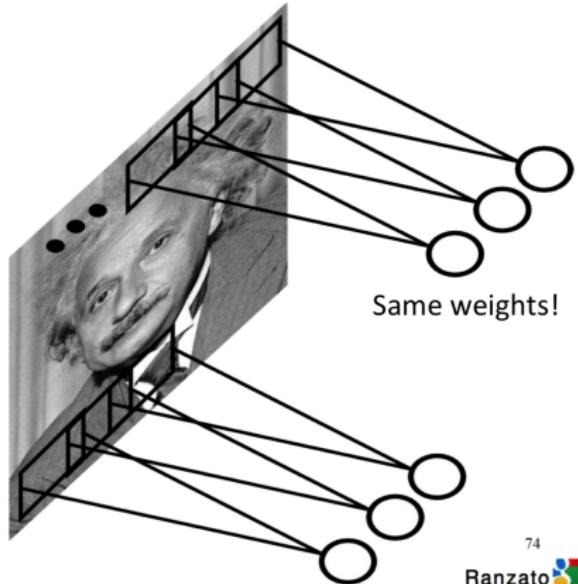
26

26

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 Image Filters

*3x3
neutral*

1	1	1
1	1	1
1	1	1

Unweighted 3x3 smoothing kernel

0	1	0
1	4	1
0	1	0

Weighted 3x3 smoothing kernel with Gaussian blur

0	-1	0
-1	5	-1
0	-1	0

Kernel to make image sharper

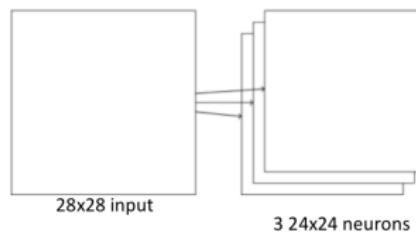
-1	-1	-1
-1	9	-1
-1	-1	-1

Intensified sharper image



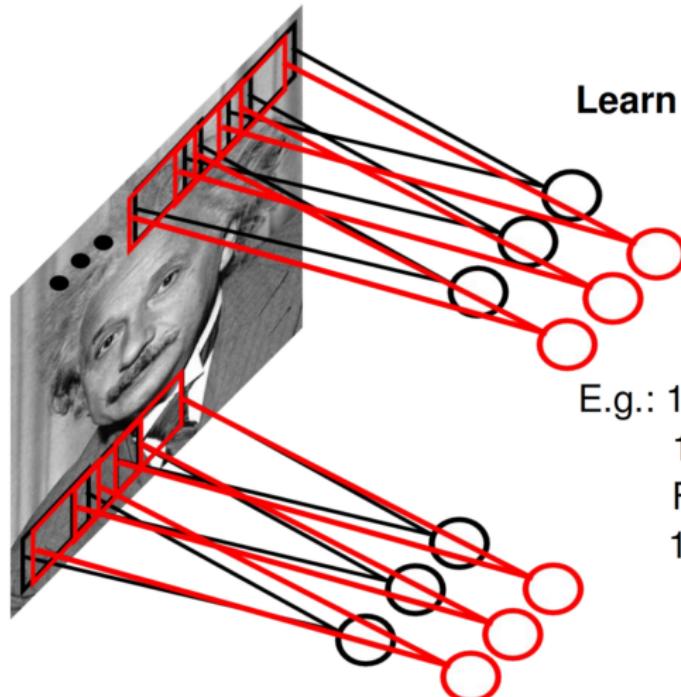
Why Only 1 Filter?

- At the i -th hidden layer n filters can be active in parallel
- A **bank of convolutional filters**, each learning a different feature (different weights and bias)



- 3 feature maps, each defined by a set of 5×5 shared weights & 1 bias
- Network detects 3 different kinds of features, with each feature being detectable across the entire image

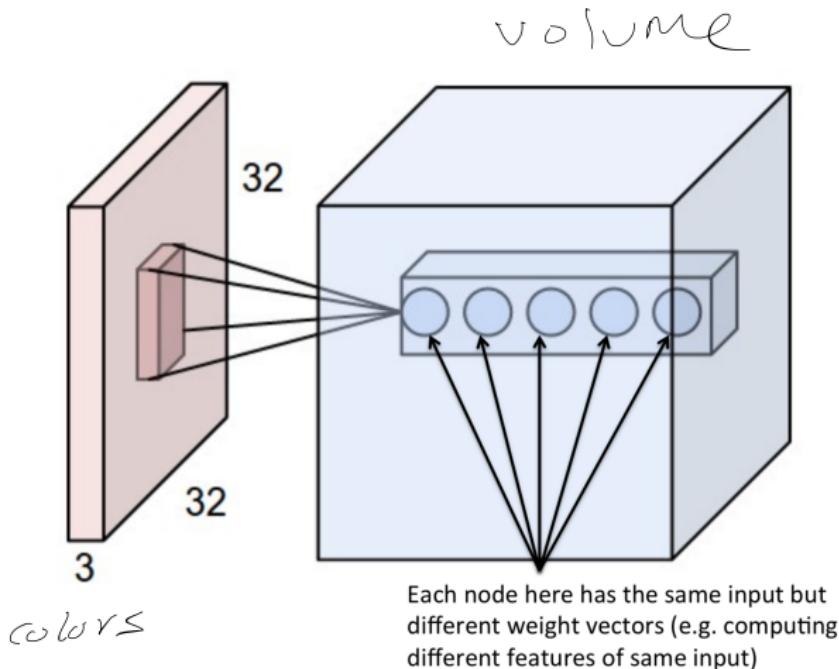
Convolutional Net



75

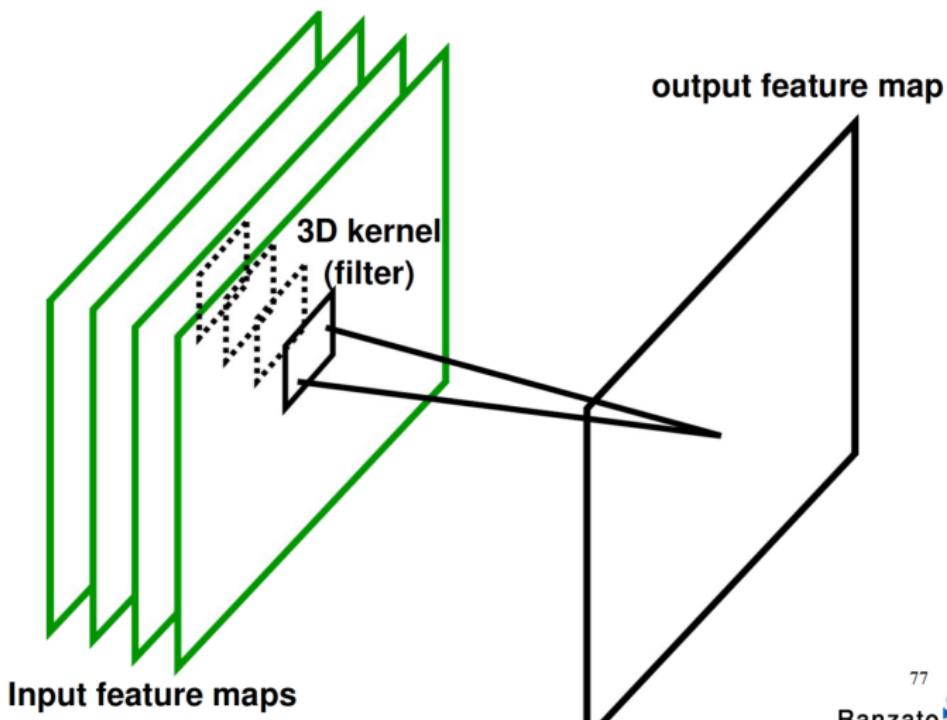
Ranzato 

Volumes and Depths



- Equivalent to applying different filters to the input, and then stacking the result of each of those filters

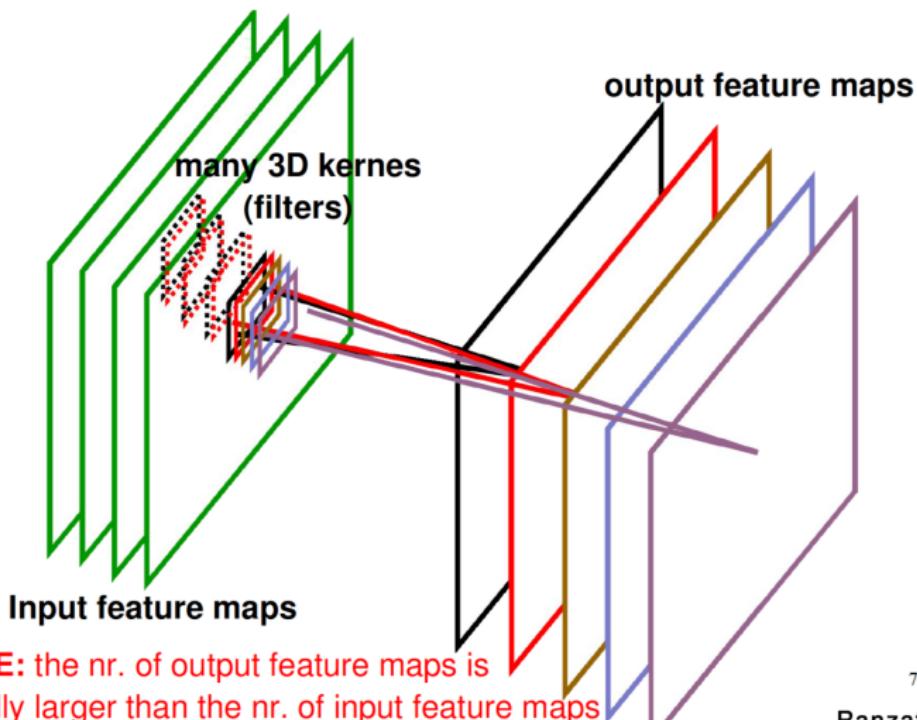
Convolutional Layer



77

Ranzato

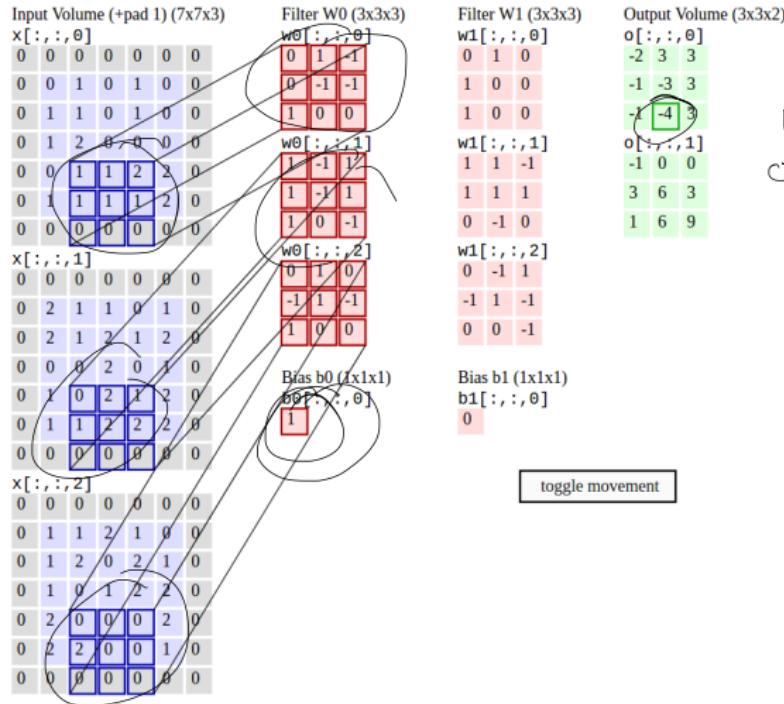
Convolutional Layer



78

Ranzato

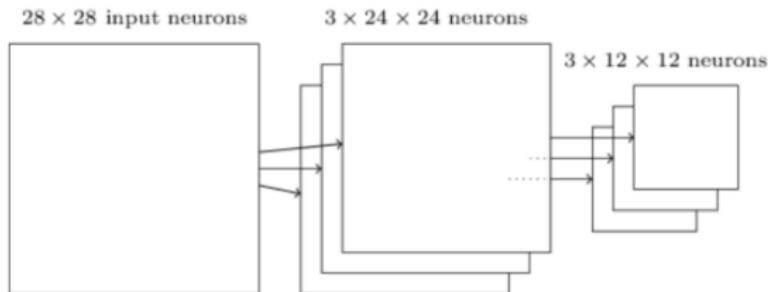
Computing the next layer³²



³²<http://cs231n.github.io/convolutional-networks/>

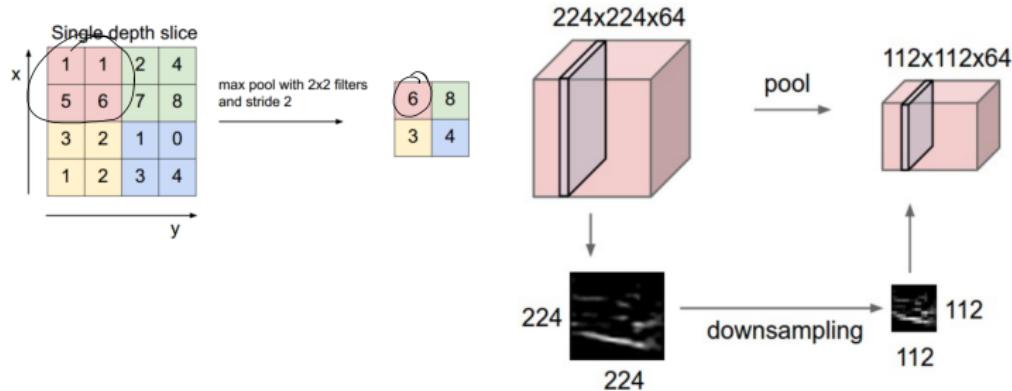
Pooling Layers

- 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



Max Pooling

- Max-pooling: a pooling unit simply outputs the max activation in the input region



Max-Pooling Benefits

- Max-pooling is a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information.
 - Pooling and stride length are both ways to perform downsampling:
max pooling : max (feature exist/how strong)
convol : avg / weights
- Once a feature has been found, its exact location isn't as important as its rough location relative to other features.
- A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers.
 - Convolutional layers can also significantly reduce the number of output features

Final Layer Typically Fully Connected

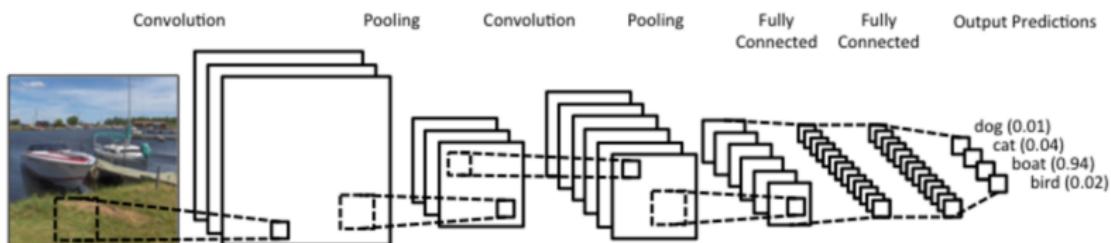


Image: <http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png>

Table of Contents

1 Convolutional Neural Nets (CNNs)

2 Deep Q Learning

Generalization

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



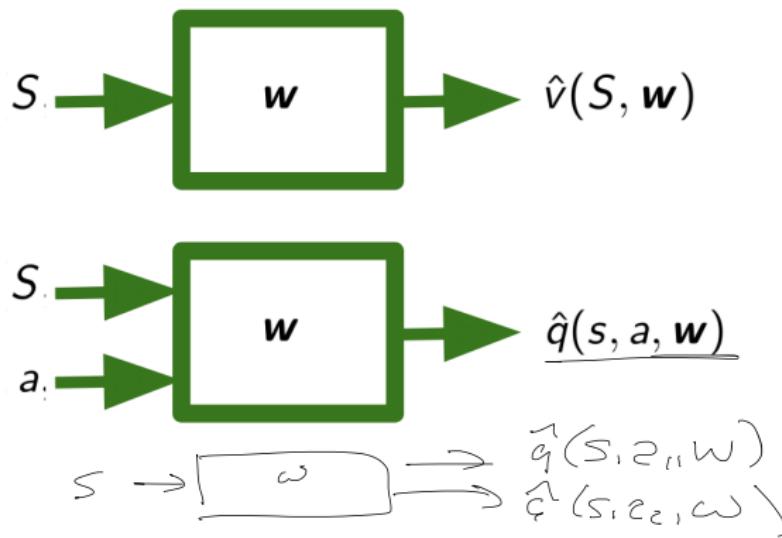
Deep Reinforcement Learning

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

Deep Q-Networks (DQNs)

- Represent value function by Q-network with weights w

$$\hat{v}(s, \omega) \approx v(s, \omega) \quad (1)$$



Recall: Action-Value Function Approximation with an Oracle

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

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

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2} \nabla_w J(w) = \mathbb{E}[(q^\pi(s, a) - \hat{q}^\pi(s, a, w)) \nabla_w \hat{q}^\pi(s, a, w)] \quad (3)$$

$$\Delta(w) = -\frac{1}{2}\alpha \nabla_w J(w) \quad (4)$$

- 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

episodic

$$\Delta w = \alpha(G_t - \hat{q}(s_t, a_t, w)) \nabla_w \hat{q}(s_t, a_t, w) \quad (5)$$

- For SARSA instead use a TD target $r + \gamma \hat{q}(s', a', w)$ which leverages the current function approximation value

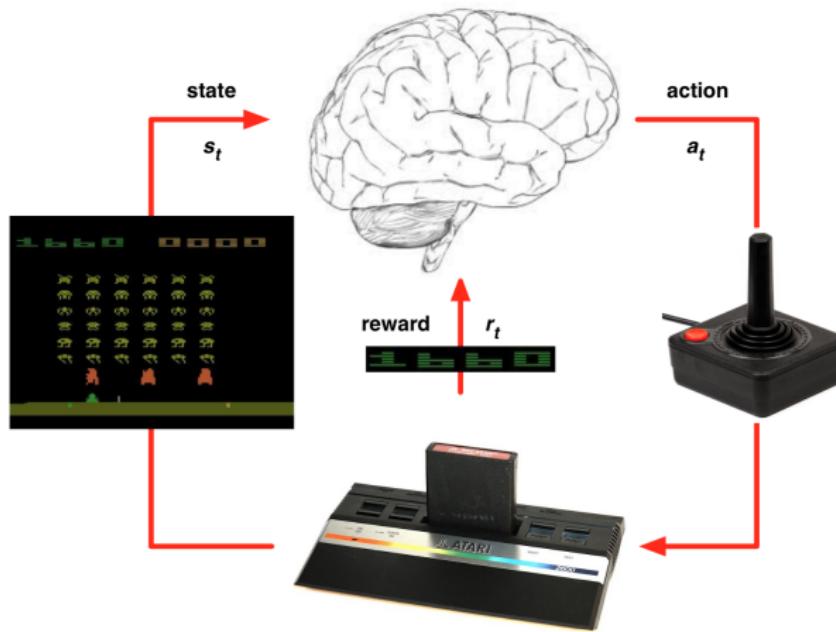
$$\Delta w = \alpha(r + \gamma \hat{q}(s', a', w) - \hat{q}(s, a, w)) \nabla_w \hat{q}(s, a, w) \quad (6)$$

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

$$\Delta w = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', w) - \hat{q}(s, a, w)) \nabla_w \hat{q}(s, a, w) \quad (7)$$

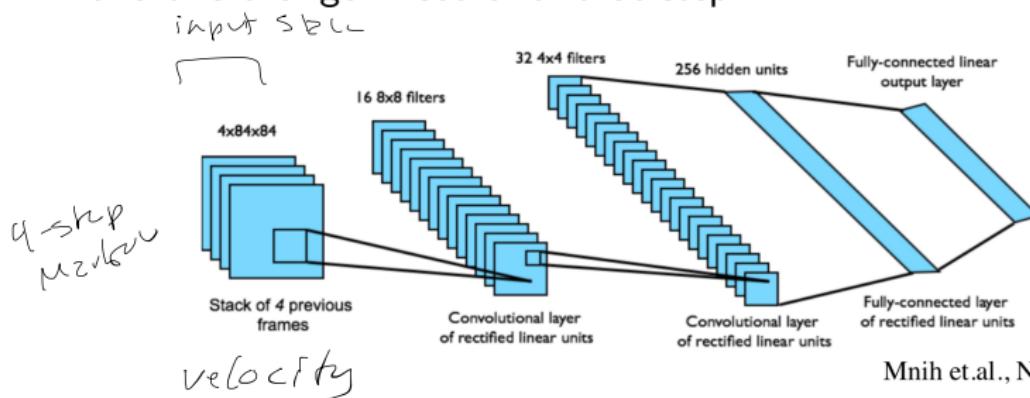
learning rate *a'* *target* *current estimate* *grd*

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

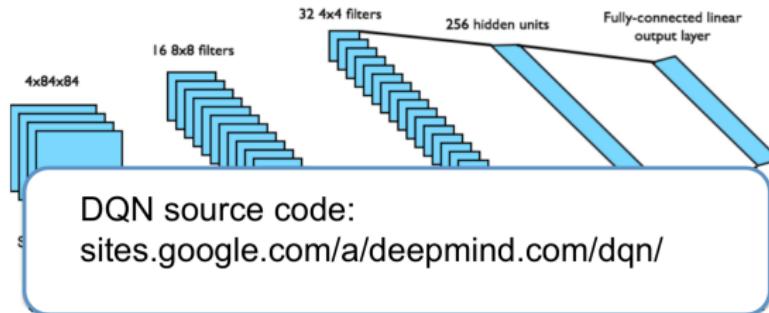


Mnih et.al., Nature, 2014

- 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



Mnih et.al., Nature, 2014

- Network architecture and hyperparameters fixed across all games

Q-Learning with Value Function Approximation

- Minimize MSE loss by stochastic gradient descent
- Converges to optimal q using table lookup representation
- But Q-learning with VFA can diverge
- Two of the issues causing problems:
 - Correlations between samples
 - Non-stationary targets *(target func of VFA)*
- Deep Q-learning (DQN) addresses both of these challenges by
 - Experience replay
 - Fixed Q-targets

DQNs: Experience Replay

$(s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1})$

- To help remove correlations, store dataset (called a **replay buffer**) \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
...
$s_t, a_t, r_{t+1}, s_{t+1}$

$\rightarrow s, a, r, s'$

s, a, \hat{Q}

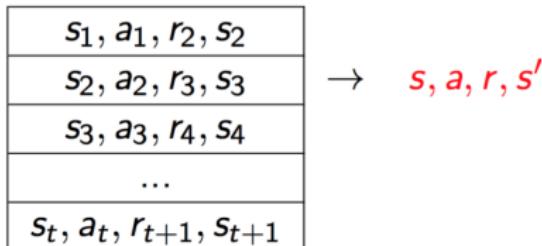
- To perform experience replay, repeat the following:

- $(s, a, r, s') \sim \mathcal{D}$: 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

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) \quad (8)$$

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

$$\Delta w = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', w) - \hat{q}(s, a, w)) \nabla_w \hat{q}(s, a, w) \quad (9)$$

- 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 network** weights used in the target calculation for multiple updates
- Use a different set of weights to compute target than is 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

$$\Delta \mathbf{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}^-) - \hat{q}(s, a, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}) \quad (10)$$

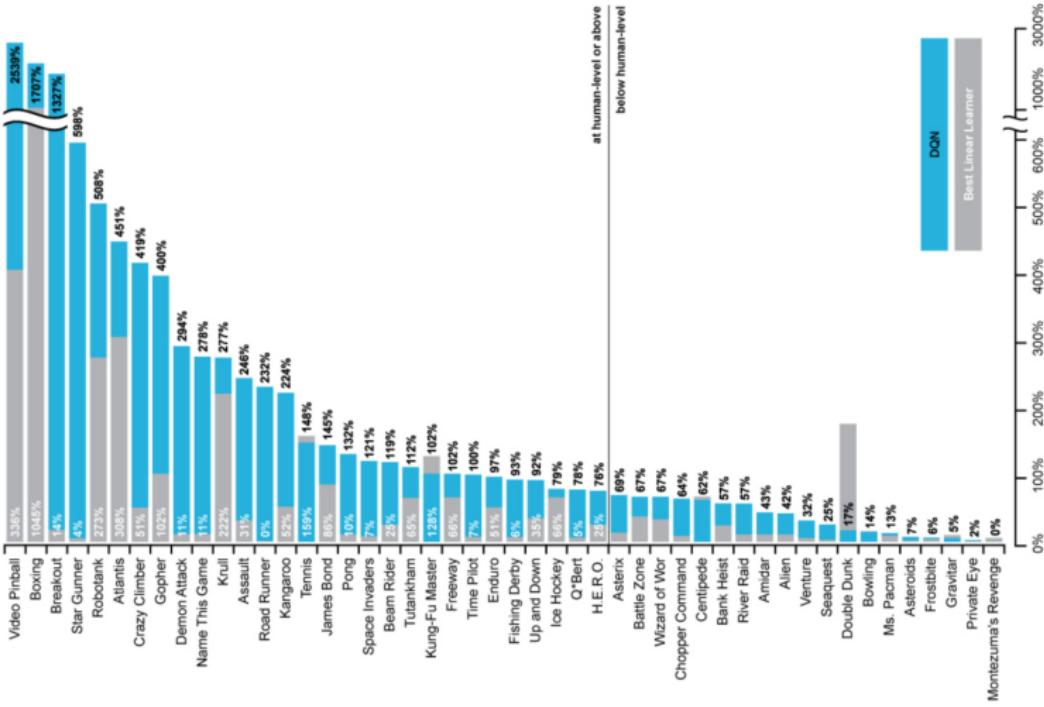
hyperparameter when to update \mathbf{w}^-

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}
*how large
how big
buffer*
- Sample random mini-batch of transitions (s, a, r, s') from \mathcal{D}
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
*when
update
 w^-*
- Optimizes MSE between Q-network and Q-learning targets
- Uses stochastic gradient descent

Demo

DQN Results in Atari



Which Aspects of DQN were Important for Success?

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 item Why? Beyond helping with correlation between samples, what does replaying do?

s_t, a_t, r_t, s_{t+1}
update on

$\rightarrow A \xrightarrow{B}$

- Success in Atari has lead to huge excitement in using deep neural networks to do value function approximation in RL
- Some immediate improvements (many others!)
 - Double DQN
 - Dueling DQN (best paper 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

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

Q-learning off policy

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_t + \underbrace{Q_1(s_{t+1}, \arg \max_{a'} Q_2(s_{t+1}, a')) - Q_1(s_t, a_t))}_{\text{use other } Q \text{ to choose best } a}) \quad (11)$$

- 6: **else**

use other Q
to choose best Q

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

- 7: **end if**
- 8: $t = t + 1$
- 9: **end loop**

Double DQN

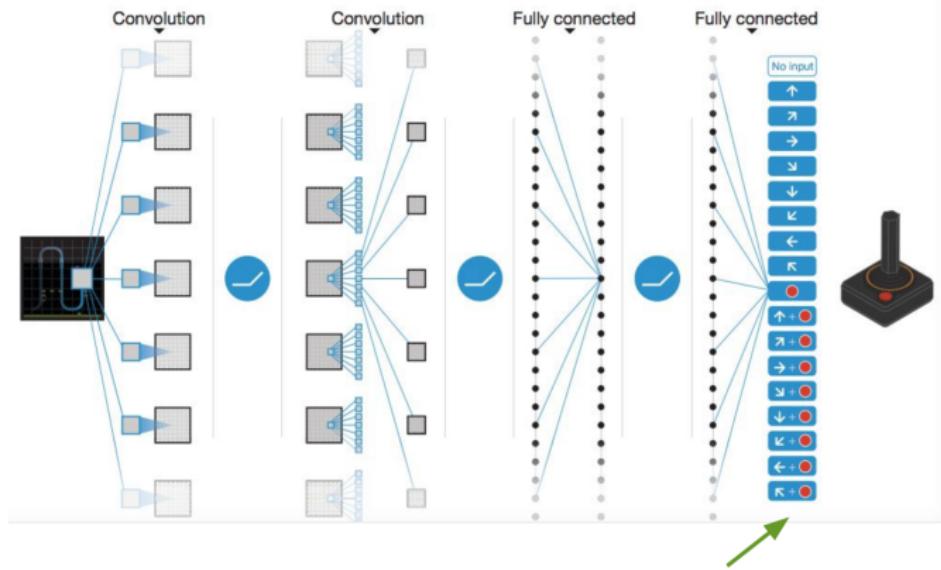
- Extend this idea to DQN
- Current Q-network \mathbf{w} is used to select actions
- Older Q-network \mathbf{w}^- is used to evaluate actions

$$\Delta \mathbf{w} = \alpha(r + \gamma \underbrace{\hat{q}(\arg \max_{a'} \hat{q}(s', a', \mathbf{w}), \mathbf{w}^-)}_{\text{Action selection: } \mathbf{w}}) - \hat{q}(s, a, \mathbf{w})) \quad (12)$$

Action evaluation: \mathbf{w}^-

Action selection: \mathbf{w}

Double DQN



1 network, outputs Q value for each action

Double DQN

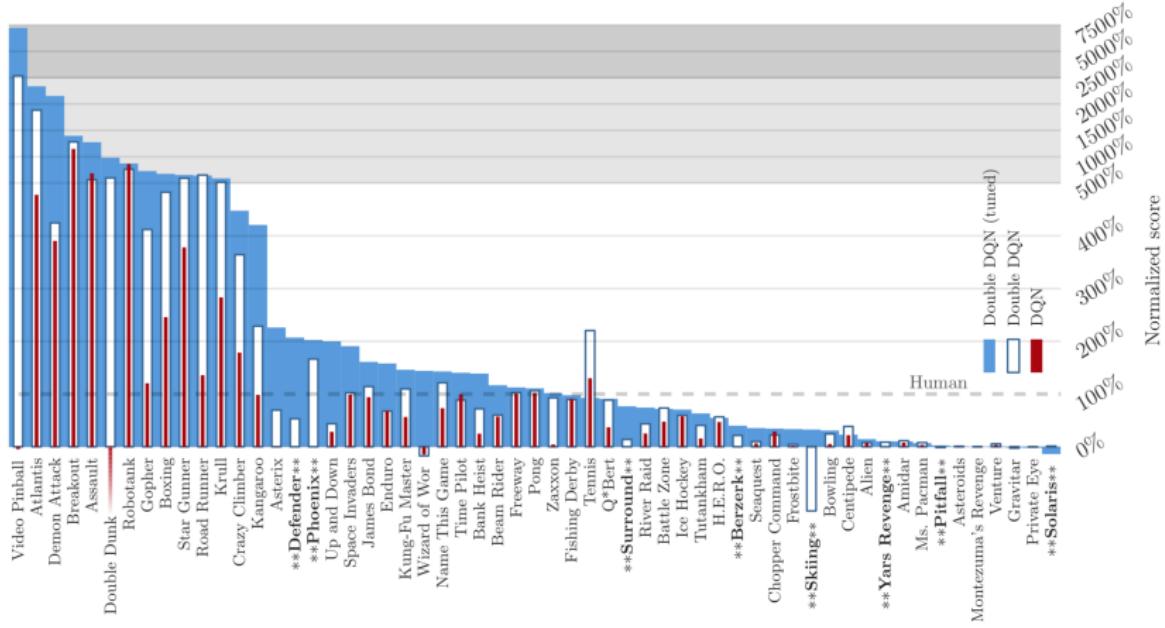


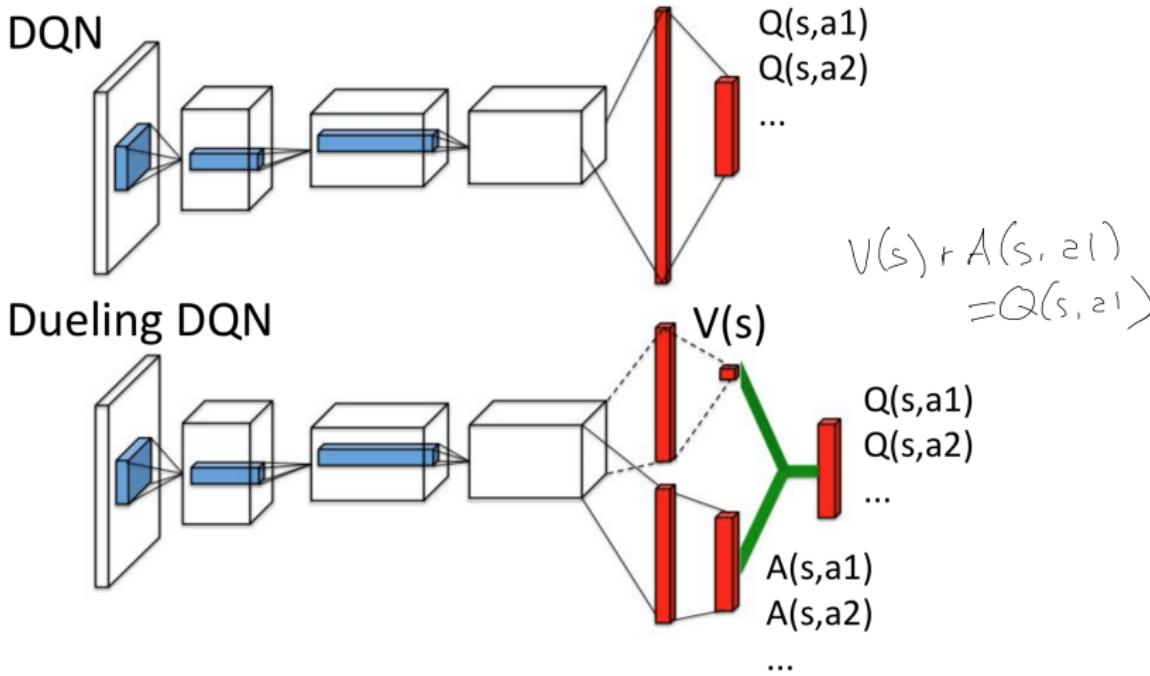
Figure: van Hasselt, Guez, Silver, 2015

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)

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$
$$A^*(s, a) = Q^*(s, a) - V^*(s)$$
$$\underset{a^*}{A^*(s, a)} = \underset{a^*}{\textcircled{O}}$$
$$a \neq a^* \quad " \quad \leftarrow \textcircled{O}$$

Dueling DQN



Wang et.al., ICML, 2016

Identifiability

- Advantage function

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

- Identifiable?

$$Q = A + V$$

give you Q , is there a unique
 $A \neq V$?

$$+/- \rightarrow A \neq V$$

Identifiability

- Advantage function

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

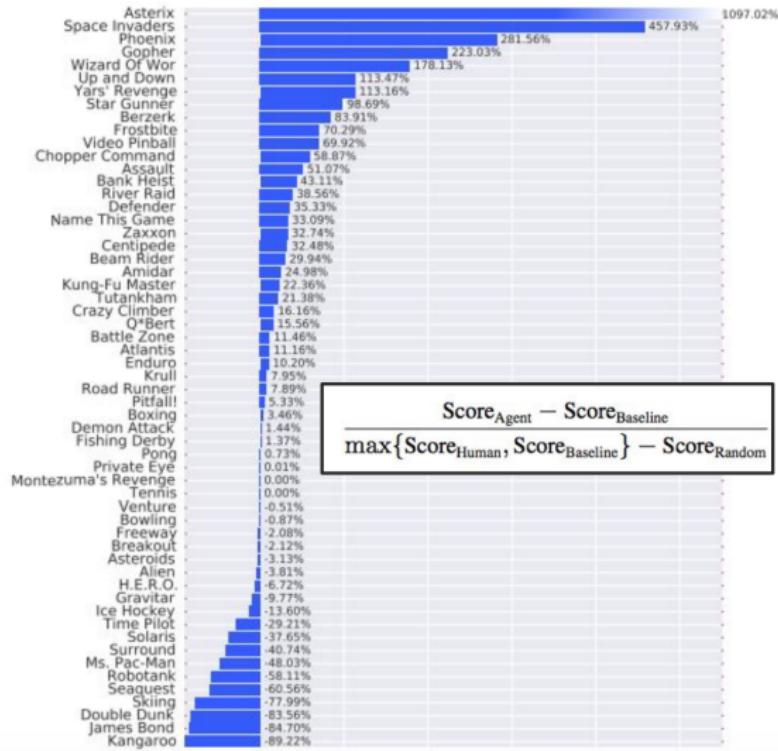
- Unidentifiable
- Option 1: Force $A(s, a) = 0$ if a is action taken

$$\hat{q}(s, a; \mathbf{w}) = \hat{v}(s; \mathbf{w}) + \left(A(s, a; \mathbf{w}) - \max_{a' \in \mathcal{A}} A(s, a'; \mathbf{w}) \right)$$

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

$$\hat{q}(s, a; \mathbf{w}) = \hat{v}(s; \mathbf{w}) + \left(A(s, a; \mathbf{w}) - \frac{1}{|\mathcal{A}|} \underbrace{\sum_{a'} A(s, a'; \mathbf{w})}_{\text{mean}} \right)$$

V.S. DDQN with Prioritized Replay



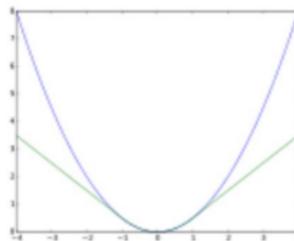
Practical Tips for DQN on Atari (from J. Schulman)

- 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

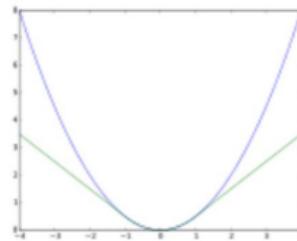
$$L(x) = \begin{cases} \frac{x^2}{2} & \text{if } |x| \leq \delta \\ \delta|x| - \frac{\delta^2}{2} & \text{otherwise} \end{cases}$$



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 3-line 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

- 1 Convolutional Neural Nets (CNNs)
- 2 Deep Q Learning

Class Structure

- Last time: Value function approximation and deep learning
- This time: Convolutional neural networks and deep RL
- Next time: Imitation learning