Reinforcement Learning: CNNs and Deep Q Learning

Emma Brunskill
Stanford University
Spring 2017

With many slides for DQN from David Silver and Ruslan Salakhutdinov and some vision slides from Gianni Di Caro

Today

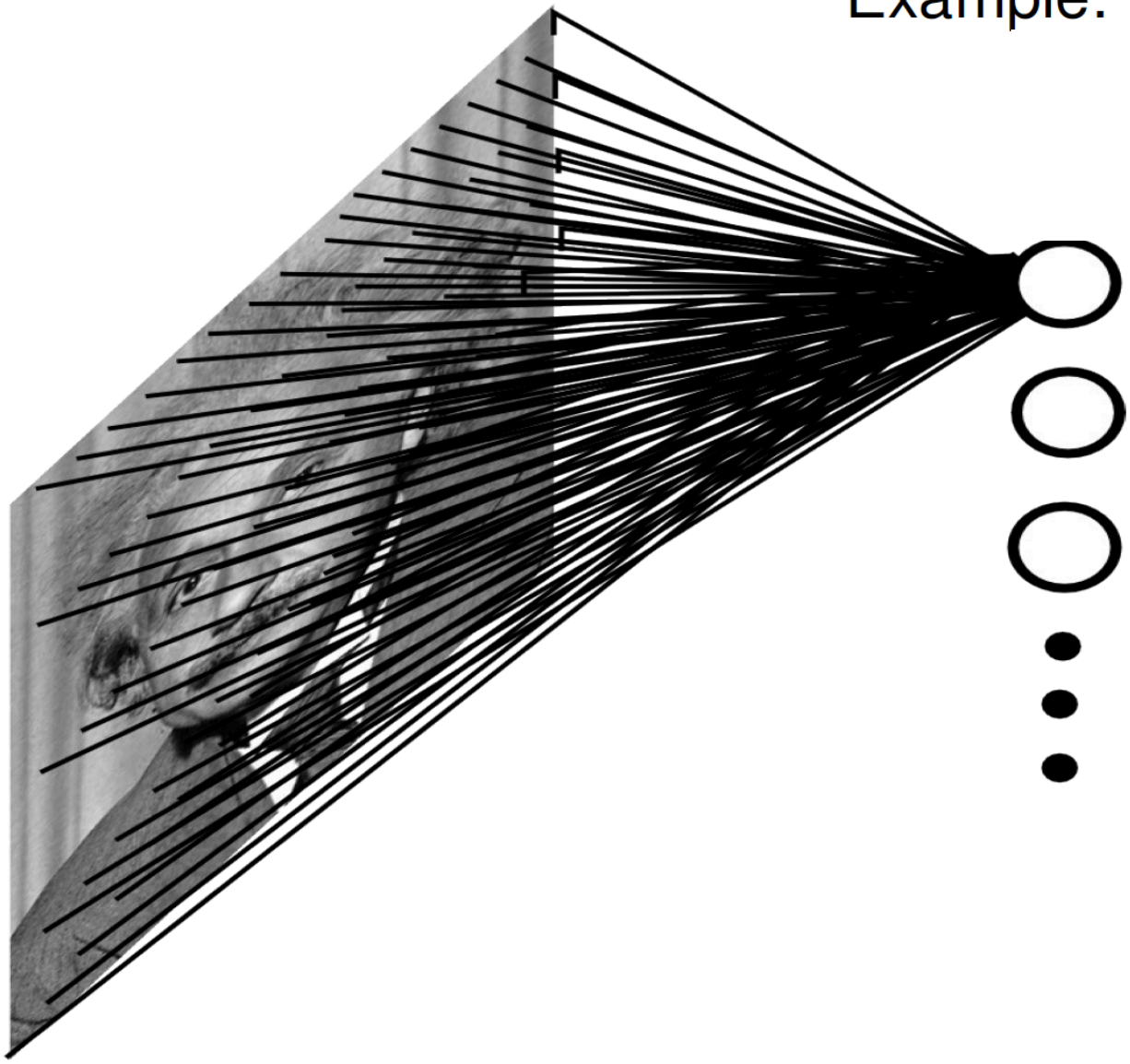
- Convolutional Neural Nets (CNNs)
- Deep Q learning

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



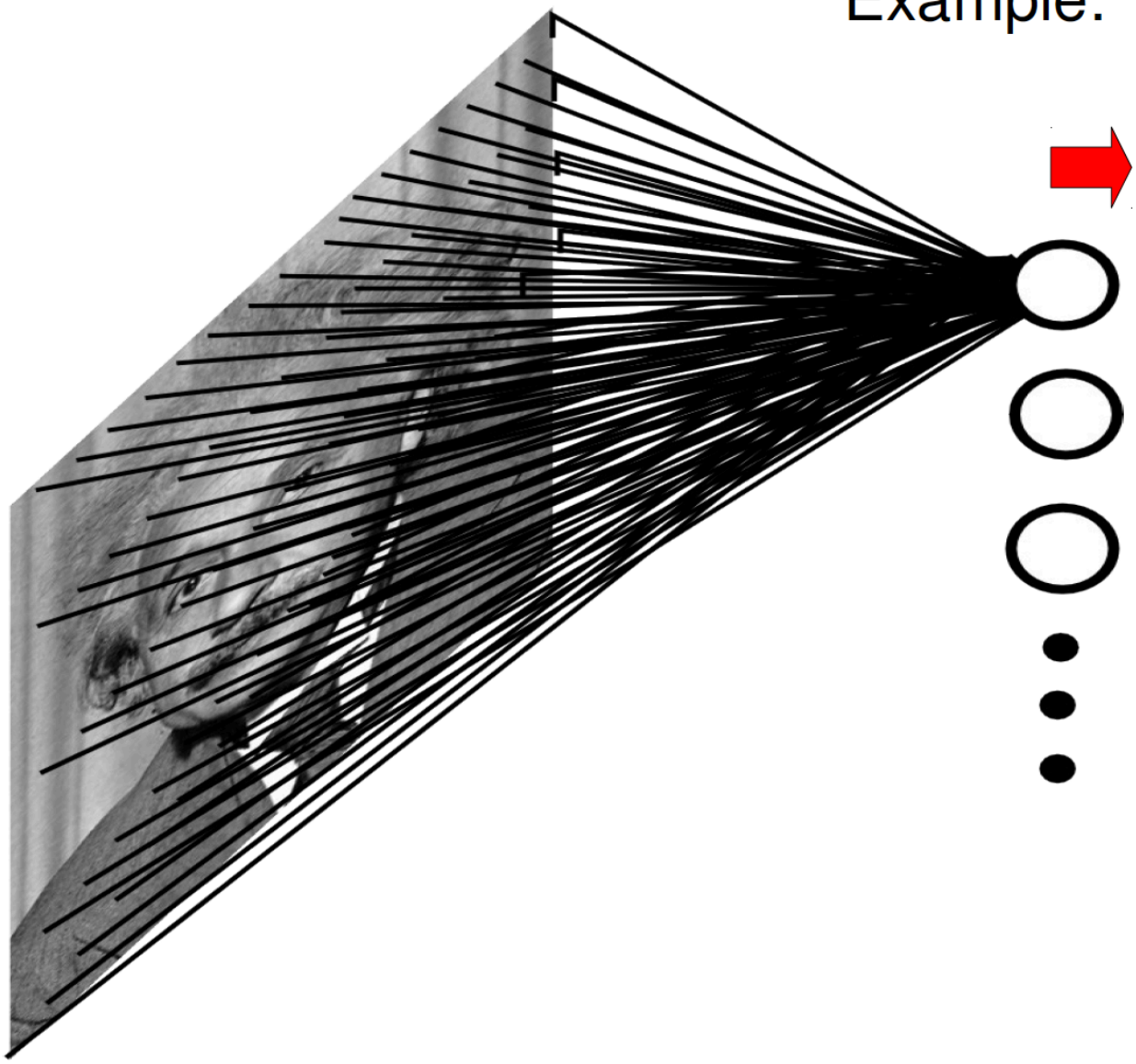
How many weight parameters for a single node which is a linear combination of input?

FULLY CONNECTED NEURAL NET

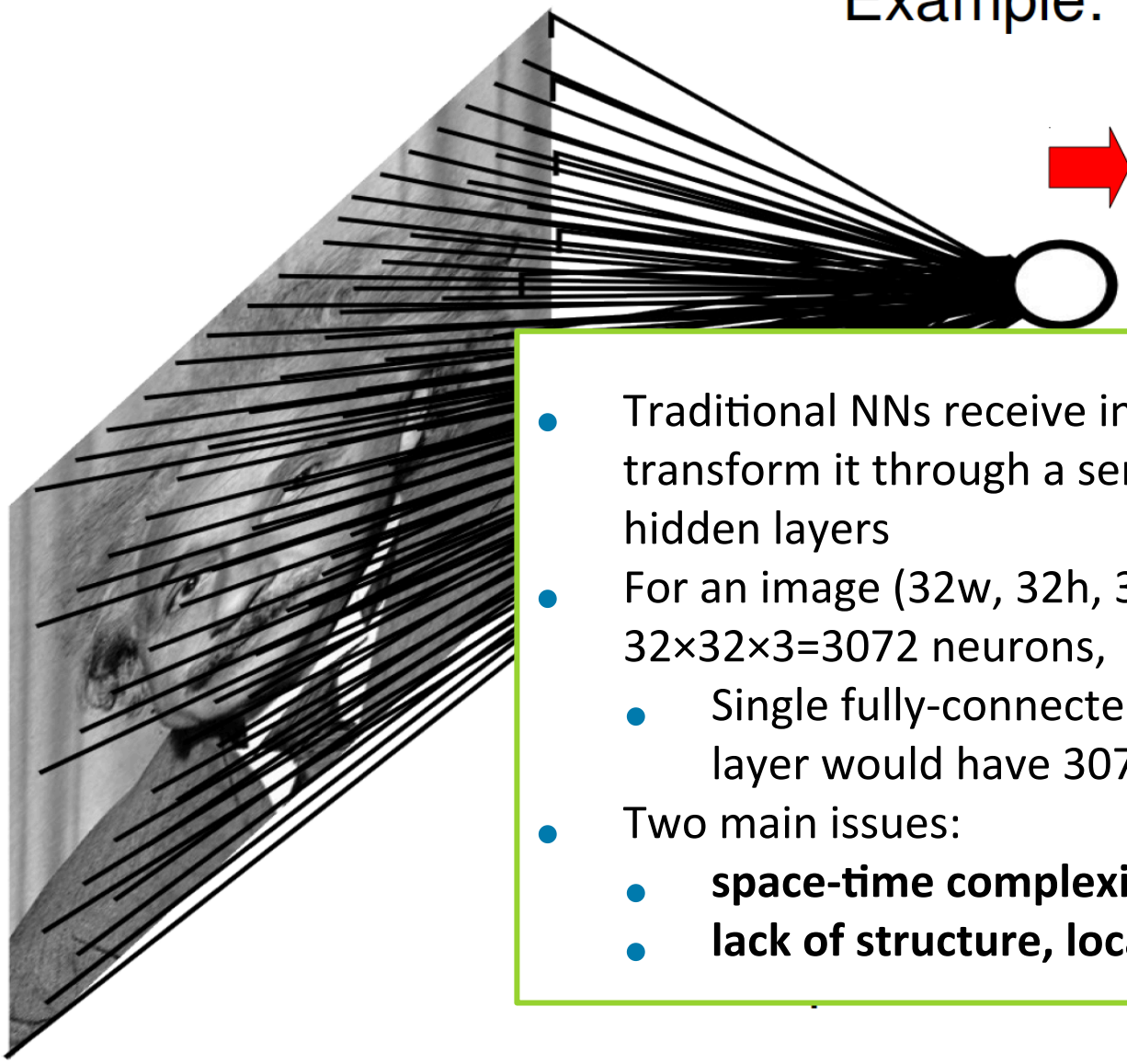
Example: 1000x1000 image

1M hidden units

10^12 parameters!!!



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**

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 wrt to known entities of interest
- Look for “interest points”: image regions that are unusual
- Coming up with these is nontrivial

Convolutional NN

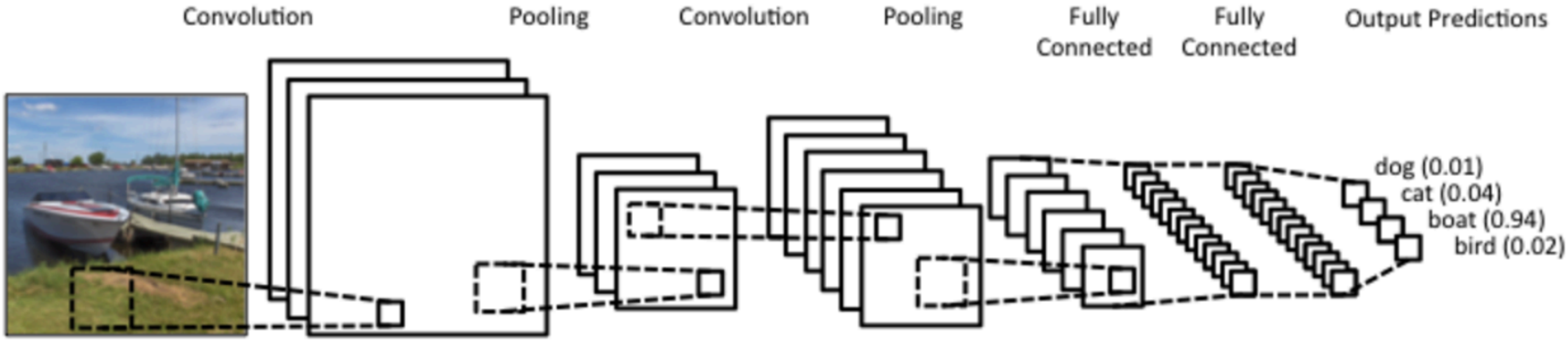
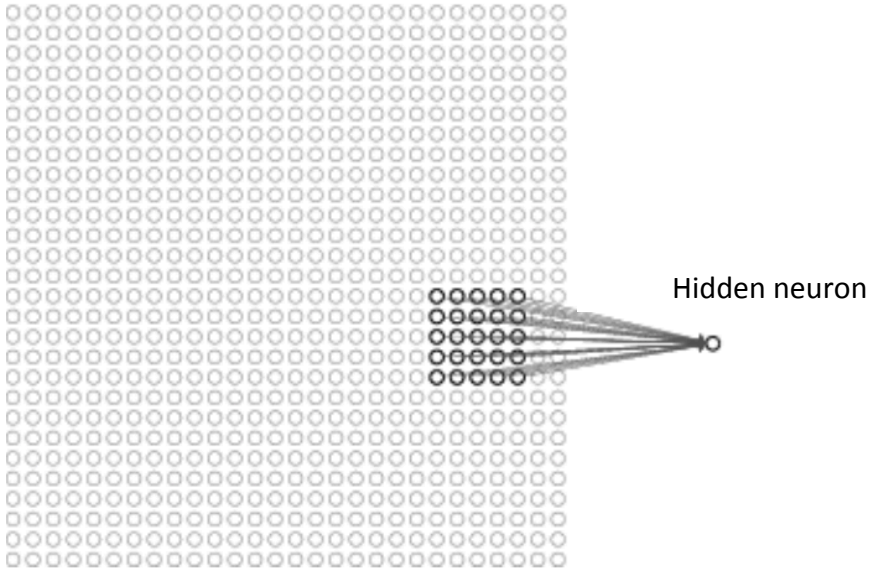
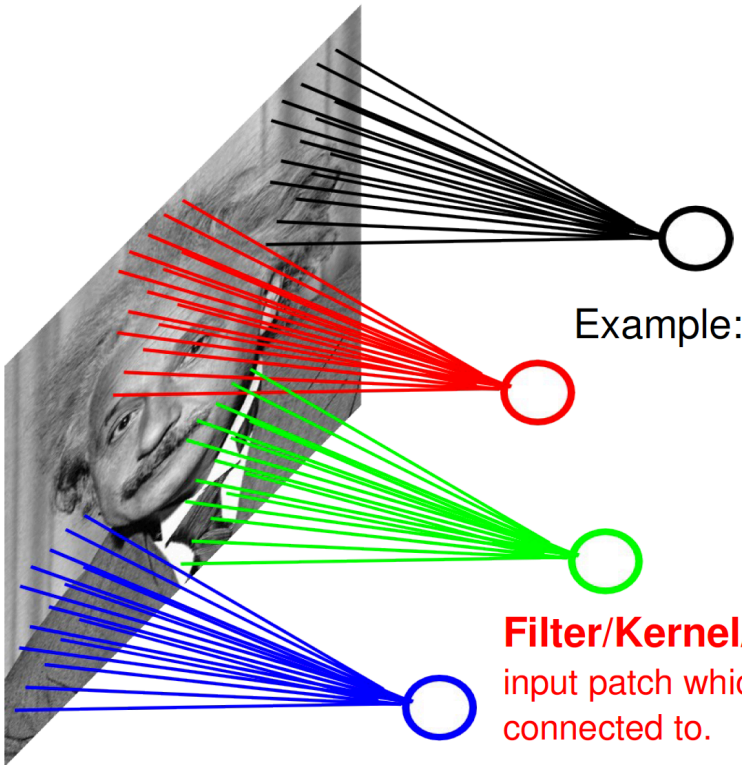


Image: http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png

- 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





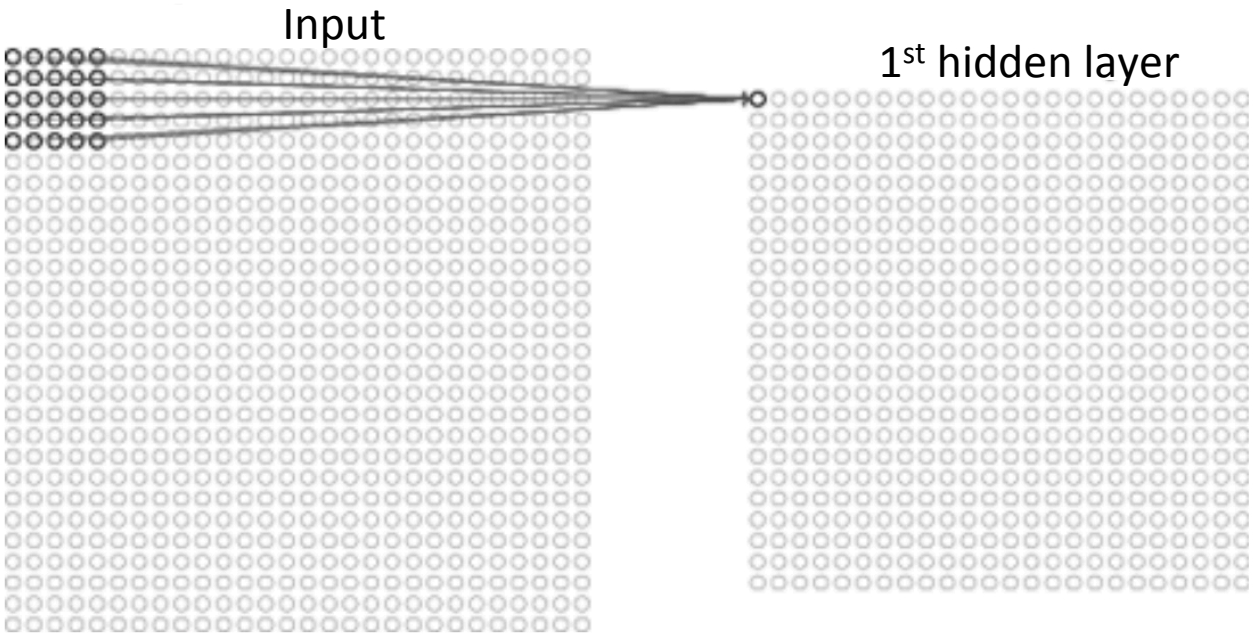
Example: 1000x1000 image
1M hidden units
Filter size: 10x10
100M parameters

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

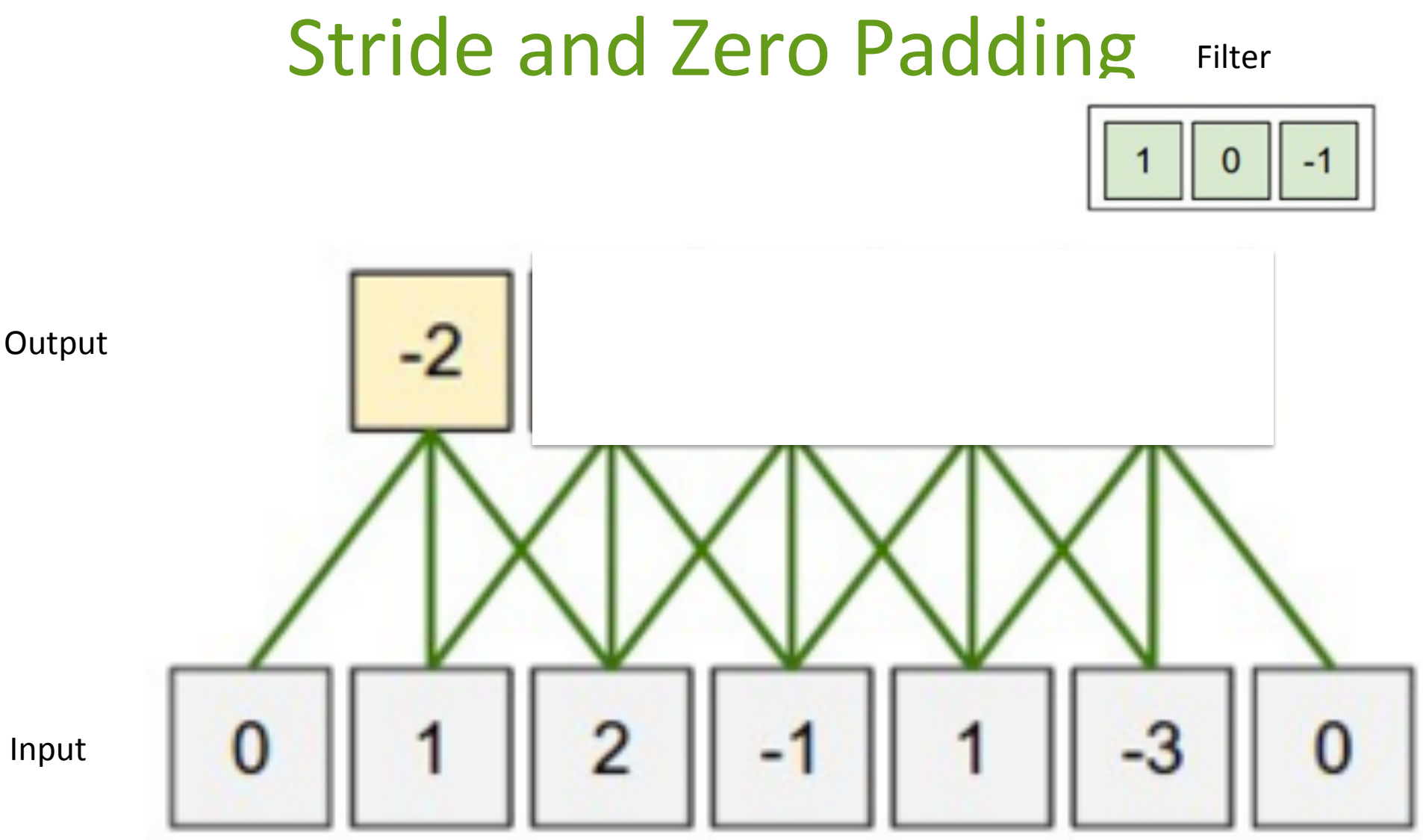
Ranzato

(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?

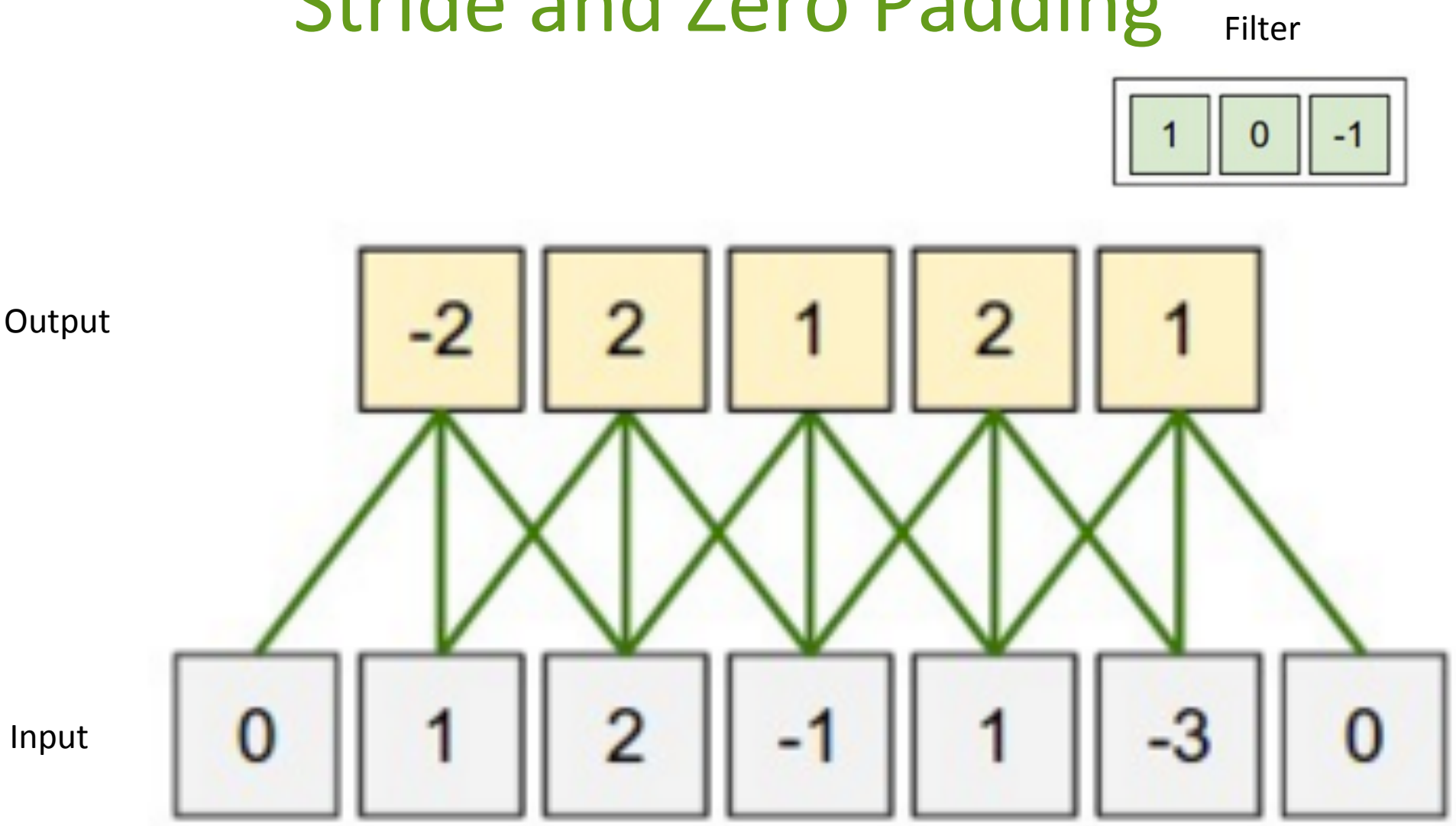


Stride and Zero Padding



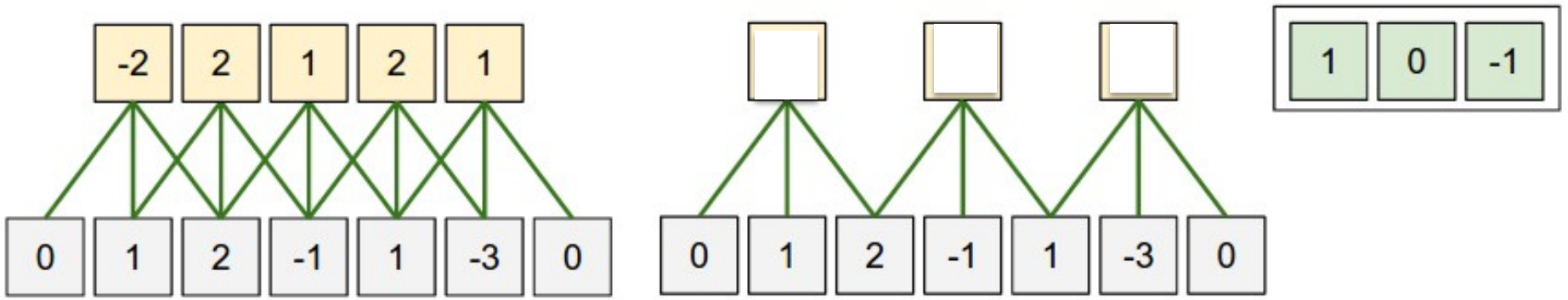
Stride: how far (spatially) move over filter
Zero padding: how many 0s to add to either side of input layer

Stride and Zero Padding

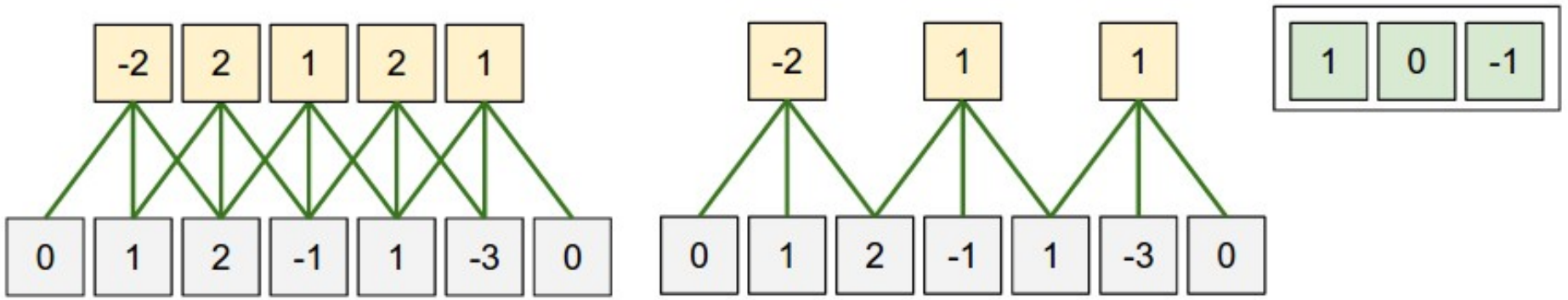


Stride: how far (spatially) move over filter
Zero padding: how many 0s to add to either side of input layer

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?

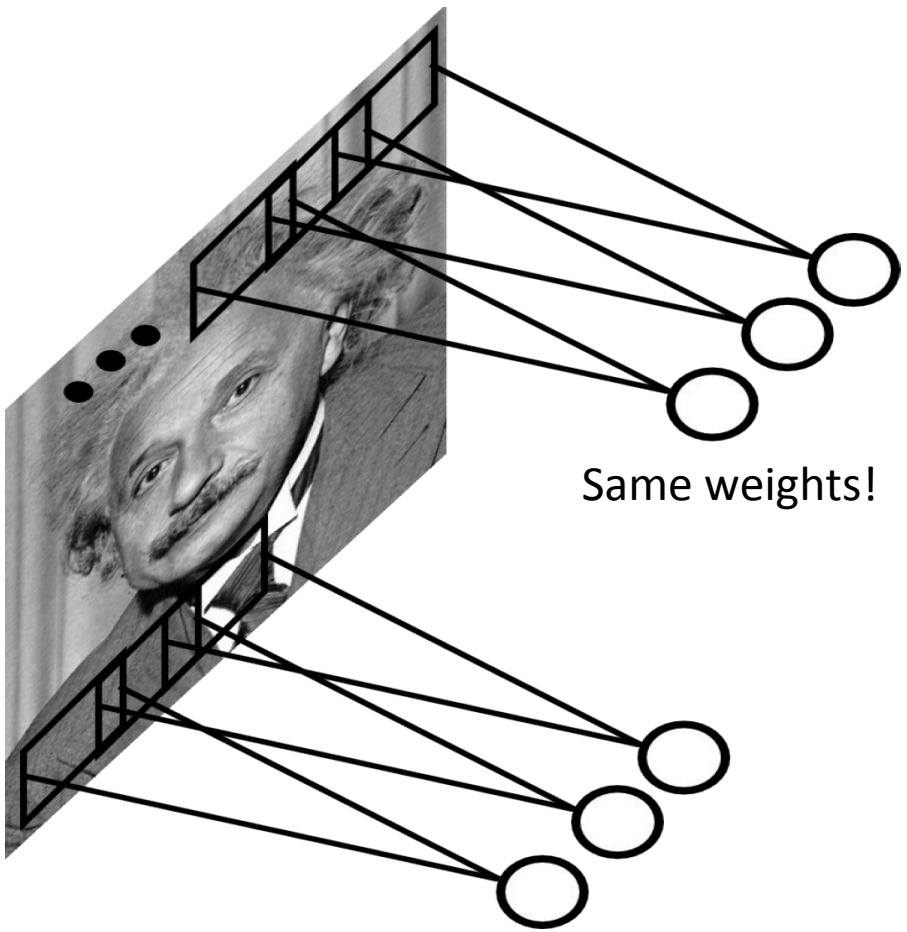
$$g(b + \sum_i w_i x_i)$$

- Sum over **i** is only over the (in this case, 25) neurons in the receptive field of the hidden layer neuron
- The *same* weights *w* and bias *b* are used for each of the (here, 24×24) hidden neurons

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 determine the neuron to "fire" or, more in general, 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.
  - And so it is useful to apply the same feature detector everywhere in 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 convolutional filter (defined by weights and bias), also known as convolution with learned kernels

Convolutional Image Filters

1	1	1
1	1	1
1	1	1

Unweighted 3x3 smoothing kernel

0	1	0
1	4	1
0	1	0

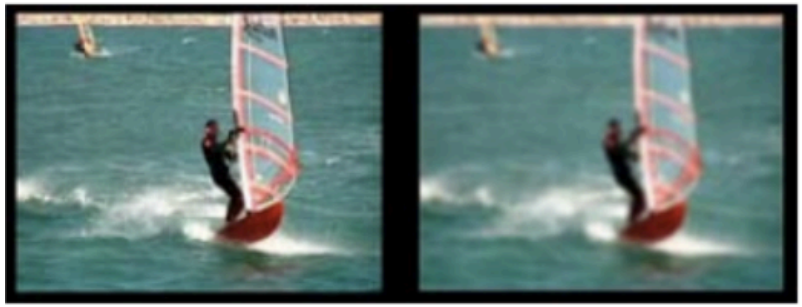
Weighted 3x3 smoothing kernel with Gaussian blur

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

Kernel to make image sharper

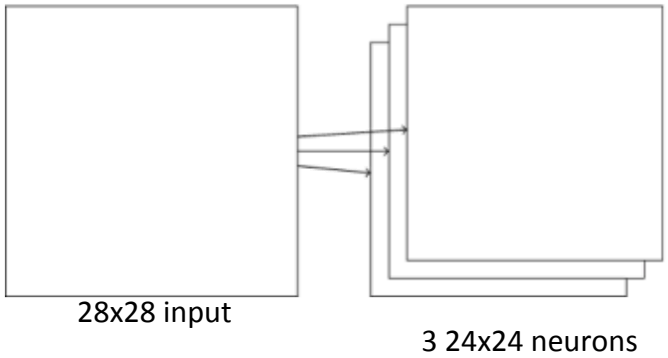
-1	-1	-1
-1	9	-1
-1	-1	-1

Intensified sharper image



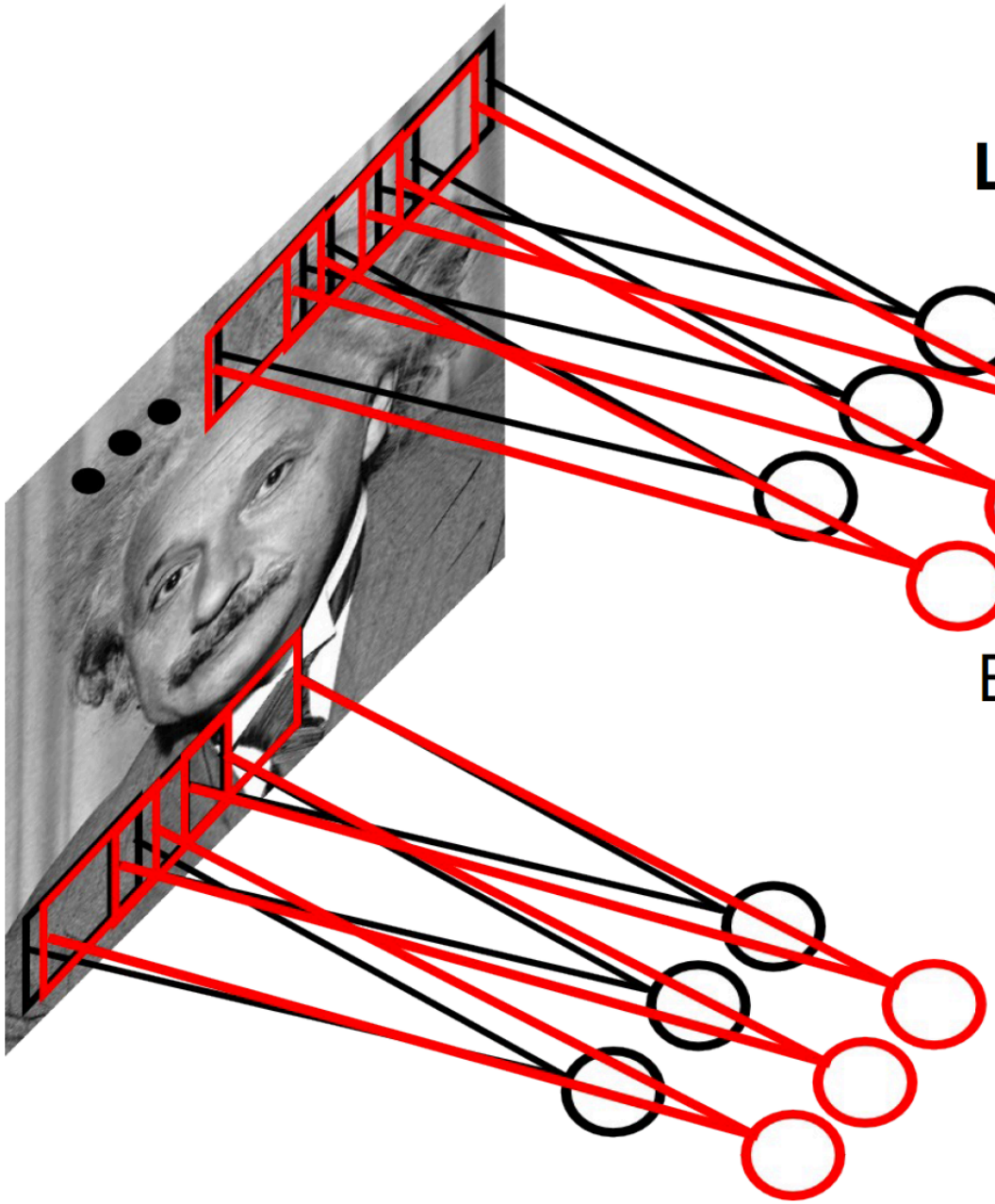
Why Only 1 Filter?

- 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 and one bias
- The result is that the network can detect 3 different kinds of features, with each feature being detectable across the entire image

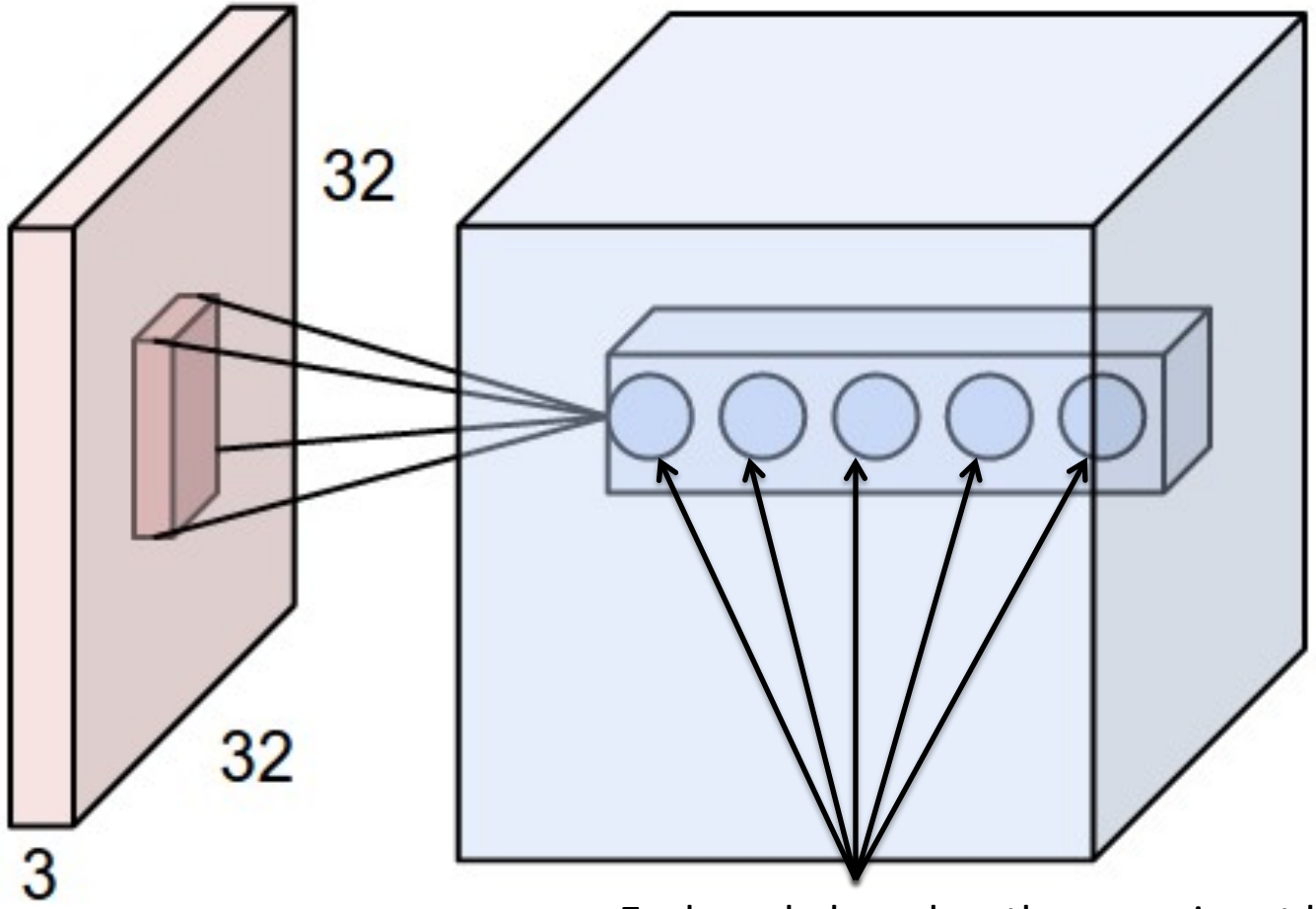
CONVOLUTIONAL NET



Learn multiple filters.

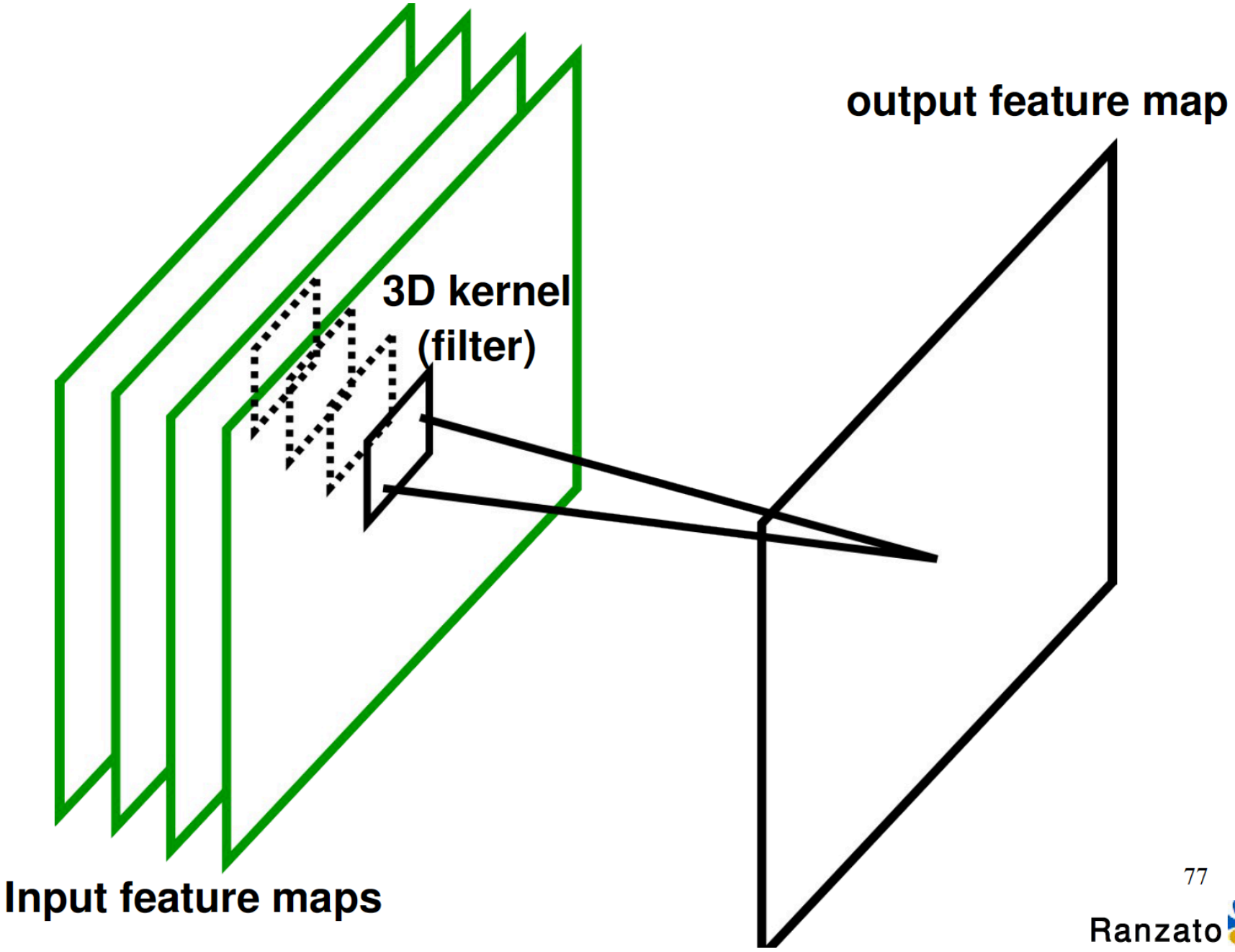
E.g.: 1000x1000 image
100 Filters
Filter size: 10x10
10K parameters

Volumes and Depths



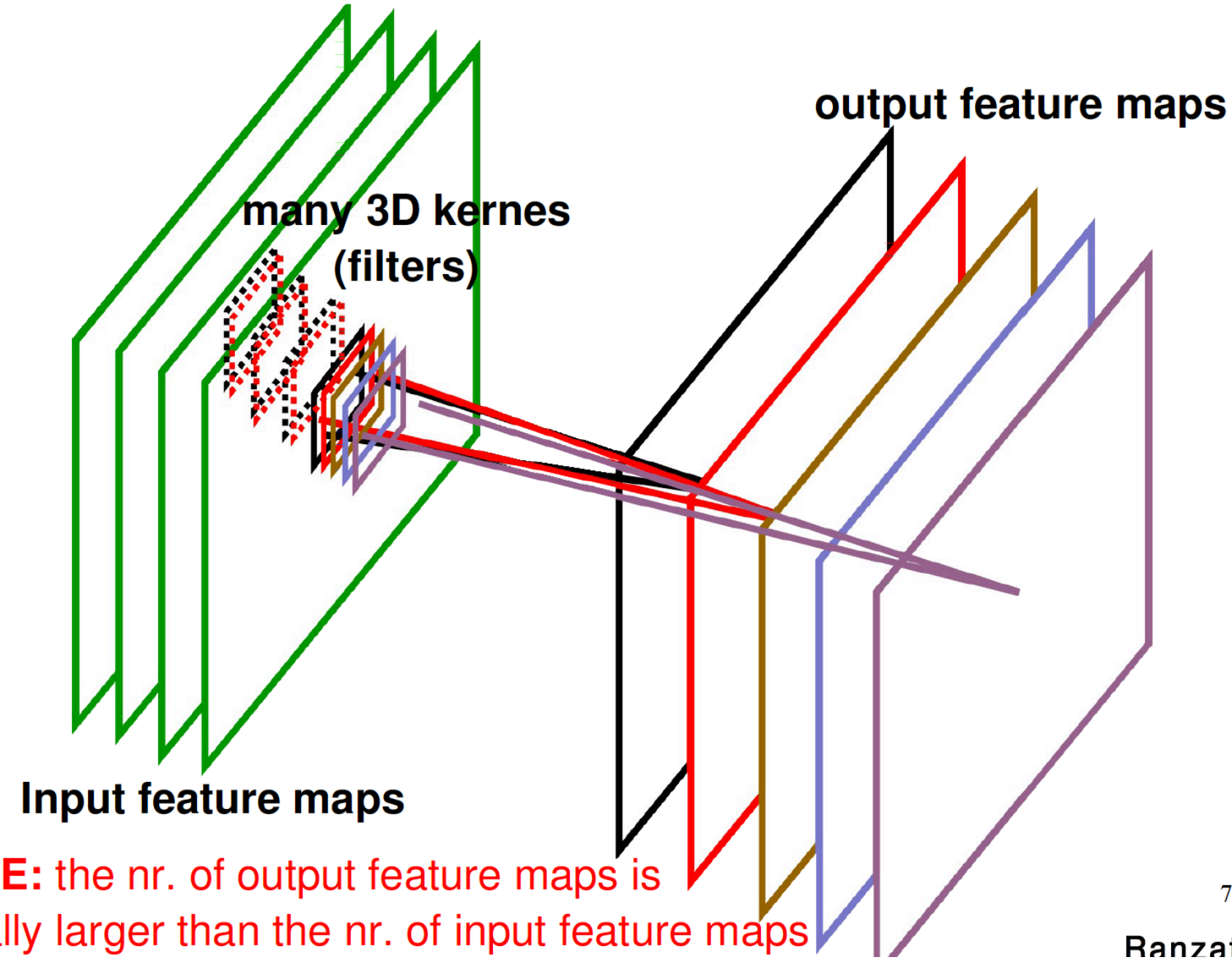
Each node here has the same input but different weight vectors (e.g. computing different features of same input)

CONVOLUTIONAL LAYER

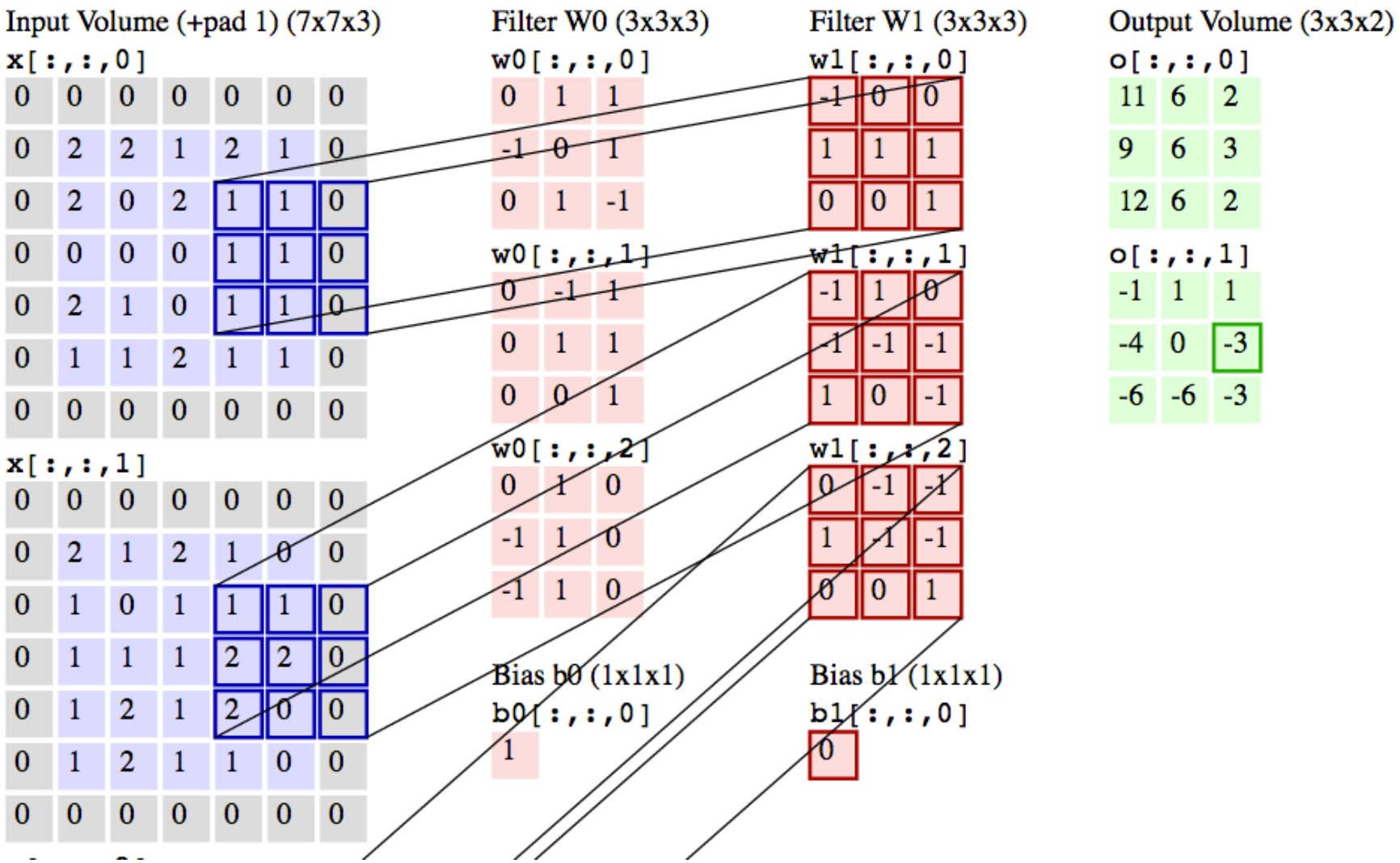


Ranzato

CONVOLUTIONAL LAYER

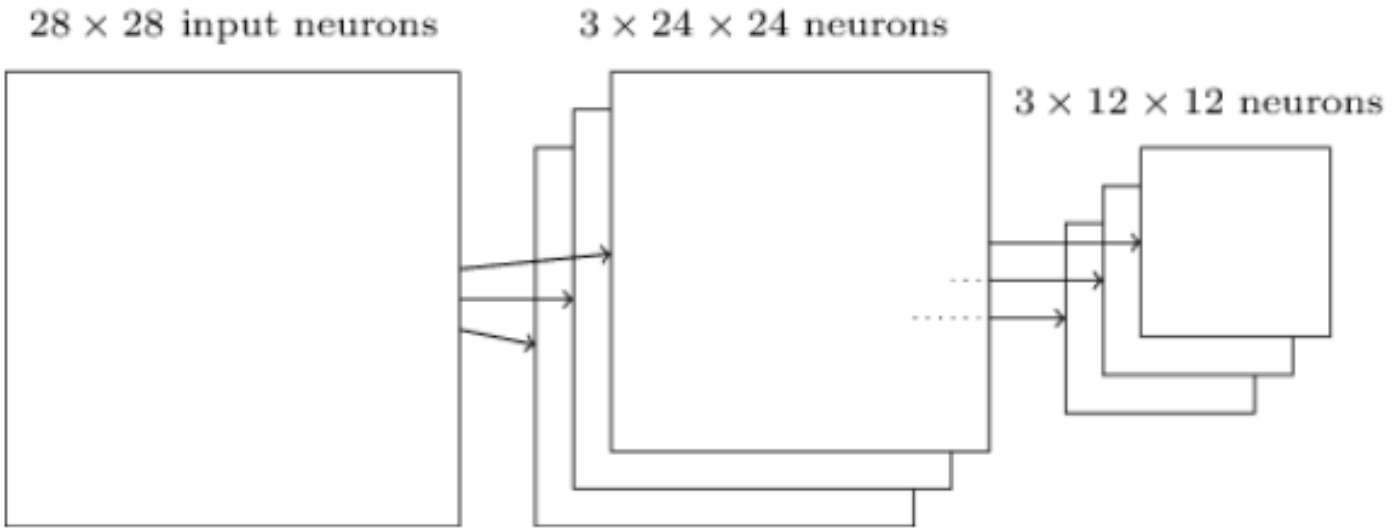


NOTE: the nr. of output feature maps is usually larger than the nr. of input feature maps


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 2 1 2 1 0
0 2 0 2 1 1 0
0 0 0 0 1 1 0
0 2 1 0 1 1 0
0 1 1 2 1 1 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 2 1 2 1 0 0
0 1 0 1 1 1 0
0 1 1 1 2 2 0
0 1 2 1 2 0 0
0 1 2 1 1 0 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
0 1 1
-1 0 1
0 1 -1
w0[:,:,1]
0 -1 1
0 1 1
0 0 1
w0[:,:,2]
0 1 0
-1 1 0
-1 1 0
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 0 0
1 1 1
0 0 1
w1[:,:,1]
-1 1 0
-1 -1 -1
1 0 -1
w1[:,:,2]
0 -1 -1
1 -1 -1
0 0 1
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
11 6 2
9 6 3
12 6 2
o[:,:,1]
-1 1 1
-4 0 -3
-6 -6 -3

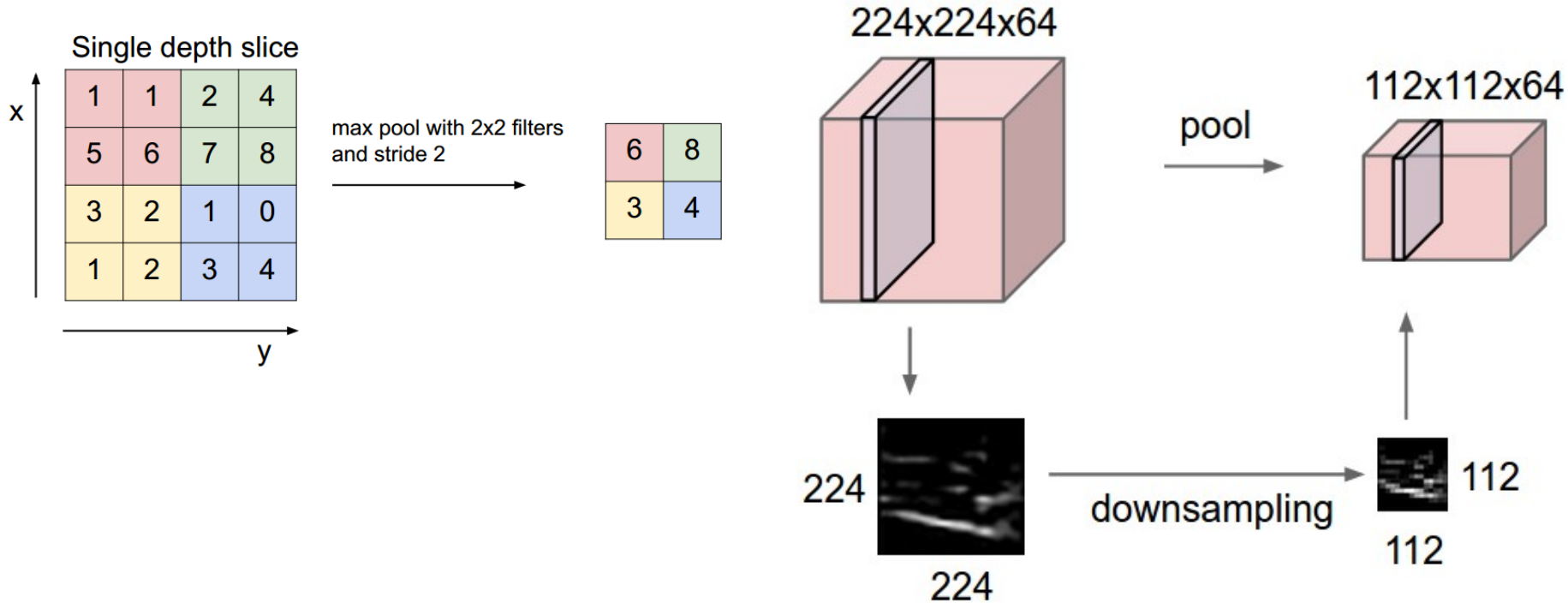
Pooling Layers

- 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

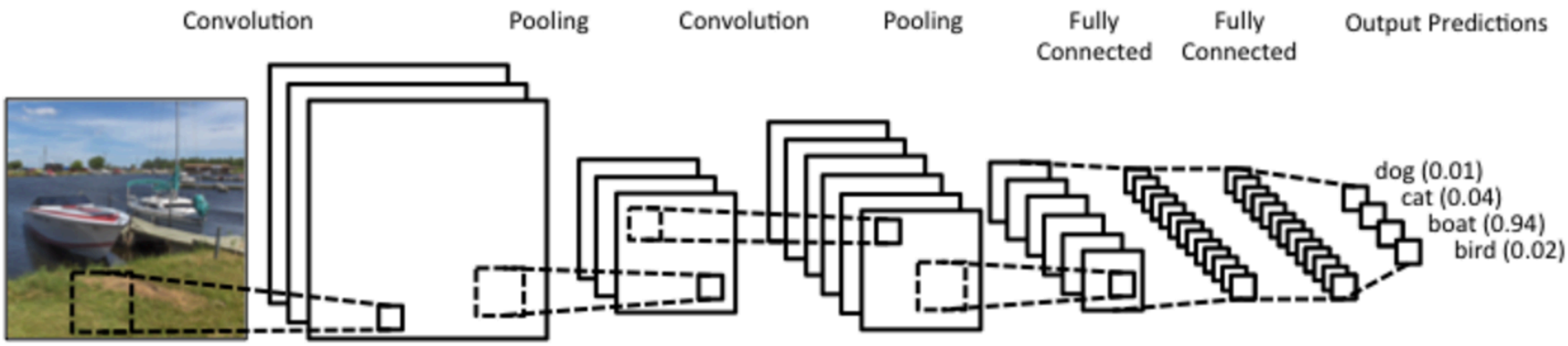
- Max-pooling: a pooling unit simply outputs the max activation in the input region



Max Pooling Benefits

- Max-pooling as 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.
- 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.

Final Layer Typically Fully Connected



Overview

- CNNs
- Deep Q learning

Generalization

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

Deep Reinforcement Learning

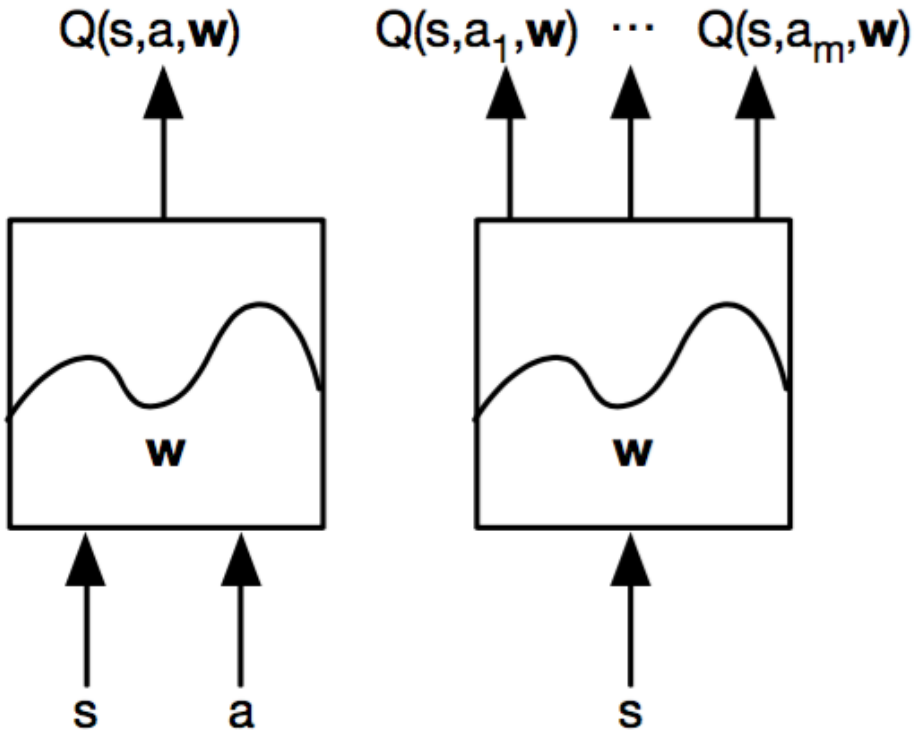
- Use deep neural networks to represent
  - Value function
  - Policy
  - Model

- Optimize loss function by stochastic gradient descent (SGD)

Deep Q-Networks (DQNs)

- Represent value function by Q-network with weights w

$$Q(s, a, \mathbf{w}) \approx Q^*(s, a)$$



Q-Learning

- Optimal Q-values should obey Bellman equation

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q(s',a')^* \mid s,a\right]$$

- Treat right-hand $r + \gamma \max_{a'} Q(s',a',\mathbf{w})$ as a target

- Minimize MSE loss by stochastic gradient descent

$$I = \left(r + \gamma \max_{a} Q(s',a',\mathbf{w}) - Q(s,a,\mathbf{w})\right)^2$$

- Remember VFA lecture: Minimize mean-squared error between the true action-value function $q_\pi(S,A)$ and the approximate Q function:

$$J(\mathbf{w}) = \mathbb{E}_\pi\left[(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))^2\right]$$

Q-Learning

- Minimize MSE loss by stochastic gradient descent

$$I = \left(r + \gamma \max_{a} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}) \right)^2$$

- Converges to Q* using table lookup representation

- But diverges using neural networks due to:
  - Correlations between samples
  - Non-stationary targets

DQNs: Experience Replay

- To remove correlations, build data-set from agent's own 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
...
$s_t, a_t, r_{t+1}, s_{t+1}$

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

- Sample experiences from data-set and apply update

$$I = \left(r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right)^2$$

- To deal with non-stationarity, target parameters w− are held fixed

Remember: Experience Replay

- Given experience consisting of ⟨state, value⟩, or <state, action/value> pairs

$$\mathcal{D} = \{\langle s_1, v_1^\pi \rangle, \langle s_2, v_2^\pi \rangle, \ldots, \langle s_T, v_T^\pi \rangle\}$$

- Repeat
  - Sample state, value from experience

$$\langle s, v^\pi \rangle \sim \mathcal{D}$$

  - Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(v^\pi - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$$

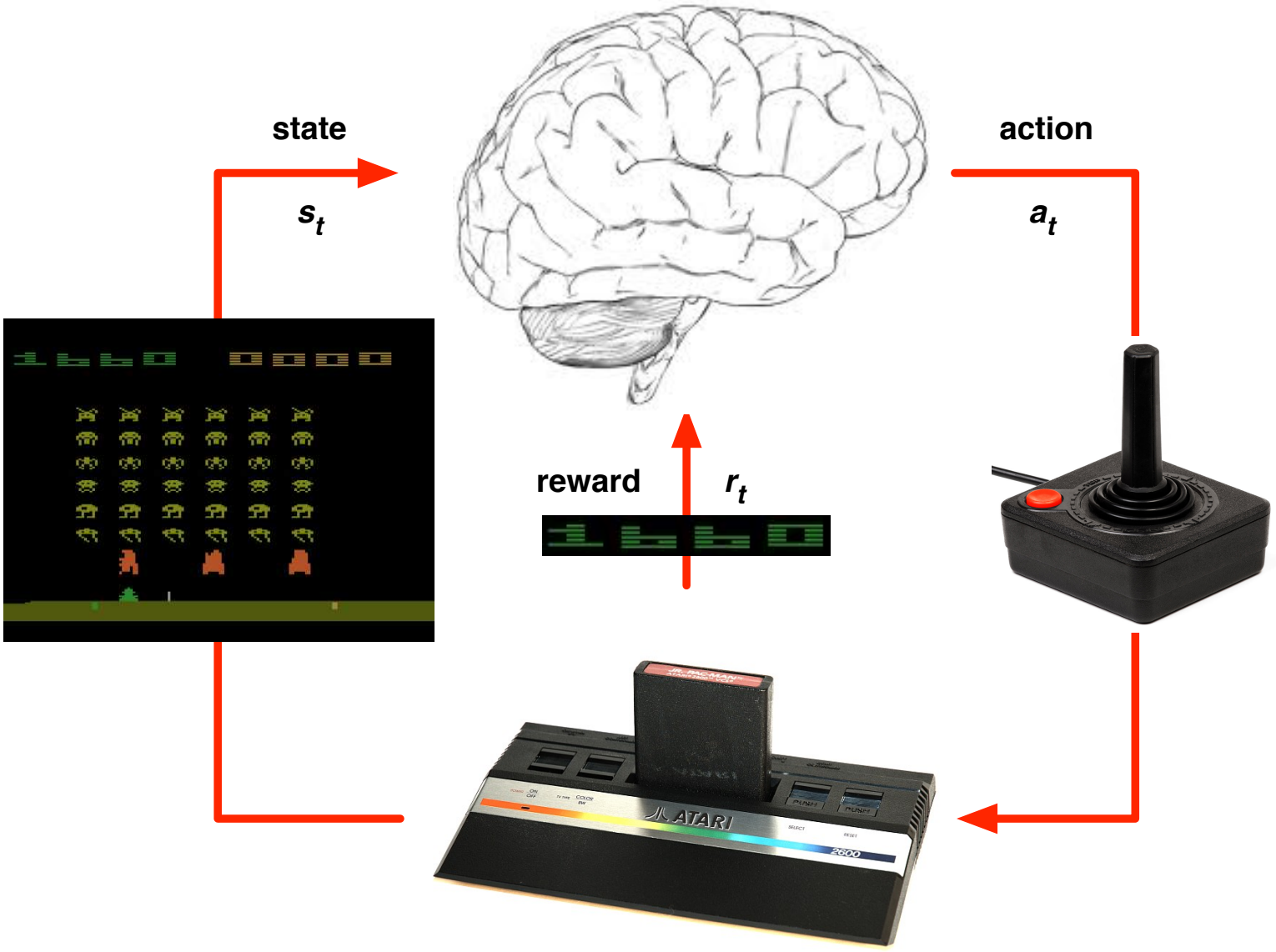
DQNs: Experience Replay

- DQN uses experience replay and fixed Q-targets
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
- Sample random mini-batch of transitions (s,a,r,s′) from D
- Compute Q-learning targets w.r.t. old, fixed parameters w−
- Optimize MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; w_i^-)}_{\text{Q-learning target}} - \underbrace{Q(s, a; w_i)}_{\text{Q-network}}\right)^2\right]$$

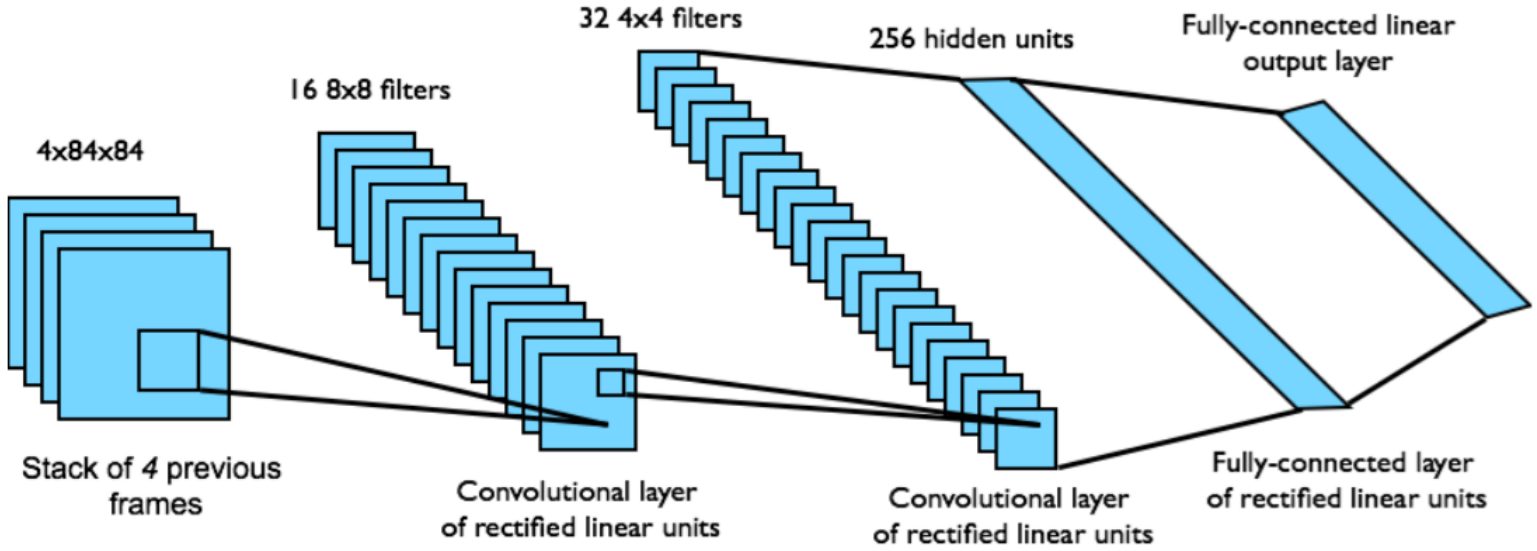
- Use stochastic gradient descent

DQNs 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

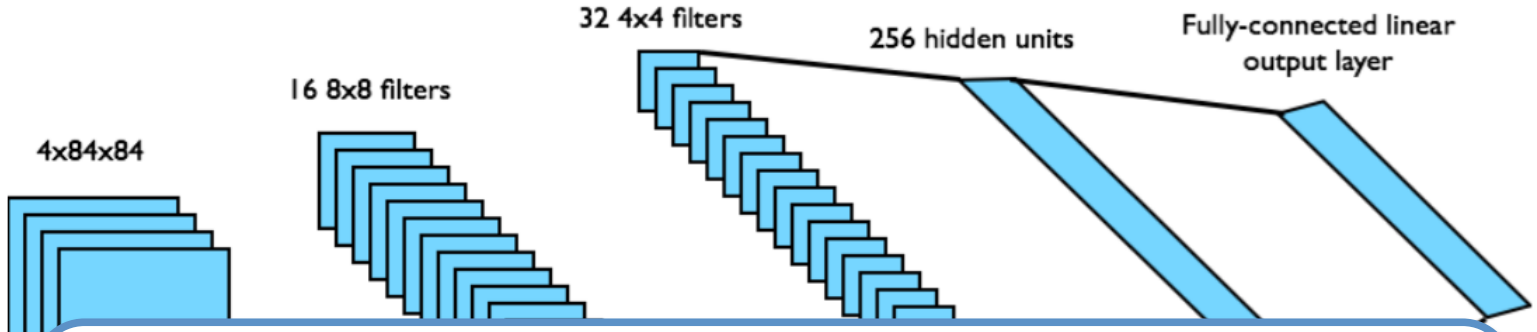


- Network architecture and hyperparameters fixed across all games

Mnih et.al., Nature, 2014

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



DQN source code:
sites.google.com/a/deepmind.com/dqn/

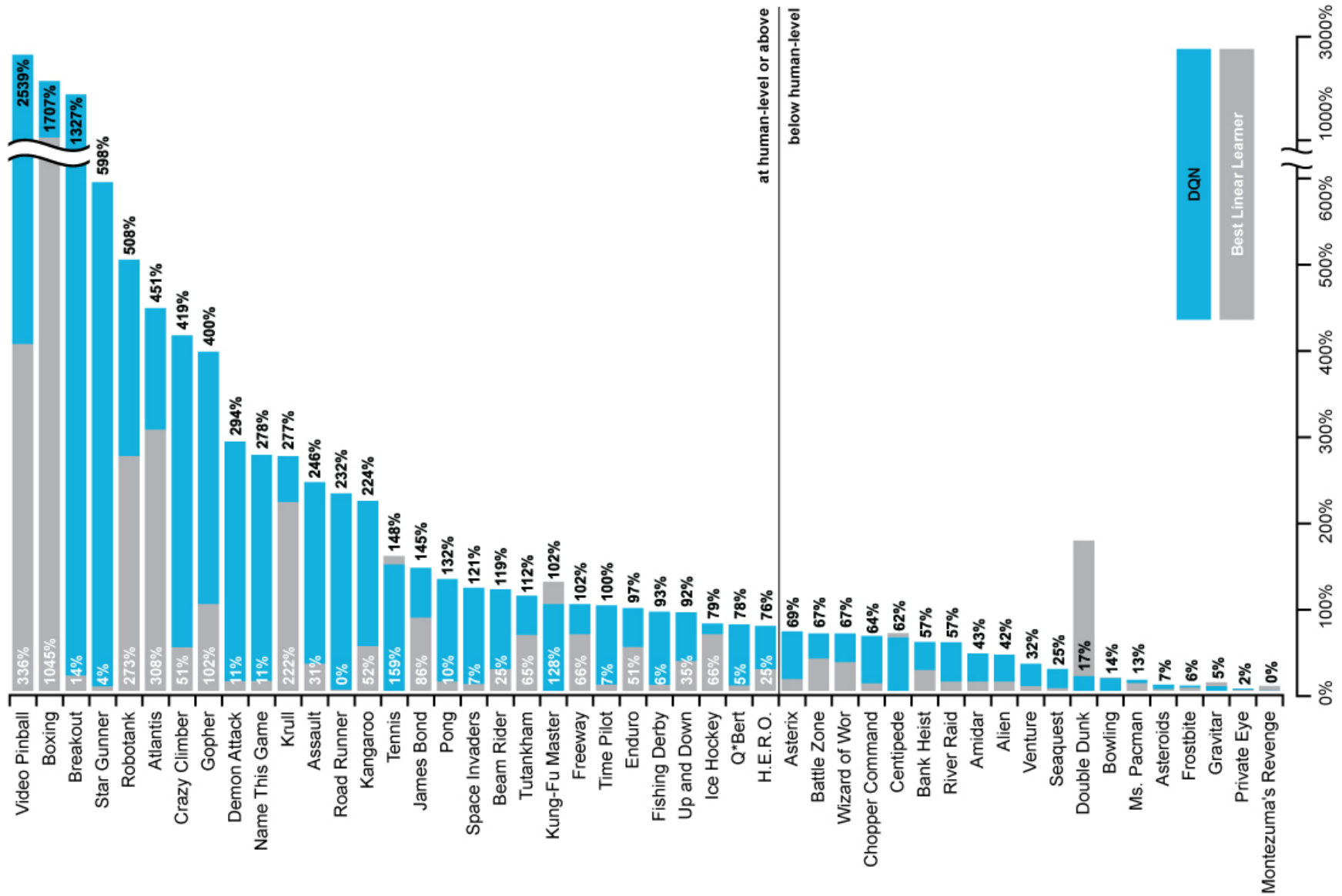
- Network architecture and hyperparameters fixed across all games

Mnih et.al., Nature, 2014

Demo

Mnih et.al., Nature, 2014

DQN Results in Atari



Which Aspects of DQN Were Important for Success?

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

- Replay is **hugely** important

Double Q-Learning

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
  - never on the same time steps (Q_1 and Q_2 are independent)
  - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1, use Q_2 for the value of the next state:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \\ + \alpha\Big(R_{t+1} + Q_2\big(S_{t+1}, \operatorname*{arg\,max}_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big)$$

- Action selections are ε-greedy with respect to the sum of Q_1 and Q_2

Double Q-Learning

Initialize $Q_1(s,a)$ and $Q_2(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state},\cdot) = Q_2(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε-greedy in $Q_1 + Q_2$)
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S',a)\big) - Q_1(S,A)\Big)$$

    else:

$$Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S',a)\big) - Q_2(S,A)\Big)$$

    $S \leftarrow S'$;
  until S is terminal

Hado van Hasselt 2010

Double DQN

- Current Q-network w is used to select actions
- Older Q-network w− is used to evaluate actions

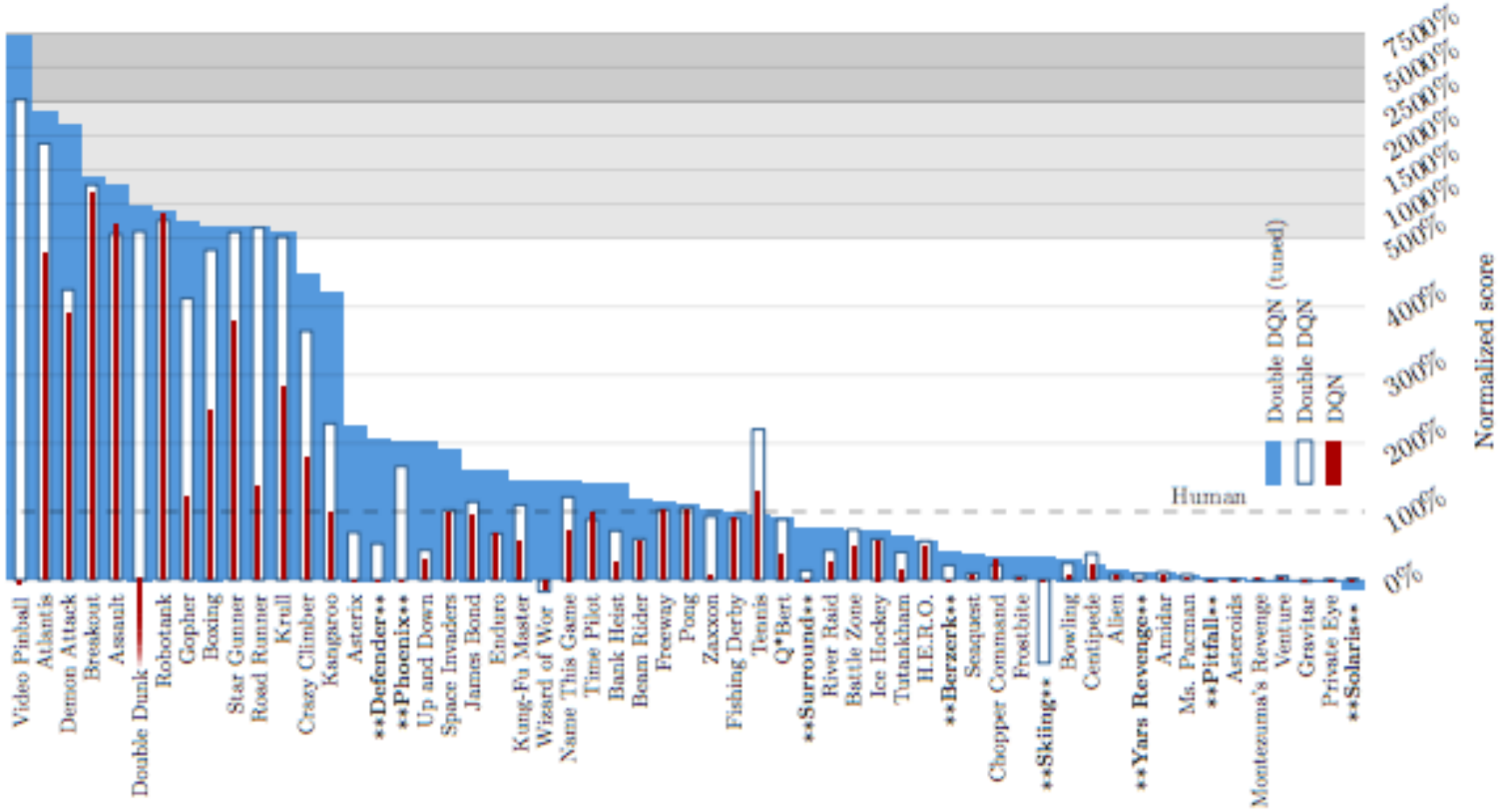
Action evaluation: w−

$$l = \left(r + \gamma Q(s', \underset{a'}{\text{argmax}}\, Q(s', a', \mathbf{w}), \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right)^2$$

Action selection: w

van Hasselt, Guez, Silver, 2015

Double DQN



van Hasselt, Guez, Silver, 2015