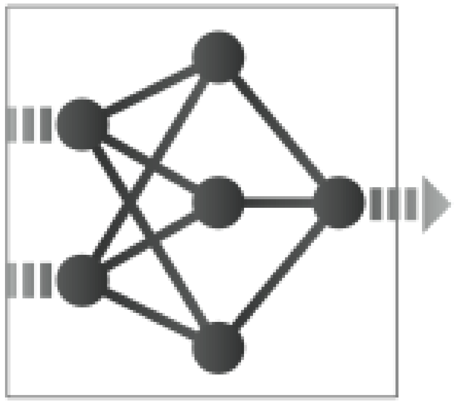
Deep & Convolutional Neural Networks
Reinforcement Learning



What you will learn in this class

- Supervised learning (continued from last week)
  - Deep learning with autoencoders
  - Deep Convolutional Neural Networks
- The Reinforcement Learning Framework
- Reward and Total Return
- The state-action value function (Q function)
- Value Learning
  - Deep Q Learning
- Policy Learning
  - Policy Gradient Learning

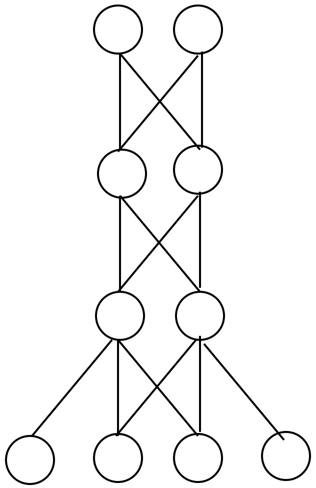
Deep vs. shallow neural networks

<u>Smaller number of weights = better generalization</u>

Compared to a network of *k layers*, a network of *k-1 layers* requires exponentially larger number of weights to achieve same learning error.
In addition, the k-1 layered network is likely to display worse generalization because it will have a comparatively higher number of weights

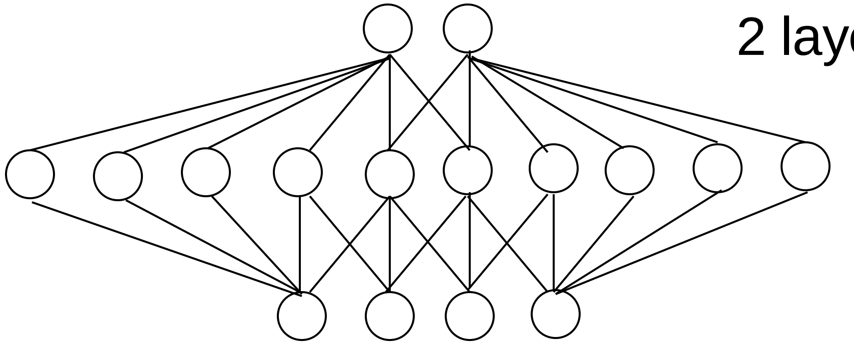
“deep”

3 layers of connections



“shallow”

2 layers of connections

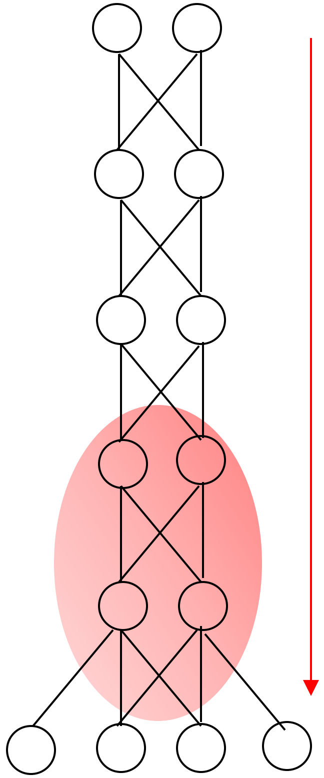


Not all connections are shown

Backpropagation in deep networks

However, Backpropagation yields poor results when applied to networks of many layers (k>3)

The problem lies in poor gradient estimation in the lower layers of the neural network, leading to smaller gradients and thus small weight modifications

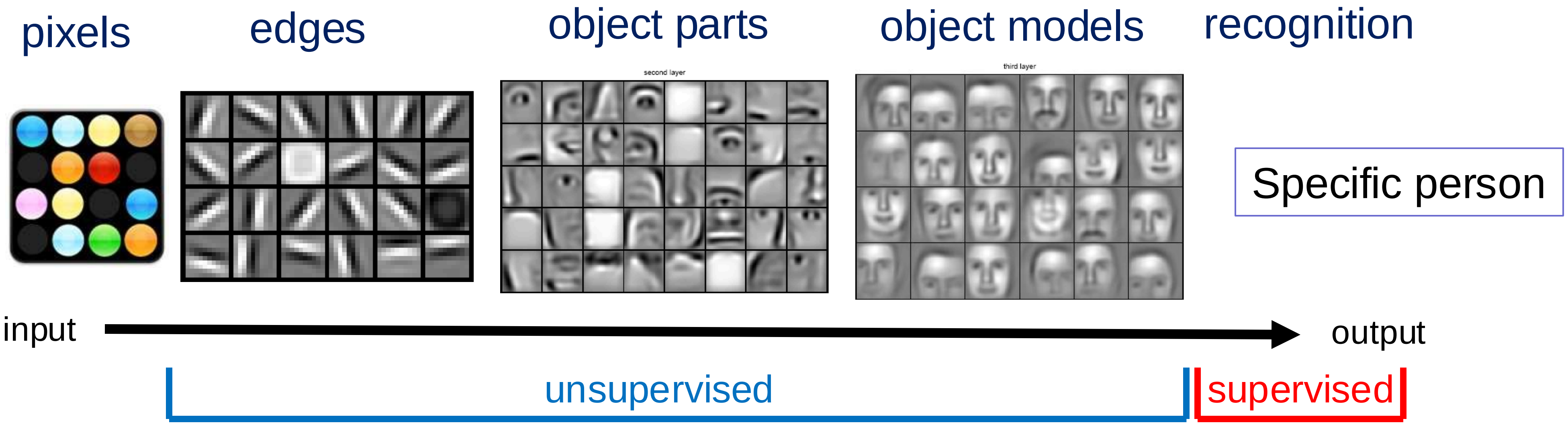


$$\delta_j = \dot{\Phi}\left(A_j\right)\sum_i w_{ij}\delta_i$$

Not all connections are shown

“Deep learning”, one layer at a time

Unsupervised training of lower layers to extract increasingly complex features of the input
Supervised training of top layer

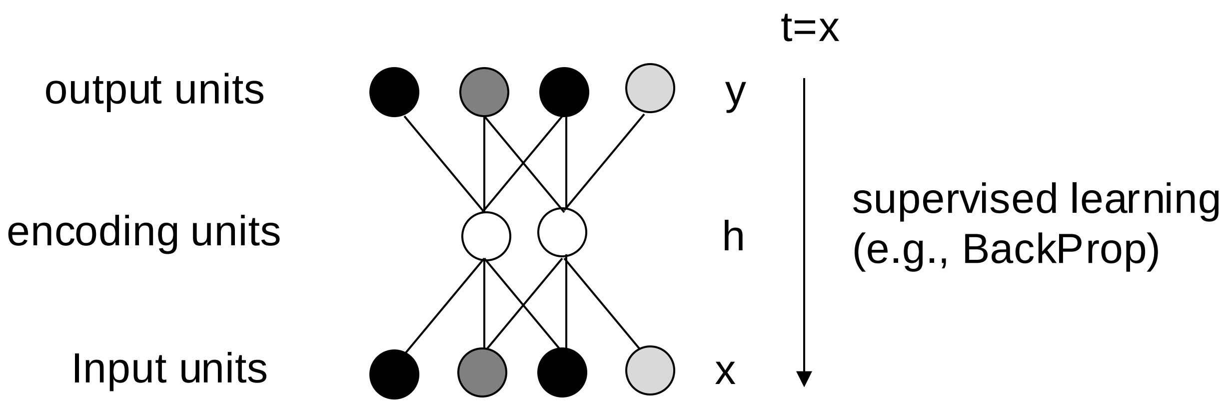


Hinton, Osindero, Teh, 2006
Bengio, Lamblin, Popovici, Larochelle, 2007
Ranzato, Poultney, Chopra, LeCun, 2007
See online also *Learning Deep Architectures for AI* by Yoshua Bengio, 2008

Unsupervised learning with Autoencoders

PCA (e.g., Oja's or Sanger's networks) are not suitable for deep networks because they are linear transformation of the input.

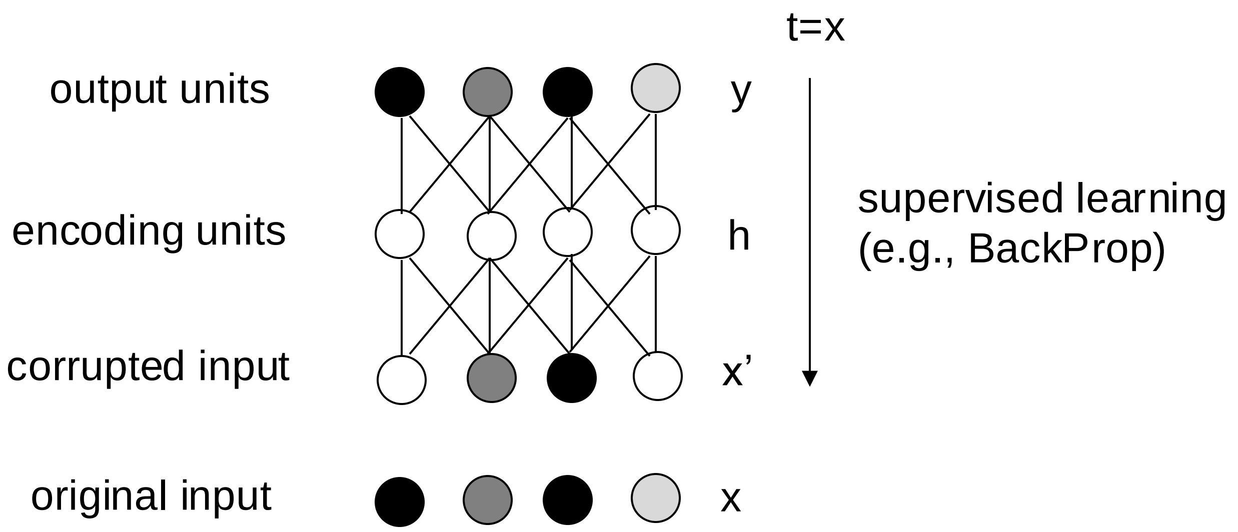
Autoencoders are non-linear supervised networks (e.g., Back-prop) that learn to reproduce the input pattern on the output layer. Usually, they have smaller set of hidden units (*encoding units*) which learn a compressed representation that spans the same space of PCA representation (but use non-linear units).



Not all connections are shown

Denoising Autoencoders (dropout)

Identity coding problem arises when encoding units are equal or larger than input units



To prevent identity encoding, use *denoising autoencoders* (Vincent et al. 2008): corrupt input by randomly switching off 50% of units while keeping teaching output equal to uncorrupted input

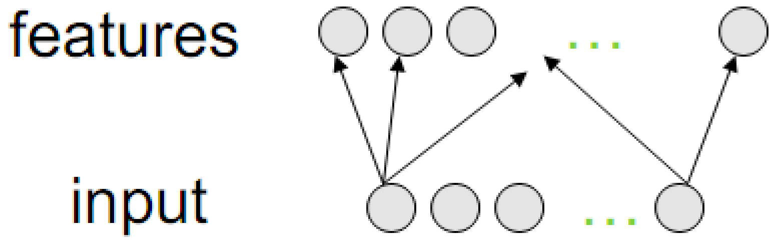
---

Not all connections are shown

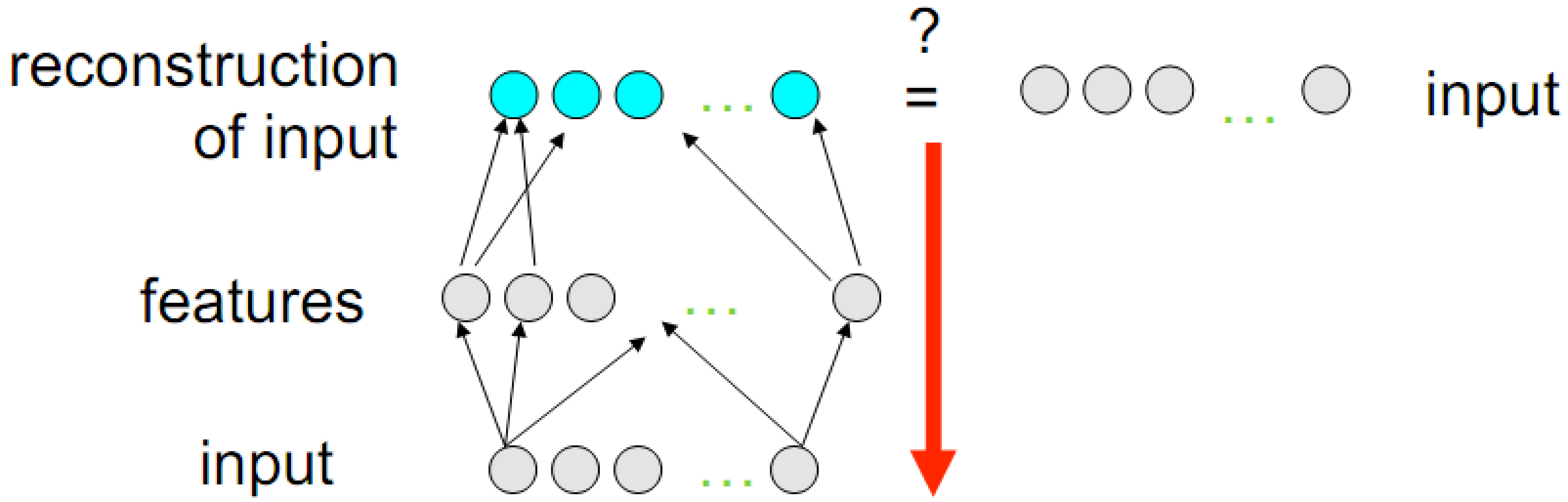
Deep training

input

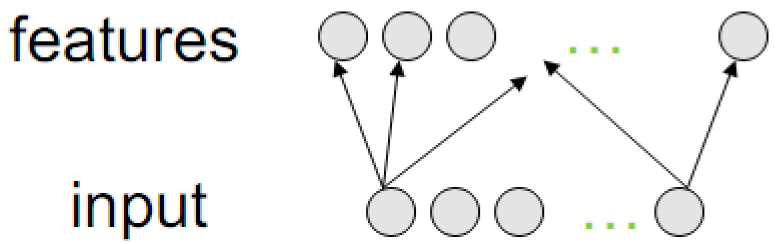
Layer-Wise Unsupervised Pre-training



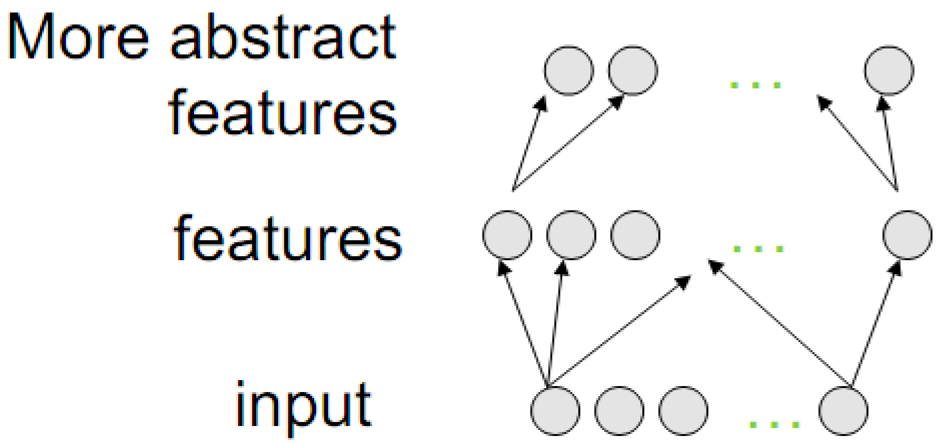
Layer-Wise Unsupervised Pre-training



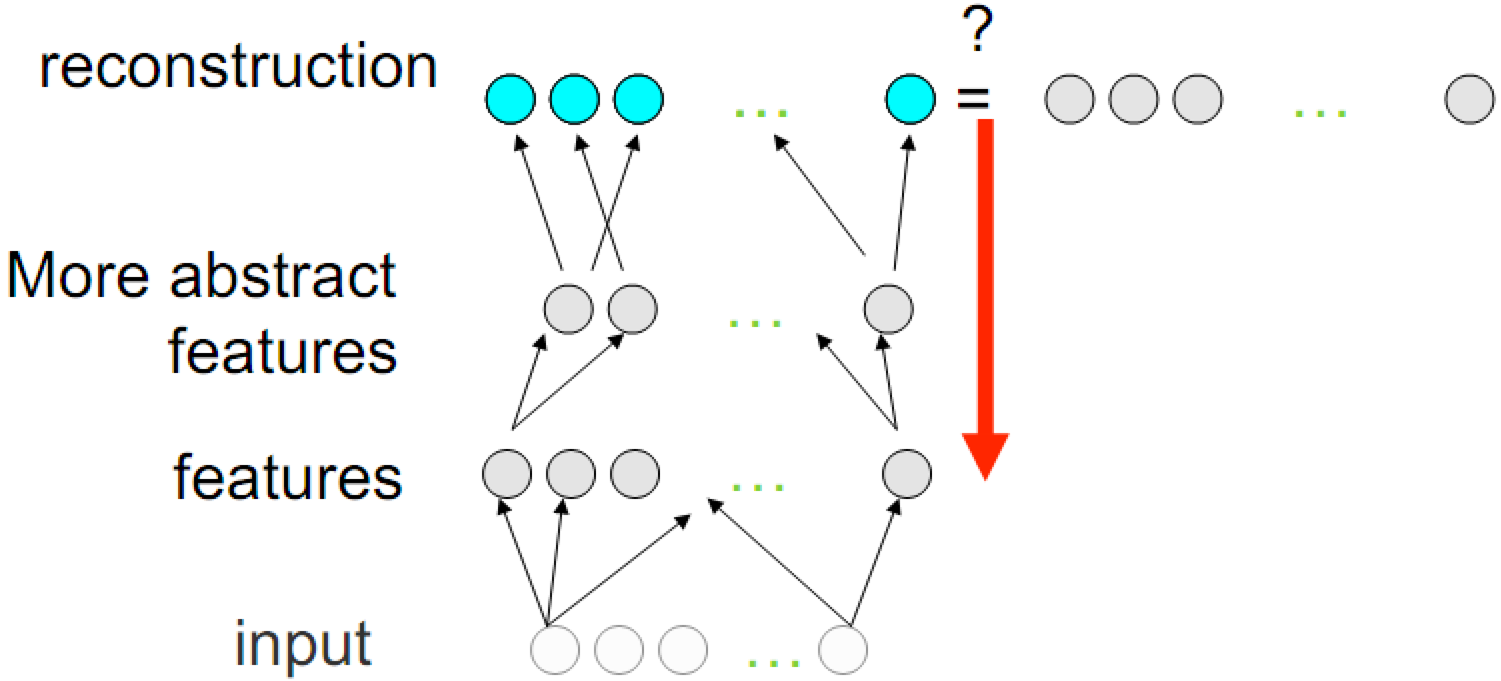
Layer-Wise Unsupervised Pre-training



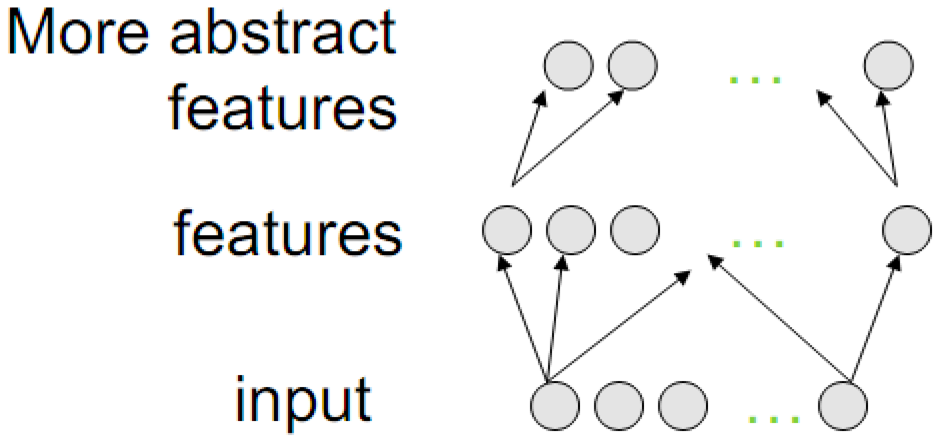
Layer-Wise Unsupervised Pre-training



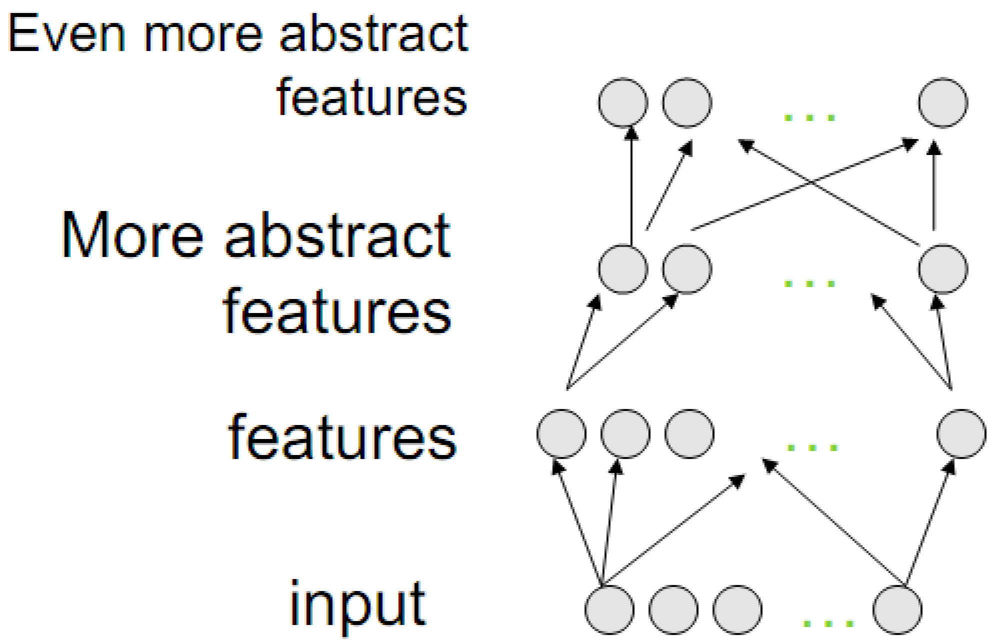
Layer-Wise Unsupervised Pre-training



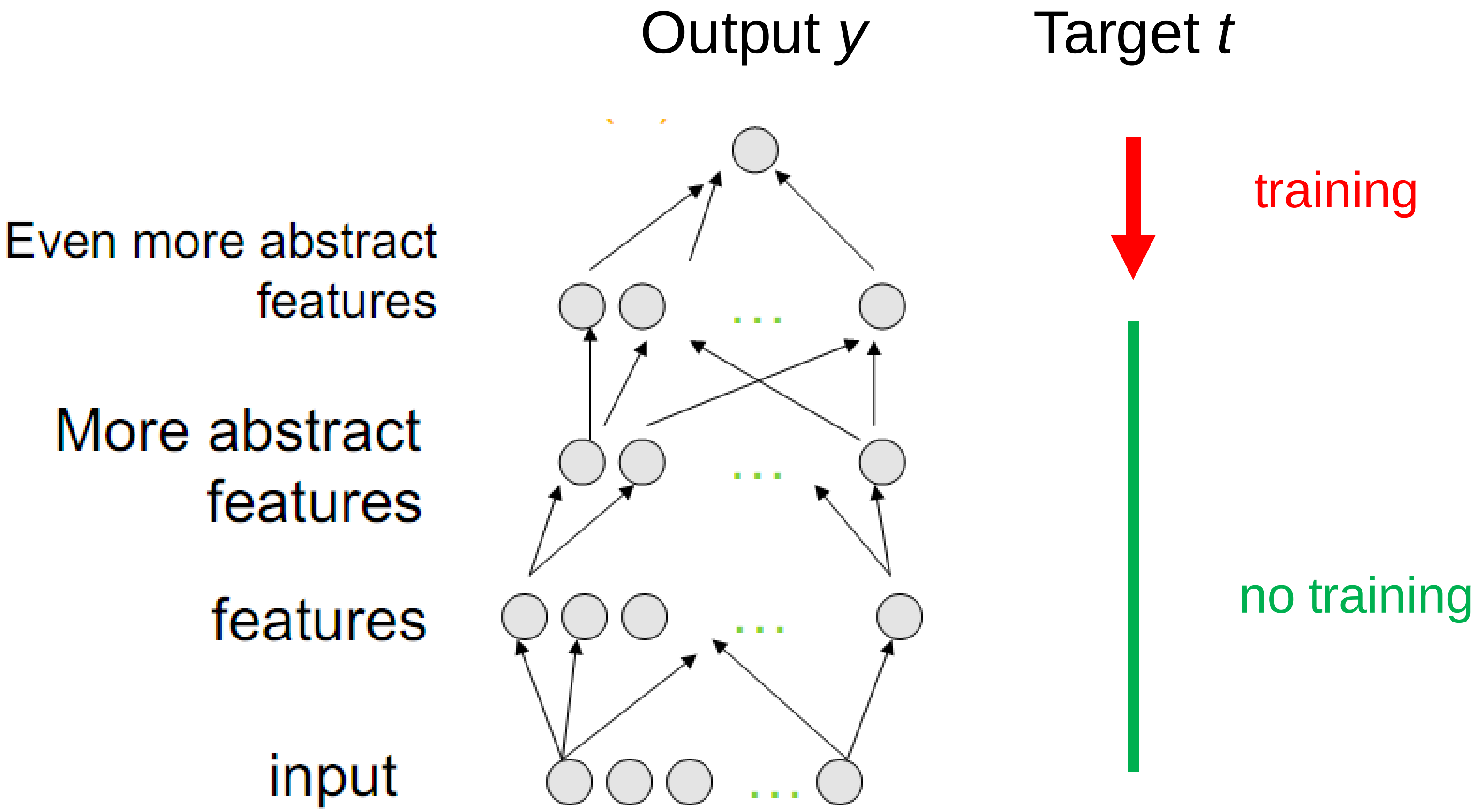
Layer-Wise Unsupervised Pre-training



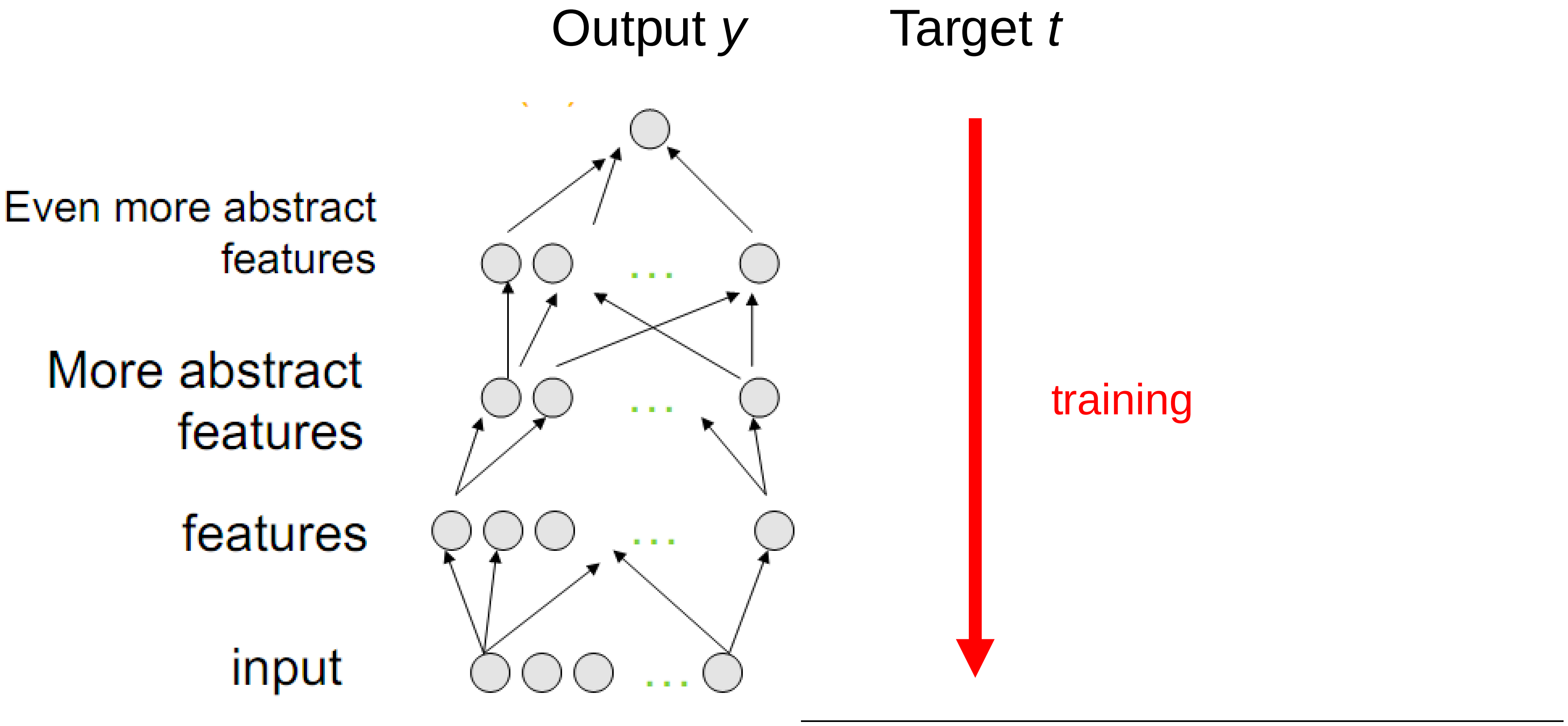
Layer-Wise Unsupervised Pre-training



Supervised training of top layer



Supervised fine tuning of entire network

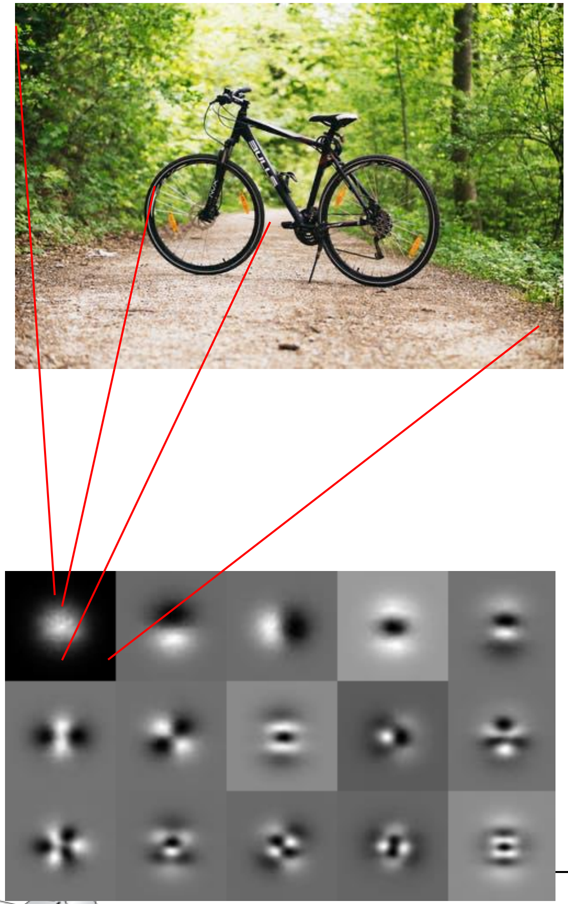


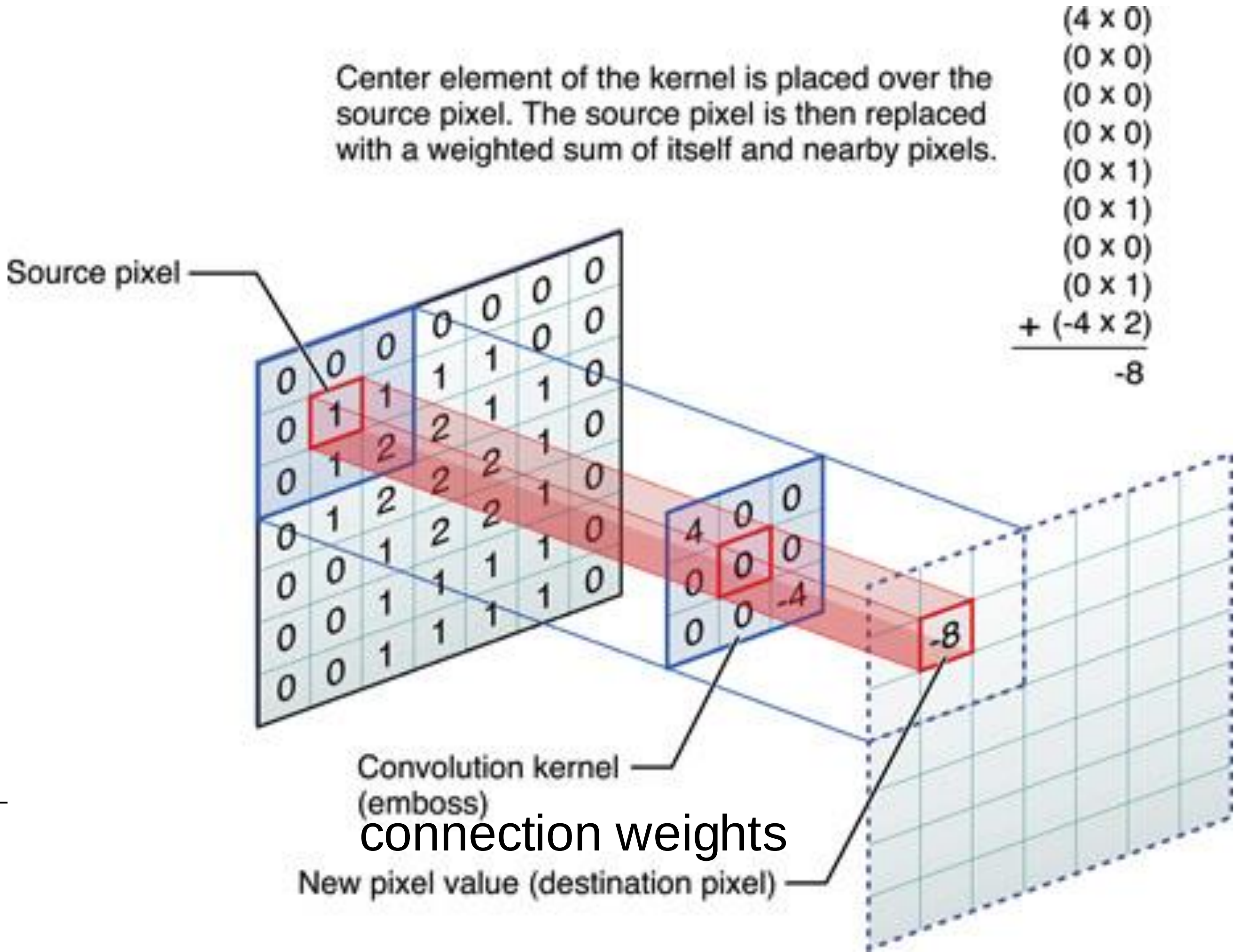
Features represent large data sets in a compact format

What do these images have in common?

Convolutional Neural Networks

Instead of training weights from all input units to each detector (filter), as autoencoders do, train only weights from few neighboring input units to each detector and convolve image to generate activations of the next layer





Filter convolution for 2D images

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

⋮ ⋮

Each filter is a feature detector

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Dot product

3 -1

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

If stride=2

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3 -3

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

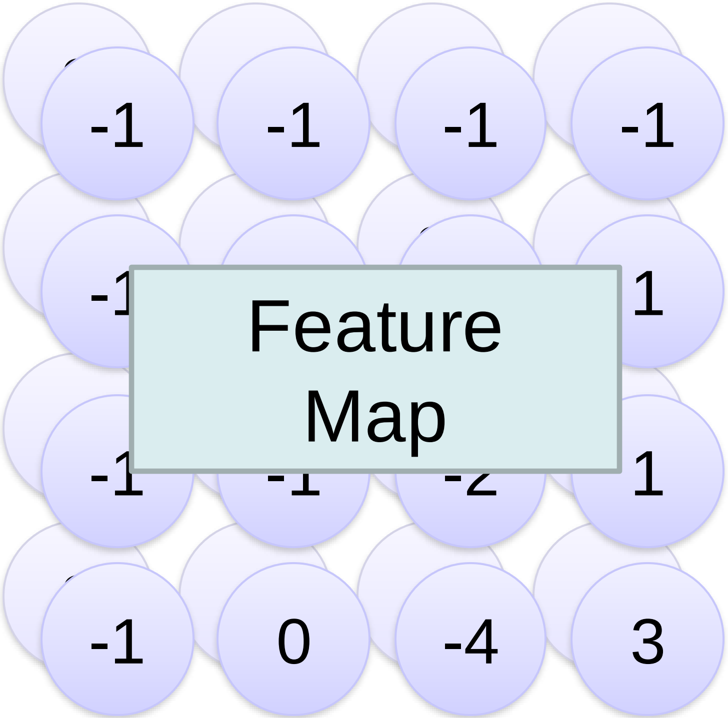
-1	1	-1	
-1	1	-1	Filter 2
-1	1	-1	

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Repeat this for each filter



x = image coordinate
y = image coordinate
d = convolutions (different filters)

Add non-linearity to each value in the block, e.g. ReLU function (Rectified Linear Unit)

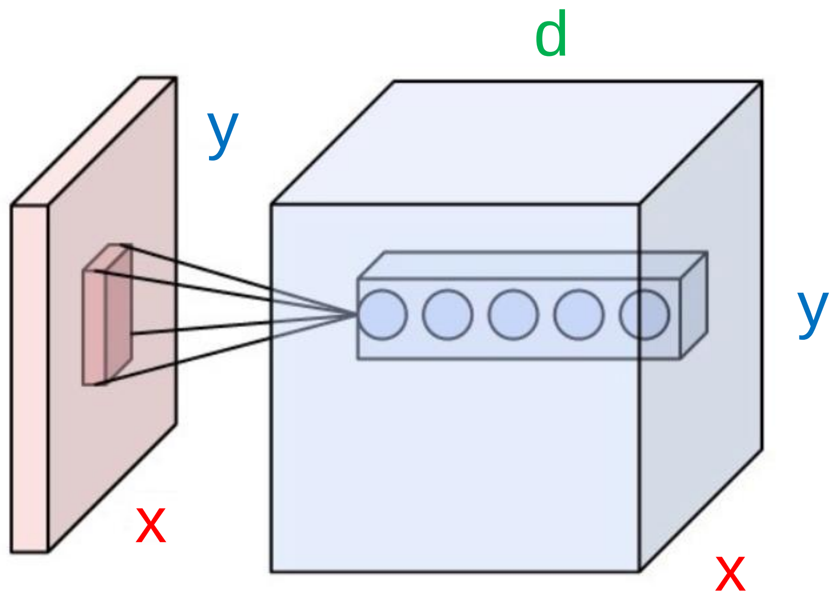
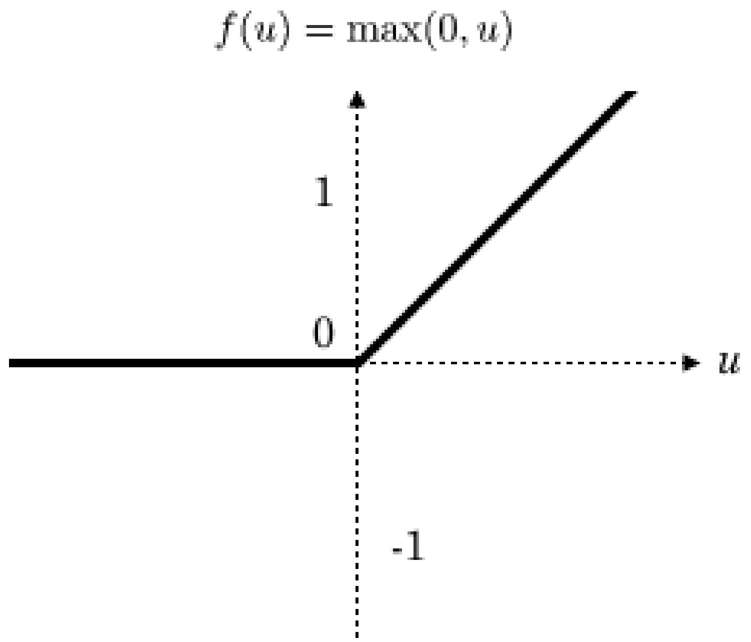
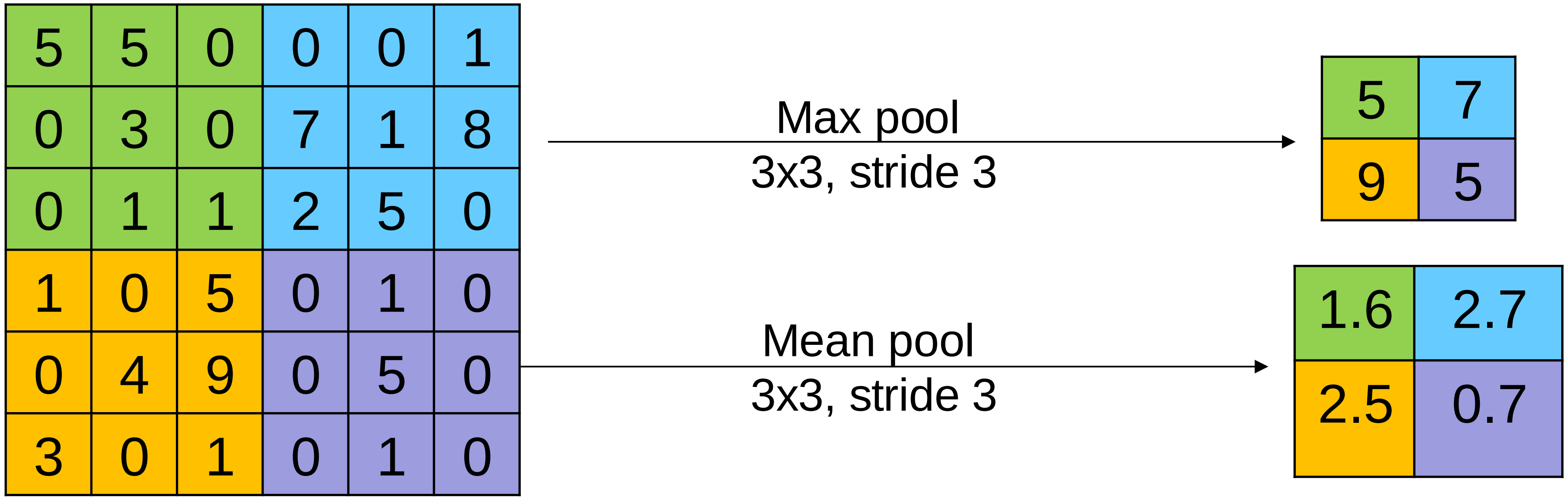


Image   Convolution block



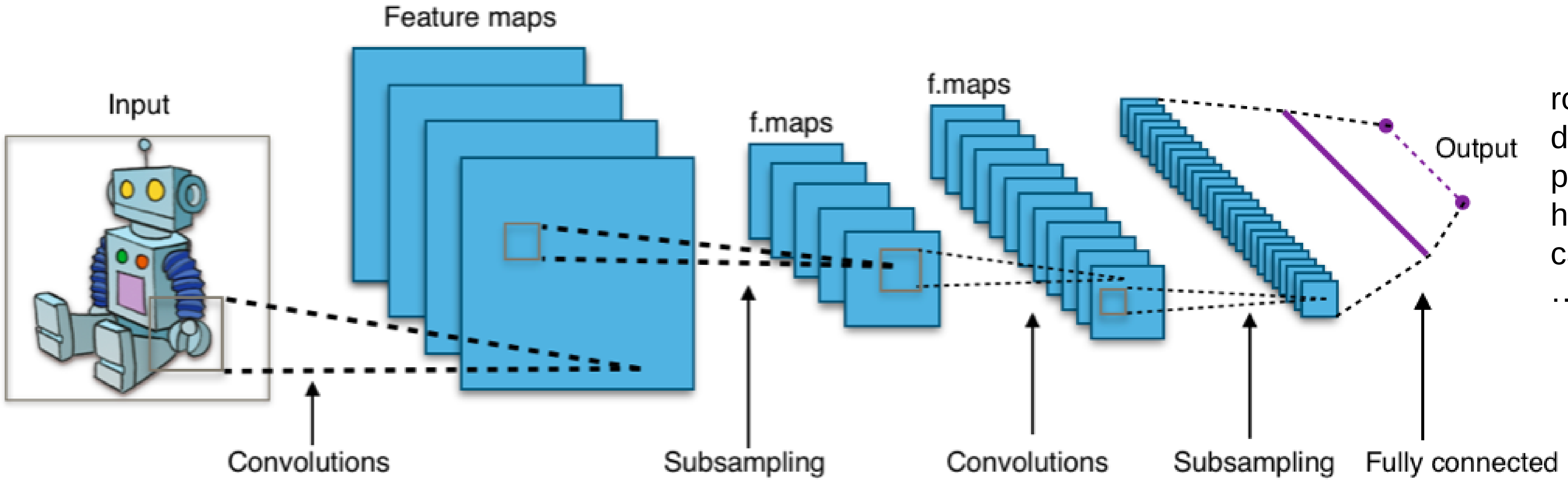
Reduce layer size by Subsampling

Layer is subdivided into pools (e.g., 3x3 neurons) and the content of each pool matrix is replaced by a single value, e.g. maximum or mean value of the pool

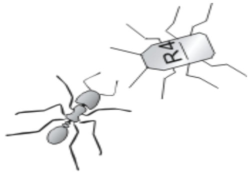


Typical Convolutional Neural Network

Only weights of one filter per layer are learned to minimize the error (loss) function



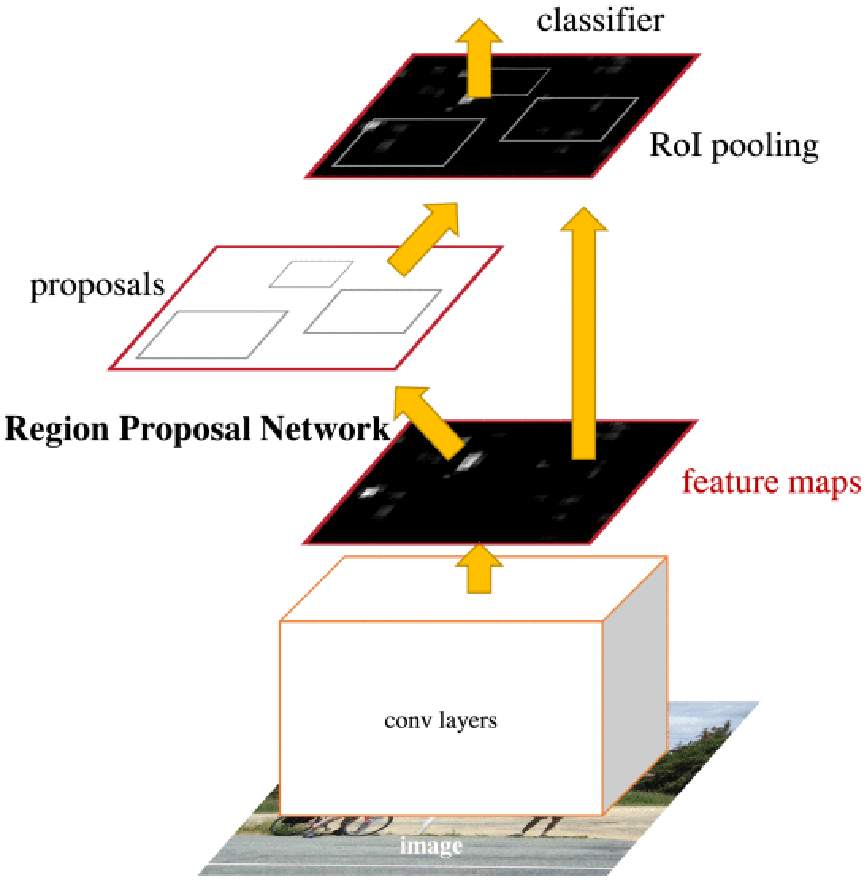
robot
dog
pigeon
human
car
…

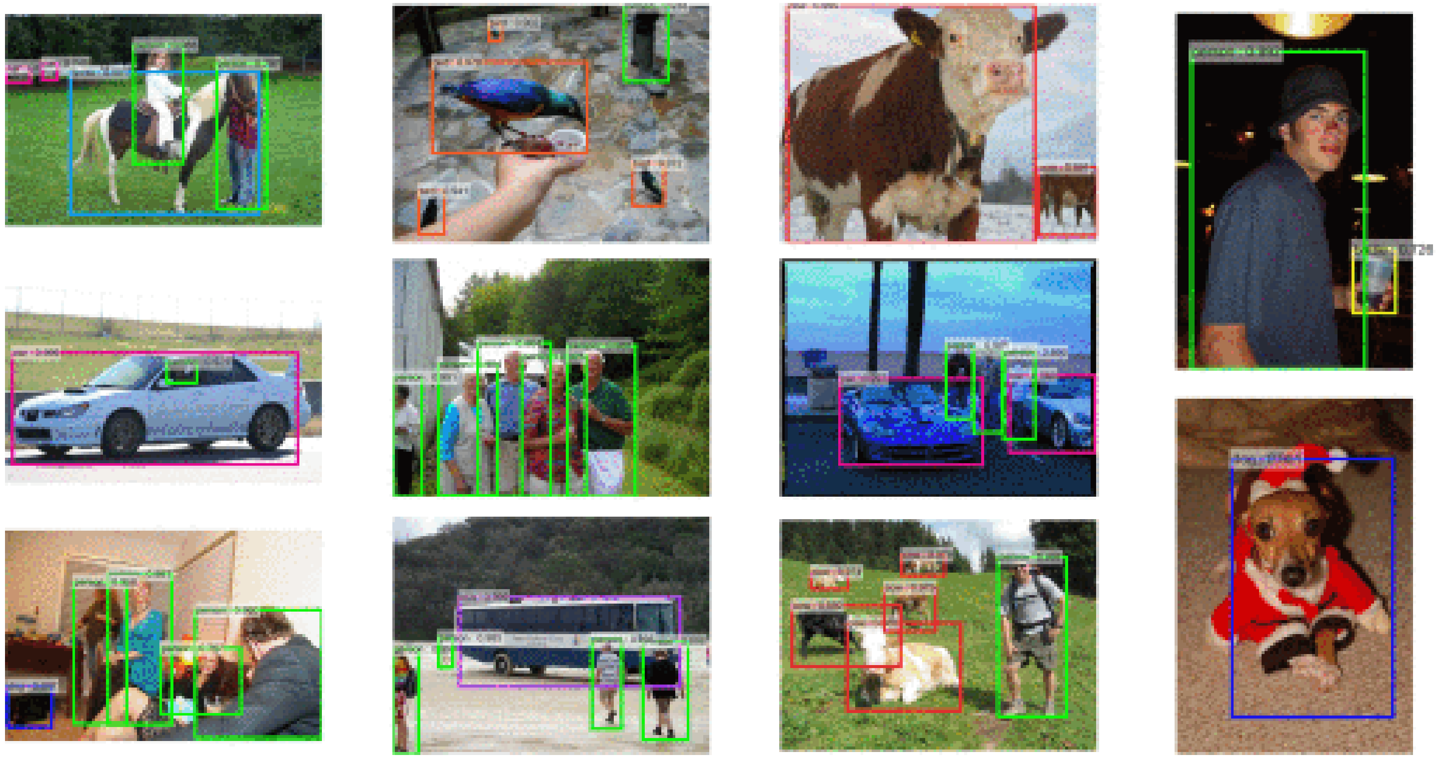


https://en.wikipedia.org/wiki/Convolutional_neural_network

Image by Aphex34 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=45679374

Learning object classification and positions





S. Ren, K. He, R. Girshick and J. Sun (2017), *IEEE Transactions on Pattern Analysis and Machine Intelligence*, doi: 10.1109/TPAMI.2016.2577031.

Reinforcement learning

Input: *state* (sensory information, position, energy, e.g.), *action* (forward, rotate, turn, e.g.)

Reward: *r* (collected dirt, e.g.)

Goal: learn *behavior* (<u>policy</u>) that maximizes the total future rewards

Reinforcement learning framework

AGENT

State s_{t+1}

Reward r_t
can be positive,
negative, or
absent

ENVIRONMENT

Action a_t

The agent wants to find a mapping from states to actions (the *policy*) that maximizes the total future reward (the *Total Return*) $R_t = \sum_{i=t}^{\infty} r_i$

Reward discount and rollouts

The *discount* factor γ is used to give more importance to present rewards than to remote future rewards

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i \qquad\qquad 0 < \gamma < 1$$

Rollout: the finite number of steps *n* during which the agent interacts with the environment until a terminal event or time limit is reached

$$R_t = \gamma^t r_t + \gamma^{t+1} r_{t+1} + \gamma^{t+2} r_{t+2} \cdots + \gamma^{t+n} r_{t+n}$$

The Q Function

The total return R_t is the discounted sum of all future rewards

$$R_t = \gamma^t r_t + \gamma^{t+1} r_{t+1} + \gamma^{t+2} r_{t+2} \cdots + \gamma^{t+n} r_{t+n}$$

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$

The Q function describes the *expected* total return that an agent in state s can receive by performing a certain action a. It can be visualized as a look-up table that the agent gradually builds by summing up the observed rewards in several rollouts; for example (*fictitious numbers!*):

Rewards	**Action A**	**Action B**
State A	3	-3
State B	1	0
State C	2	0

Q values	**Action A**	**Action B**
State A	0	0
State B	-2	4
State C	-6	0

Finding the optimal policy

s_a, a?
s_b, a?
s_c, a?
s_d, a?
…

A policy $\pi(s)$ is a strategy to select an action a for a state s

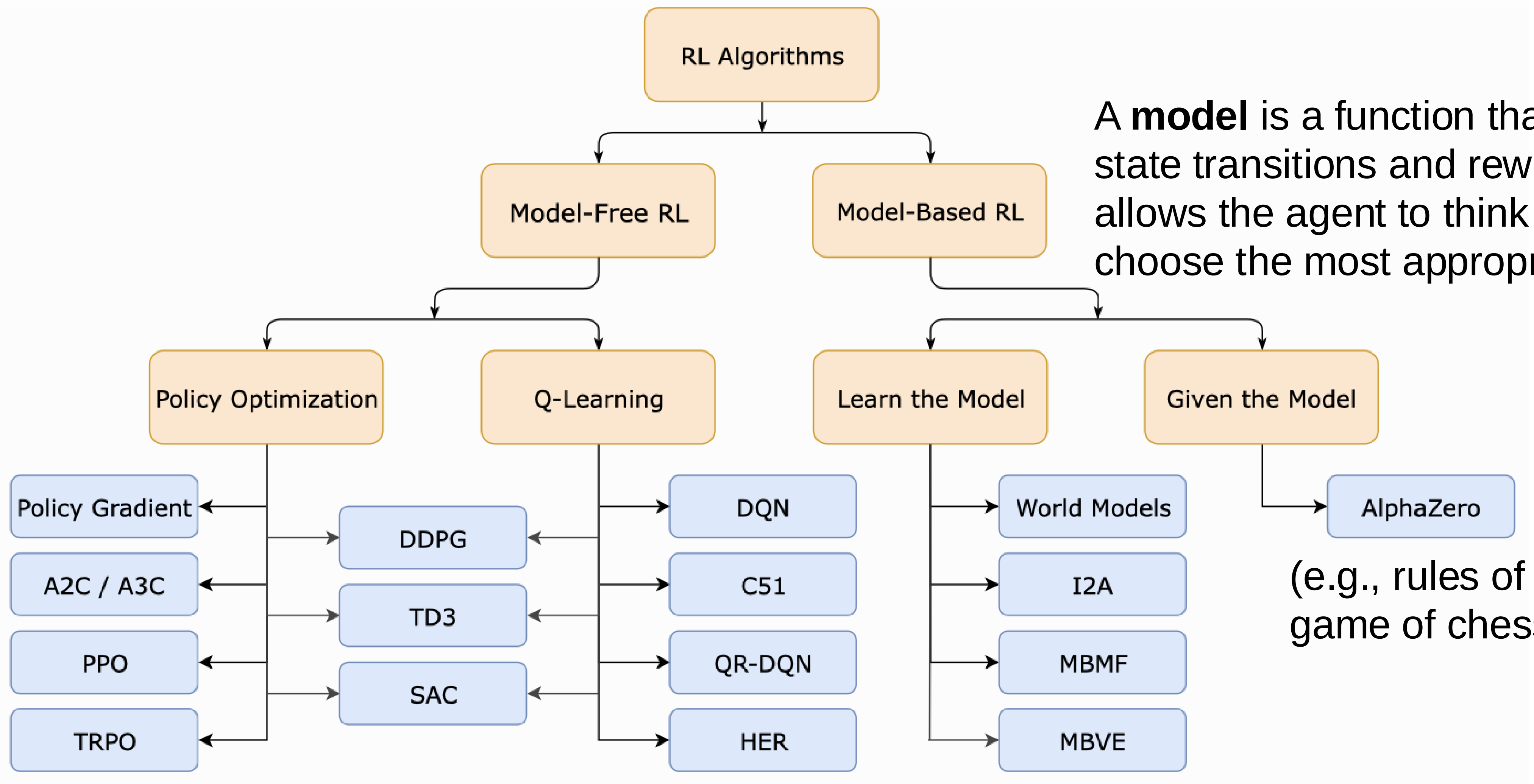
The optimal policy $\pi^*(s)$ is a policy that maximizes the expected total return, which is described by the Q function

If the agent knows the Q function, the optimal policy consists in finding for each state s the best action a over all possible actions that maximize the Q function

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q(s,a)$$

Adapted from MIT 6.S191: Reinforcement Learning, by Alexander Amini

A taxonomy of modern RL algorithms (2018)



A **model** is a function that predicts state transitions and rewards: it allows the agent to think ahead and choose the most appropriate action

(e.g., rules of the game of chess)

Source: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Model-free RL Methods

Q-VALUE LEARNING

Find

$Q(s,a)$

and pick best action

$$a = \underset{a}{\operatorname{argmax}} Q(s,a)$$

POLICY LEARNING

Directly find

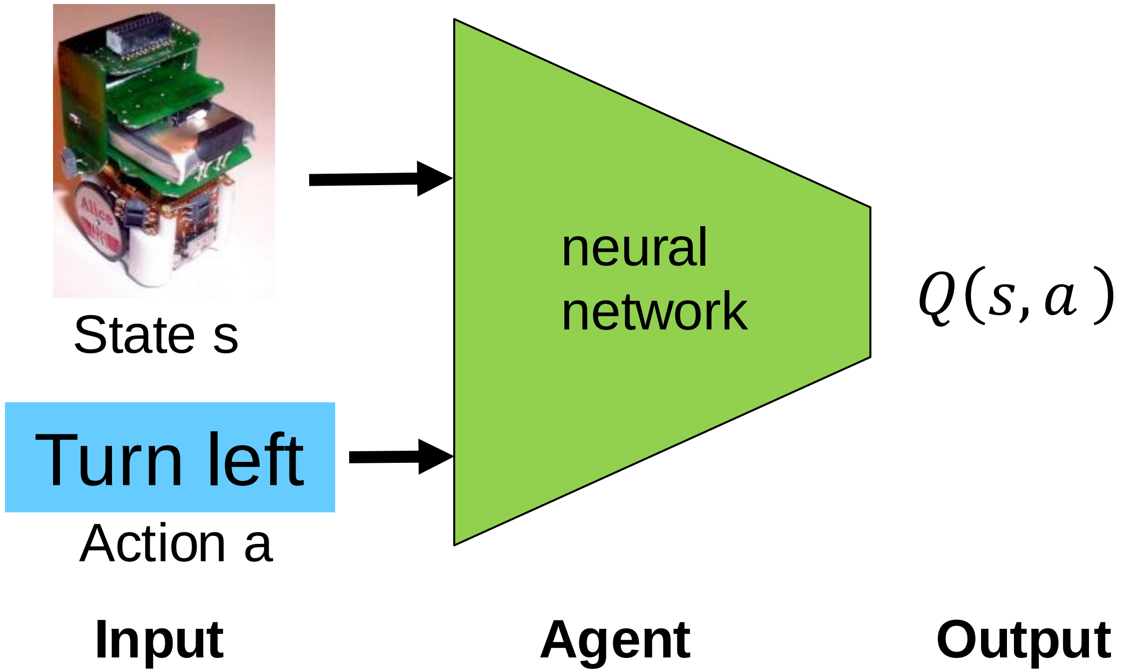
$\pi(s)$

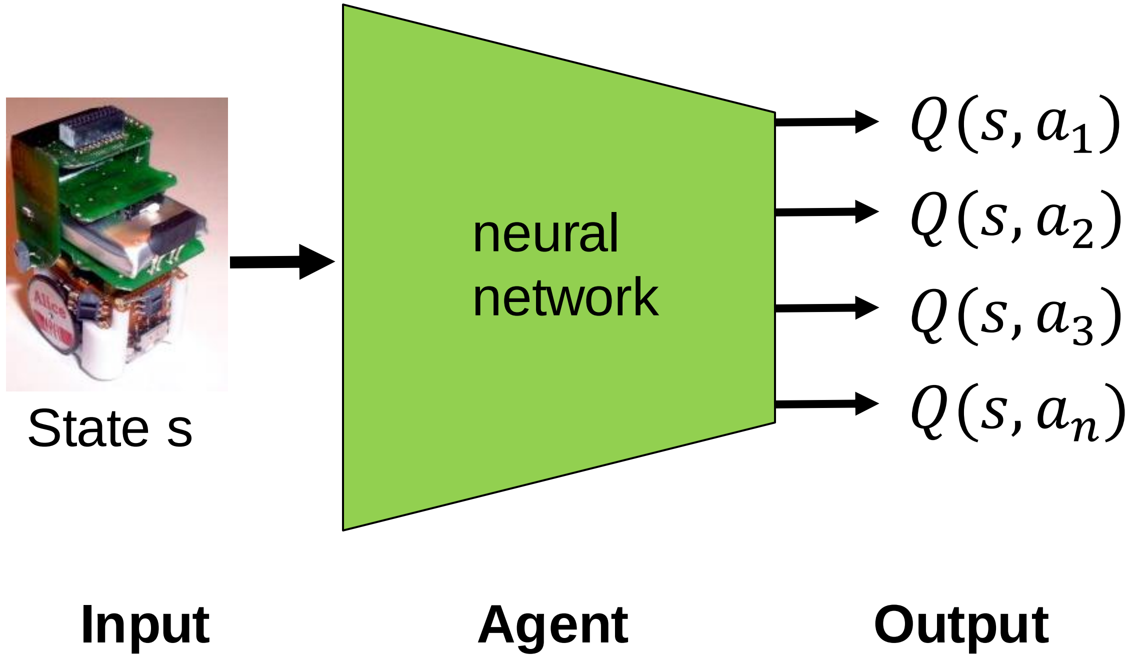
and sample (try) action

$$a \sim \pi(s)$$

Deep Q-Networks (DQN)

DQN assumes a <u>discrete action space</u>

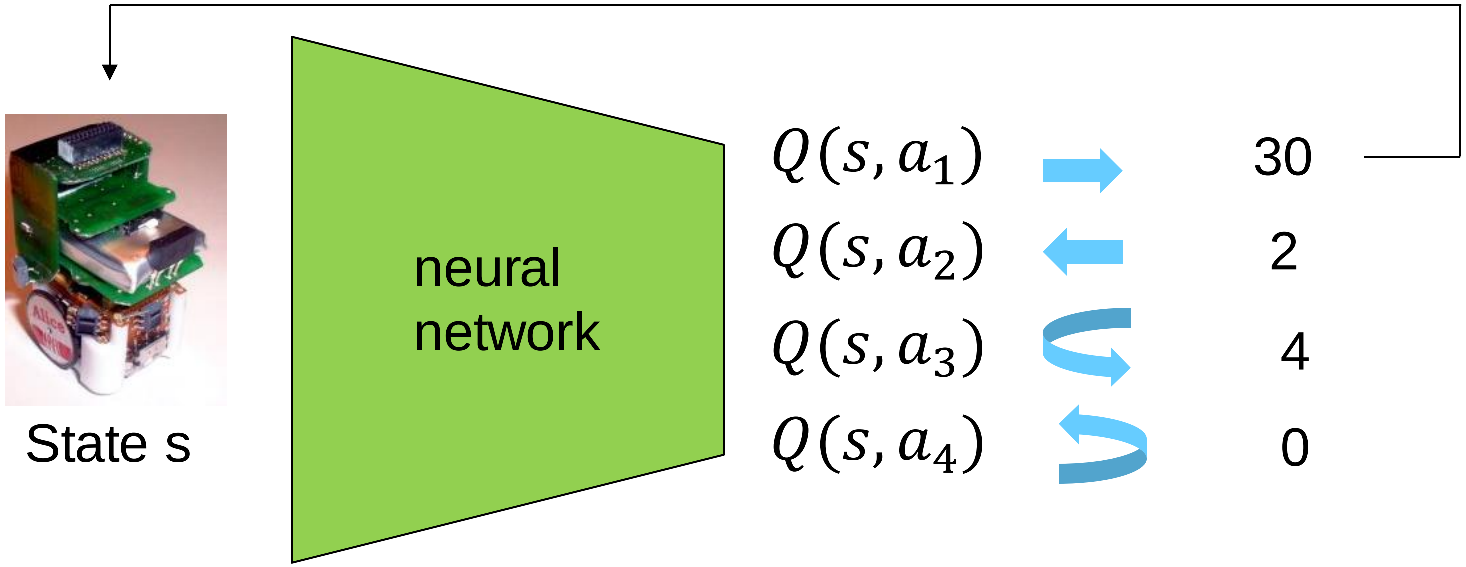




Problem: Q value must be recomputed for all possible actions at input state s

Solution: ask network to compute Q values for all possible actions of input state s

DQN learning



- Initialize random weights
- Select random action with small probability ε, otherwise select action with highest prediction value
- After termination event, compute Q loss and perform gradient descent on weights

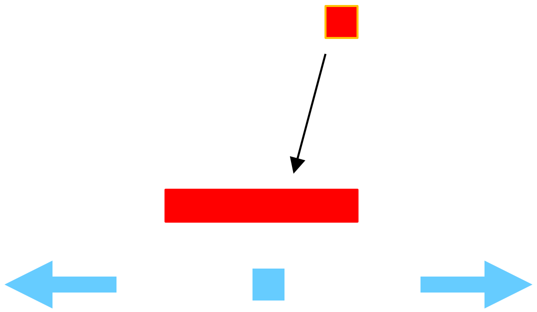
$$\text{Q-loss} = \mathbb{E}\left[\left\| \overbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}^{\text{Observation}} - \overbrace{Q(s, a)}^{\text{Prediction}} \right\|^2\right]$$

Use back-propagation of error to adapt network weights

DQN learning to play Atari Breakout game

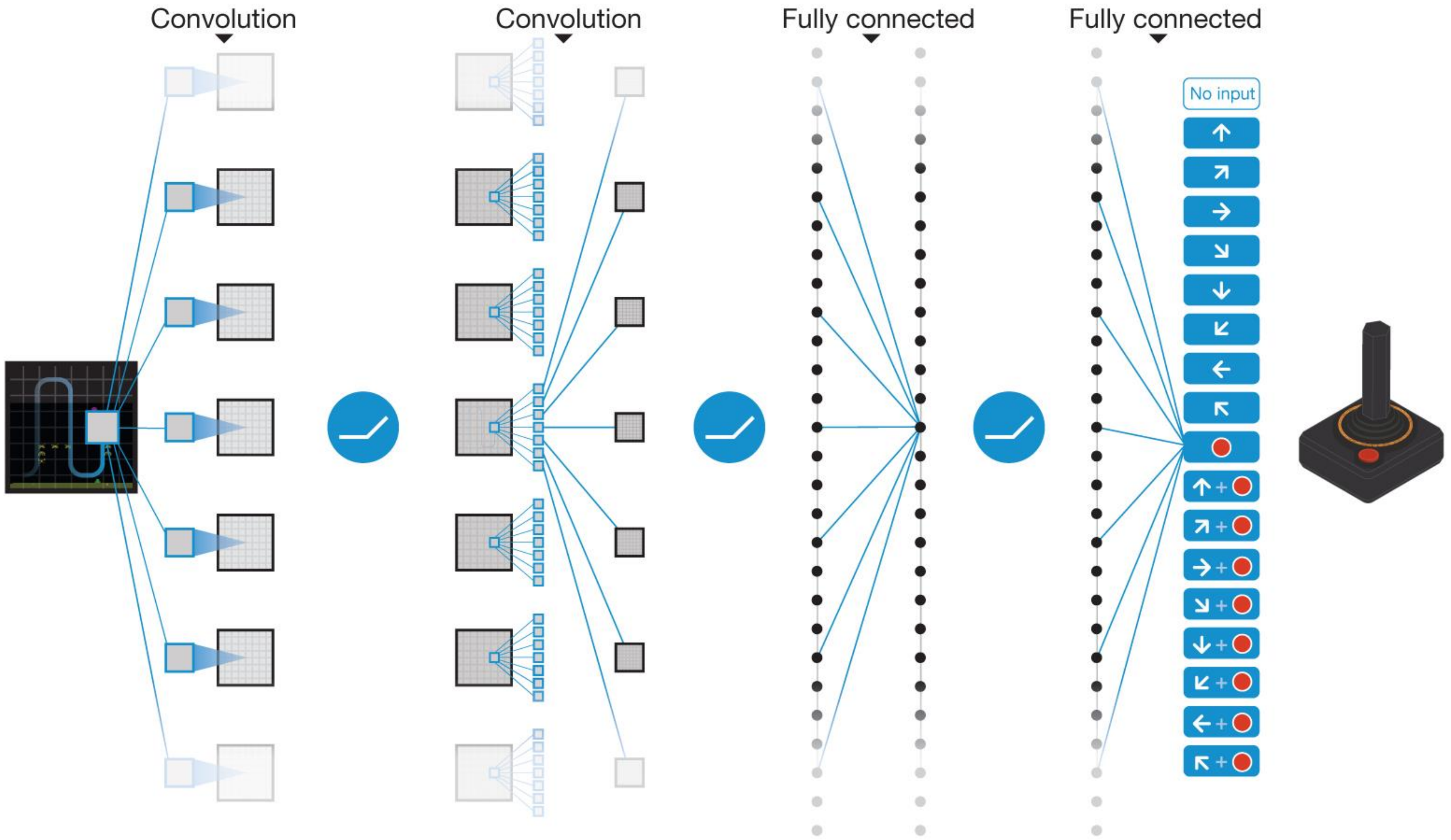
State = screen image

Paddle actions = left, stay, right

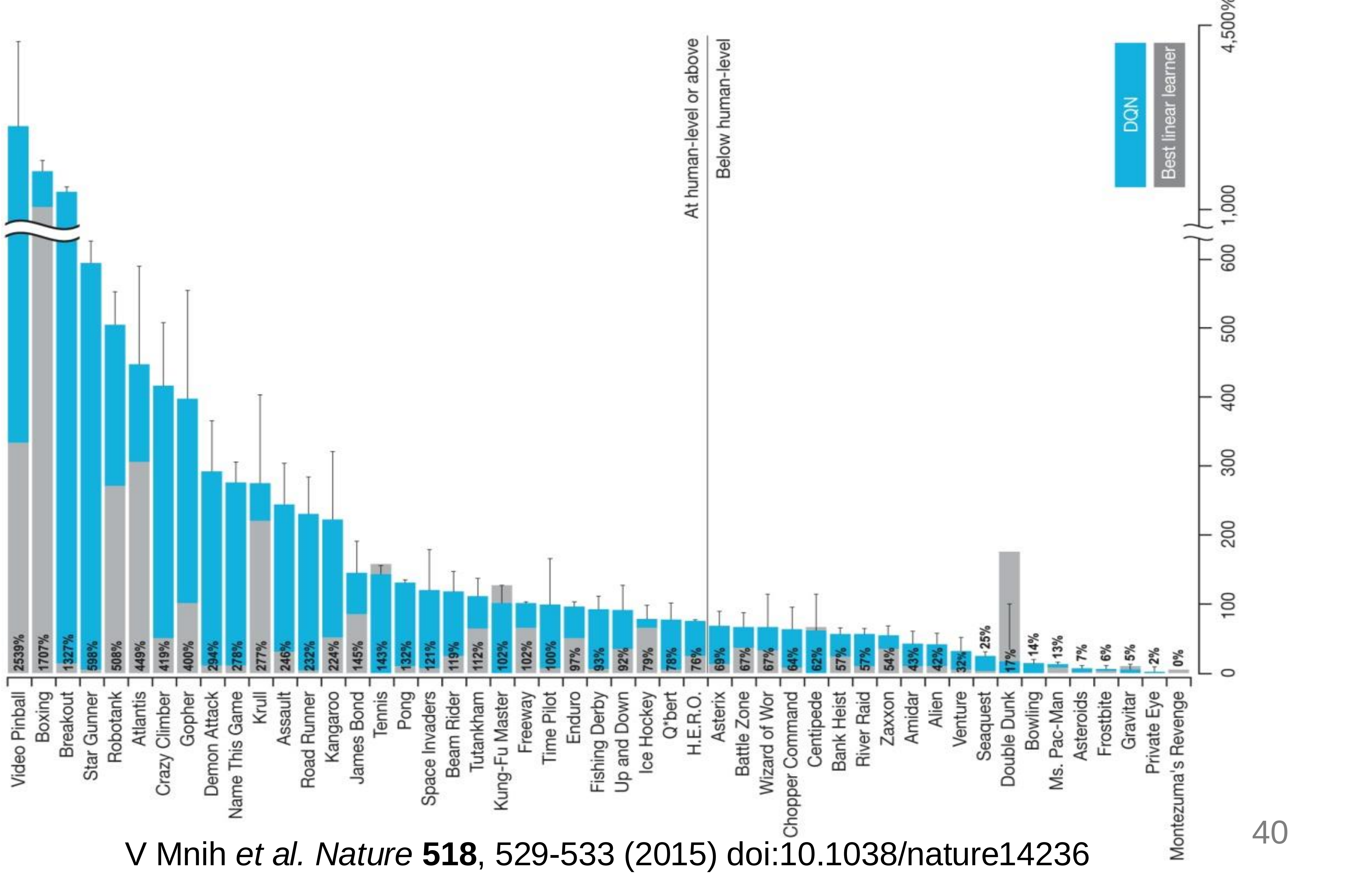


https://www.youtube.com/watch?v=V1eYniJ0Rnk

DQN playing Atari games



V Mnih *et al. Nature* **518**, 529-533 (2015) doi:10.1038/nature14236



V Mnih *et al.* *Nature* **518**, 529-533 (2015) doi:10.1038/nature14236

Q learning: strengths and limitations

It guarantees the possibility of identifying the optimal policy if the Q function is learned

BUT

It requires a discrete action space (turn left, go forward, stay, etc.)

It only works for deterministic situations (it cannot learn stochastic policies)

Policy learning

Directly learn the policy $\pi(s)$: <u>discrete action space</u>

State s

neural network

probabilities

$P(a_1|s)$ 0.4

$P(a_2|s)$ 0.3

$P(a_3|s)$ 0.3

$P(a_4|s)$ 0.0

Probability Distribution Function must sum to 1

$\pi(s) \sim P(a|s)$

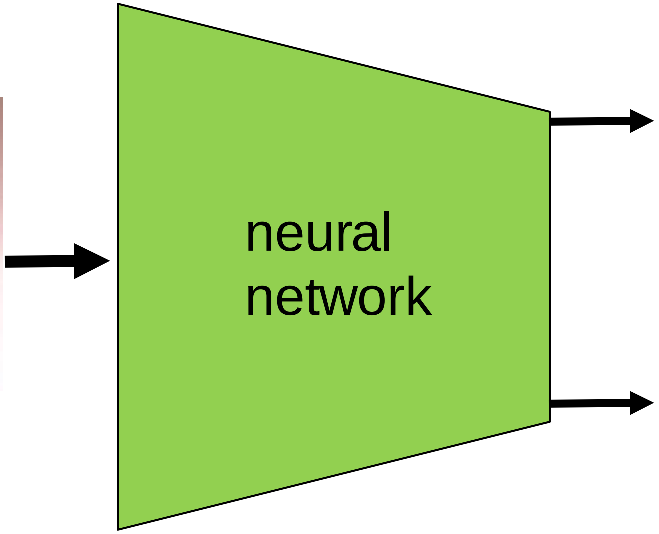
Sample the probability distribution to select action: for example, a_1

Input **Agent** **Output**

Policy learning

Directly learn the policy $\pi(s)$: <u>continuous action space</u>

State s

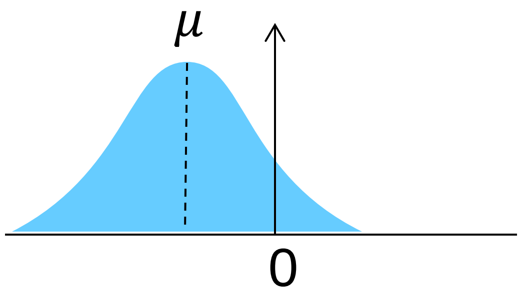


$\mu = -0.8$

$\sigma^2 = 0.5$

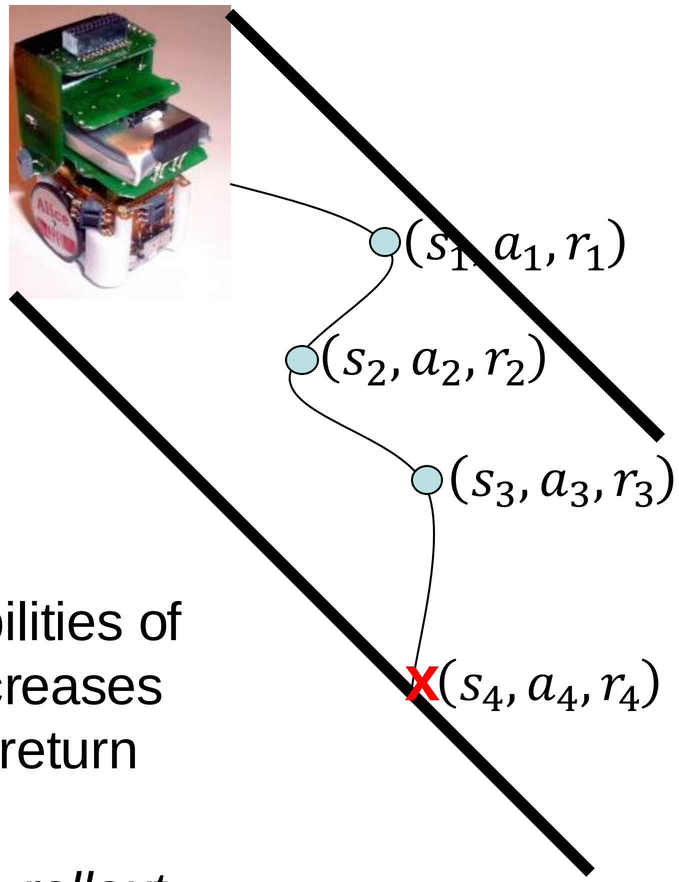
Input **Agent** **Output**

$P(a|s) = \mathcal{N}(\mu, \sigma^2)$



Policy Gradient Learning

1. Initialize weights of the agent
2. Run the agent (*policy*) until termination (*rollout*)
3. At each time step of the rollout, record the triplet (s_t, a_t, r_t)
4. Increase probability of actions that led to high reward
5. Decrease probability of actions that led to low reward



$$\boxed{loss = -\log P(a_t|s_t)\, R_t}$$

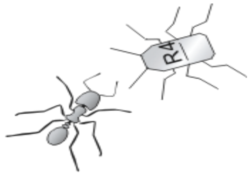
The loss function increases the probabilities of actions with higher total return and decreases probabilities of actions with lower total return

$$\Delta w = -\nabla loss$$
$$\Delta w = \nabla \log P(a_t|s_t)\, R_t$$

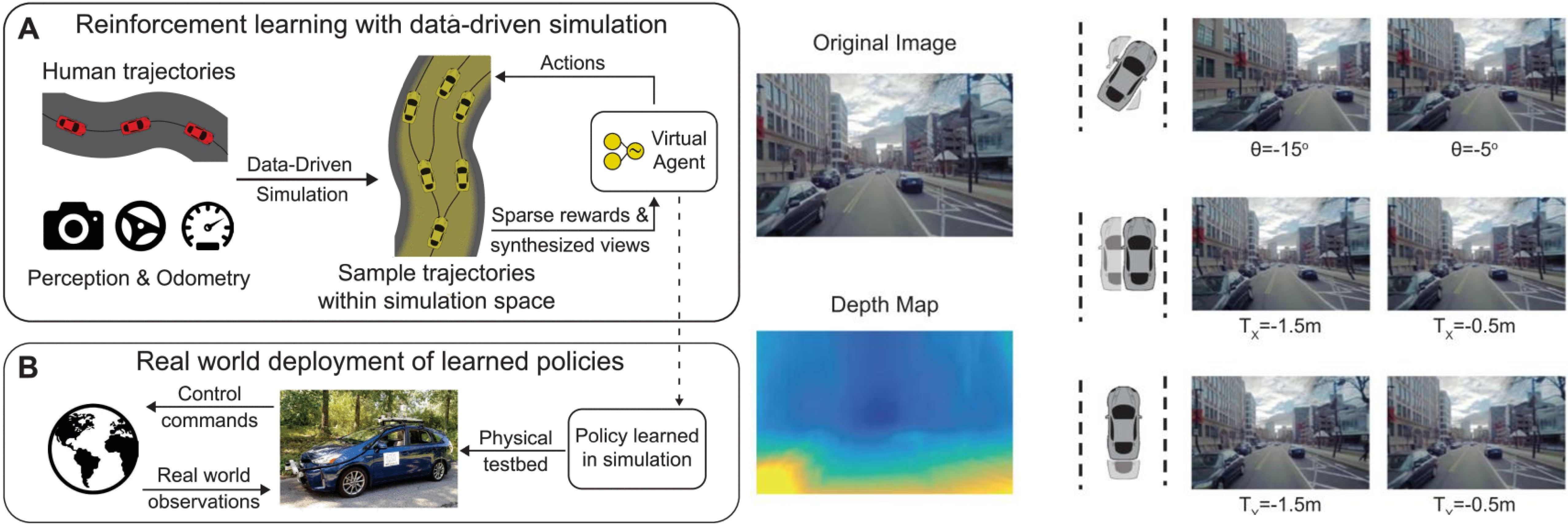
Weight change is performed after each *rollout*

For full derivation; https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

An alternative method that does not use gradient ascent is evolutionary computation



Autonomous driving by Policy Gradient Learning



A. Amini *et al*., Learning Robust Control Policies for End-to-End Autonomous Driving From Data-Driven Simulation, (2020) *IEEE Robotics and Automation Letters*, 5(2), 1143-1150

Contributions

Our paper makes the following contributions