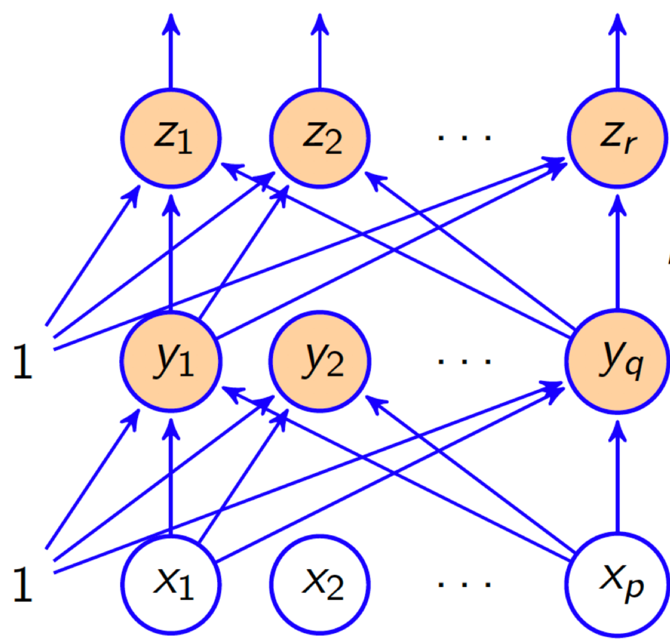
Deep Reinforcement Learning
CS440/ECE448 Lecture 22

Slides by Svetlana Lazebnik, 11/2017
Modified by Mark Hasegawa-Johnson, 4/2018



$$\vec{z} = h(\vec{x}, U, V)$$

$$z_\ell = g(b_\ell) \qquad \vec{z} = g(\vec{b})$$

$$b_\ell = v_{k0} + \sum_{k=1}^{q} v_{\ell k} y_k \qquad \vec{b} = V\vec{y}$$

$$y_k = f(a_k) \qquad \vec{y} = f(\vec{a})$$

$$a_k = u_{k0} + \sum_{j=1}^{p} u_{kj} x_j \qquad \vec{a} = U\vec{x}$$

$\vec{x}$ is the input vector

Last time: Q-learning for discrete s, a

- So far, we've assumed a *lookup table* representation for utility function U(s) or action-utility function Q(s,a)
- This does not work if the state space is really large or continuous

This time: Function approximation

- Approximate $Q(s, a)$ by a ***parameterized function***, that is, by a function $\hat{Q}(s, a; W)$ that depends on some matrix of trainable parameters, *W*.
- Learn W by playing the game.

Outline

- One-layer neural net
- Multi-layer neural net
- Training a neural net
- On-line Q-learning
- Policy learning
- Imitation learning

One-layer neural network

- Suppose you have a vector of input features,

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix}$$

- Your trainable parameters are a weight matrix and an offset vector,

$$W = \begin{bmatrix} w_{11} & \cdots & w_{1p} \\ \vdots & \vdots & \vdots \\ w_{q1} & \cdots & w_{qp} \end{bmatrix}, \vec{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_q \end{bmatrix}$$

One-layer neural network

- First, compute an affine transform using parameters W and $\vec{b}$:

$$\vec{a} = W\vec{x} + \vec{b}$$

- Second, compute element-wise nonlinearity:

$$\vec{y} = g(\vec{a})$$

…by which we mean that $y_k = g(a_k)$ for each element k.

- The goal of machine learning: to find parameters W and $\vec{b}$, and a nonlinearity $g(\vec{a})$, so that $\vec{y}$ is as close as possible to the function you want.

What about that nonlinearity?

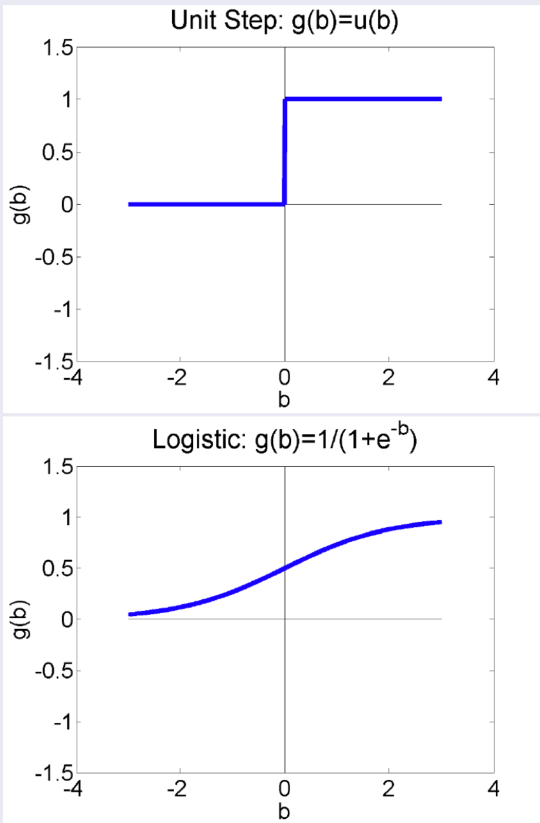
- The nonlinearity $g(a) = u(a)$, the unit step, is appropriate if the output should always be either 0 or 1.
- The nonlinearity $g(a) = sgn(a)$, the signum function, is appropriate if the output should always be either -1 or +1.
- The max nonlinearity is appropriate for a multi-class classification problem:

$$g(a_k) = \begin{cases} 1 & a_k = \max_j a_j \\ 0 & \text{else} \end{cases}$$

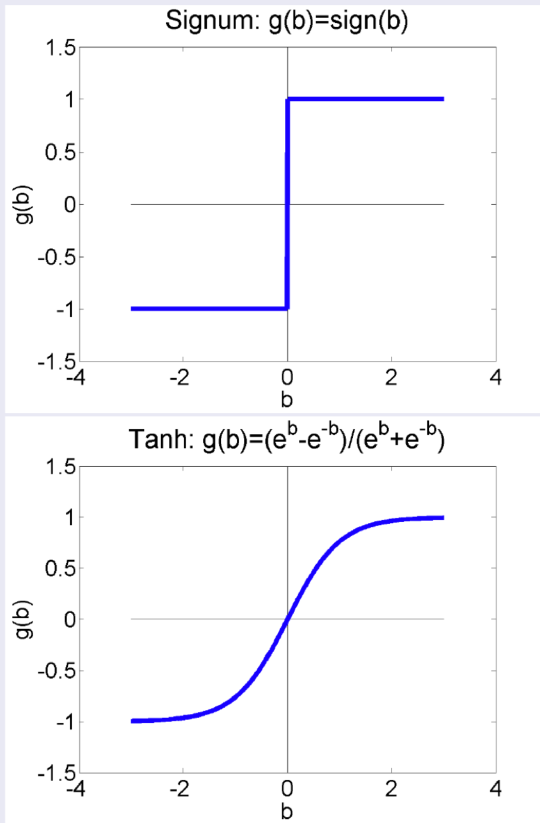
- Unfortunately, we need to train the neural net using gradient descent, and none of those nonlinearities are differentiable!!!

Differentiable approximations to non-differentiable nonlinearities

Step and Logistic nonlinearities

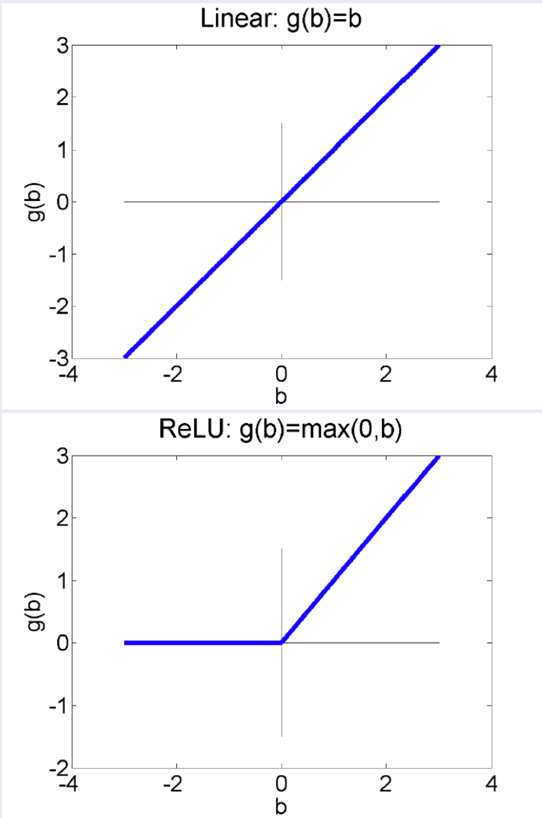


Signum and Tanh nonlinearities



Differentiable approximations to non-differentiable nonlinearities

"Linear Nonlinearity" and ReLU



Max and Softmax

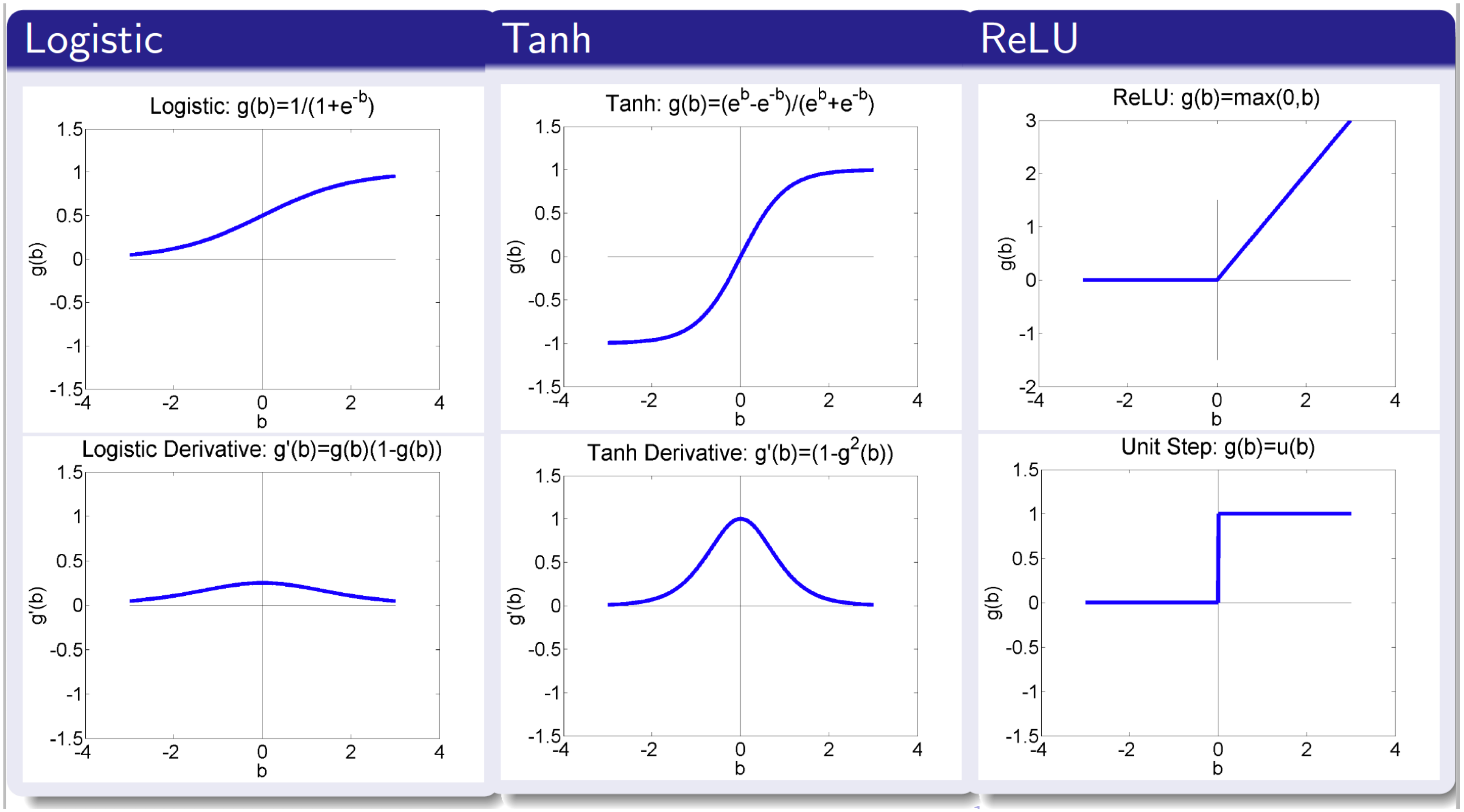
Max:

$$z_\ell = \begin{cases} 1 & b_\ell = \max_m b_m \\ 0 & \text{otherwise} \end{cases}$$

Softmax:

$$z_\ell = \frac{e^{b_\ell}}{\sum_m e^{b_m}}$$

…and their derivatives



Outline

- One-layer neural net
- Multi-layer neural net
- Training a neural net
- On-line Q-learning
- Policy learning
- Imitation learning

Multi-layer neural network

- A multi-layer neural net is parameterized by a series of weight matrices, and a series of offset vectors, one for each layer.

$$W^l = \begin{bmatrix} w_{11}^l & \dots & w_{1p}^l \\ \vdots & \vdots & \vdots \\ w_{p1}^l & \dots & w_{pp}^l \end{bmatrix}, \vec{b}^l = \begin{bmatrix} b_1^l \\ \vdots \\ b_q^l \end{bmatrix}$$

- Here l is the layer number; if the neural net has L layers, that means that $1 \leq l \leq L$.
- Each w_{kj}^l and each b_k^l is a ***<u>different</u>*** trainable parameter, so there are a total of Lp(p+1) trainable parameters!!!

Forward propagation

- Input: $\vec{x}^0$ are the input features.
- For each layer, $1 \leq l \leq L$:
  - Given an input vector $\vec{x}^{l-1}$, first, compute an affine transform using parameters W^l and $\vec{b}^l$:

$$\vec{a}^l = W^l \vec{x}^{l-1} + \vec{b}^l$$

  - Second, compute element-wise nonlinearity:

$$\vec{x}^l = g(\vec{a}^l)$$

- Output: $\vec{x}^L$ is the output of the neural net.

Outline

- One-layer neural net
- Multi-layer neural net
- Training a neural net
- On-line Q-learning
- Policy learning
- Imitation learning

Training a neural net

- Suppose we have a whole bunch of training examples $(\vec{x}_i, \vec{y}_i)$,

$$\vec{x}_i = \begin{bmatrix} x_{i1} \\ \vdots \\ x_{ip} \end{bmatrix}, \qquad \vec{y}_i = \begin{bmatrix} y_{i1} \\ \vdots \\ y_{iq} \end{bmatrix}$$

- The goal of training is to find a set of parameters $\mathrm{W} = \{W^1, \ldots, W^L\}$ and $\mathrm{B} = \{\vec{b}^1, \ldots, \vec{b}^L\}$ in order to minimize

$$E = \sum_{i=1}^{n} \sum_{k=1}^{q} (y_{ik} - f_k(\vec{x}_i; W, B))^2$$

Where, by $f_k(\vec{x}_i; W, B)$, we mean the k'th component of the output of the neural net.

Training a neural net: Gradient descent

- We train the network using gradient descent:

$$w_{kj}^{l} \longleftarrow w_{kj}^{l} - \eta \frac{\partial E}{\partial w_{kj}^{l}}$$

- That means that we need to calculate

$$\frac{\partial E}{\partial w_{kj}^{l}} = \frac{\partial \sum_{i=1}^{n} \sum_{k=1}^{q} (y_{ik} - f_k(\vec{x}_i; W, B))^2}{\partial w_{kj}^{l}}$$

for every layer l, for every weight.

Training a neural net: Chain rule

- Remember what the output of the neural net is; it's just a series of affine transforms and scalar nonlinearities. Let's use the abbreviation $f_{ik} = f_k(\vec{x}_i; W, B)$. Remember that it's given by

$$f_{ik} = x_{ik}^L = g\left(a_{ik}^L\right) = \mathrm{g}\left(b_k^L + \sum w_{kj}^L x_{ij}^{L-1}\right) = \mathrm{g}\left(b_k^L + \sum w_{kj}^L g\left(a_{ij}^{L-1}\right)\right) = \cdots$$

- So we can solve the derivative using the chain rule!!

$$\frac{\partial \sum_{i=1}^{n} \sum_{k=1}^{q} (y_{ik} - f_{ik})^2}{\partial w_{kj}^l} = 2 \sum_{i=1}^{n} \sum_{k=1}^{q} \left(x_{ik}^L - y_{ik}\right) \frac{\partial x_{ik}^L}{\partial w_{kj}^l}$$

$$= 2 \sum_{i=1}^{n} \sum_{k=1}^{q} \left(x_{ik}^L - y_{ik}\right) \frac{\partial x_{ik}^L}{\partial a_{ik}^L} \frac{\partial a_{ik}^L}{\partial w_{kj}^l} = \cdots$$

Training a neural net: The chain rule

- The chain rule requires us to find, over and over again, the derivative of the output of a layer with respect to its input. But this is just a recursive function call!! Furthermore, there are only two derivatives we need to remember:

$$\left(\vec{a} = W\vec{x} + \vec{b}\right) \Longrightarrow \left(\frac{\partial a_k}{\partial x_j} = w_{kj}\right)$$

…and…

$$(\vec{x} = g(\vec{a})) \Longrightarrow \left(\frac{\partial x_k}{\partial a_k} = g'(a_k)\right)$$

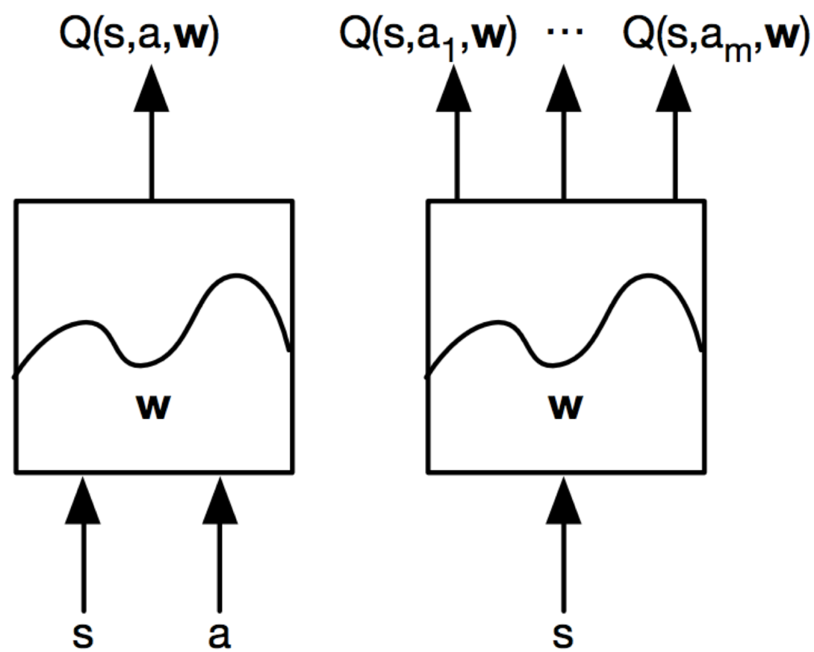
Those two derivatives get iterated, over and over again, backward through the network. This is called ***back-propagation***.

Outline

- One-layer neural net
- Multi-layer neural net
- Training a neural net
- On-line Q-learning
- Policy learning
- Imitation learning

Deep Q learning

- Train a deep neural network to output Q values:



Source: D. Silver

Deep Q learning

- SARSA update: "nudge" Q(s,a) toward value we observe it to have in the most recent action:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(\boxed{R(s) + \gamma \max_{a'} Q(s',a')} - Q(s,a)\right)$$

- Deep Q learning: encourage estimate to match the target by minimizing squared error:

$$L(w) = \left(\underbrace{\boxed{R(s) + \gamma \max_{a'} Q(s',a';w)}}_{\text{target}} - \underbrace{\boxed{Q(s,a;w)}}_{\text{estimate}}\right)^2$$

Deep Q learning

- Regular TD update: "nudge" Q(s,a) towards the target

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(\boxed{R(s) + \gamma \max_{a'} Q(s',a')} - Q(s,a)\right)$$

- Deep Q learning: encourage estimate to match the target by minimizing squared error:

$$L(w) = \left(\underbrace{\boxed{R(s) + \gamma \max_{a'} Q(s',a';w)}}_{\text{target}} - \underbrace{\boxed{Q(s,a;w)}}_{\text{estimate}}\right)^2$$

- Compare to supervised learning:

$$L(w) = \left(y - f(x;w)\right)^2$$

  – Key difference: the target in Q learning is also moving!

Online Q learning algorithm

- Perform action *a*, get observed tuple: *(s,a,s')*
- Observe: $Q^{local}(s,a) = R(s) + \gamma \max_{a'} Q(s', a'; W)$
- Update weights to reduce the error

$$L(W) = \left(Q^{local} - Q(s,a;W)\right)^2$$

- Gradient:

$$\nabla_W L = \left(Q(s,a;W) - Q^{local}\right)\nabla_W Q$$

- Weight update:

$$W \longleftarrow W - \eta \nabla_W L$$

- This is called *stochastic gradient descent* (SGD)
- "Stochastic" because the training sample (s,a,s') was chosen at random by our exploration function

Dealing with training instability

- Challenges
  - Target values are not fixed
  - Successive experiences are correlated and dependent on the policy
  - Policy may change rapidly with slight changes to parameters, leading to drastic change in data distribution
- Solutions
  - Freeze target Q network
  - Use *experience replay*

Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Experience replay

- At each time step:
  - Take action a_t according to epsilon-greedy policy
  - Store experience (s_t, a_t, r_{t+1}, s_{t+1}) in *replay memory buffer*
  - Randomly sample *mini-batch* of experiences from the buffer

$s_t, a_t, r_{t+1}, s_{t+1} \quad \rightarrow$

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\ldots$
$s_t, a_t, r_{t+1}, s_{t+1}$

Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Experience replay

At each time step:

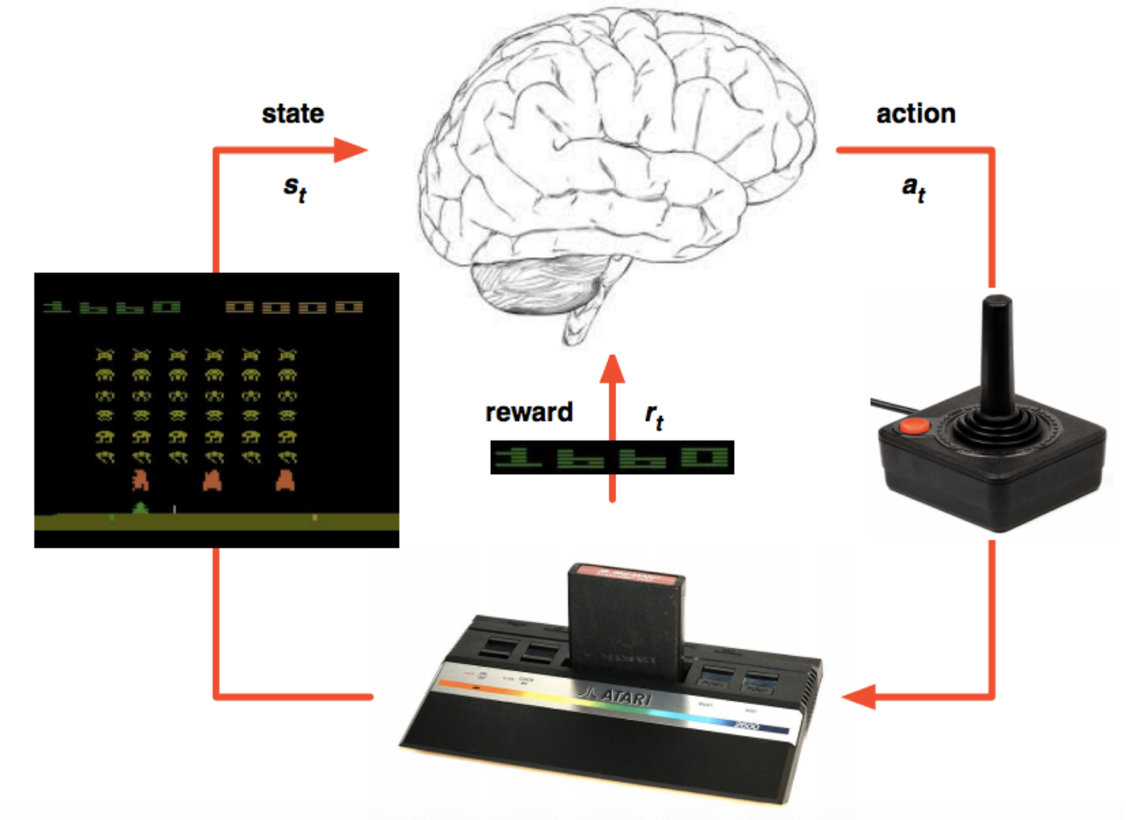
- Take action a_t according to epsilon-greedy policy
- Store experience (s_t, a_t, r_{t+1}, s_{t+1}) in *replay memory buffer*
- Randomly sample *mini-batch* of experiences from the buffer
- Perform update to reduce objective function

$$\mathbf{E}_{s,a,s'}\left[\left(R(s)+\gamma \max_{a'} \boxed{Q(s',a';w^{-})} - Q(s,a;w)\right)^2\right]$$

Keep parameters of *target network* fixed during the entire mini-batch; only update between mini-batches

Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

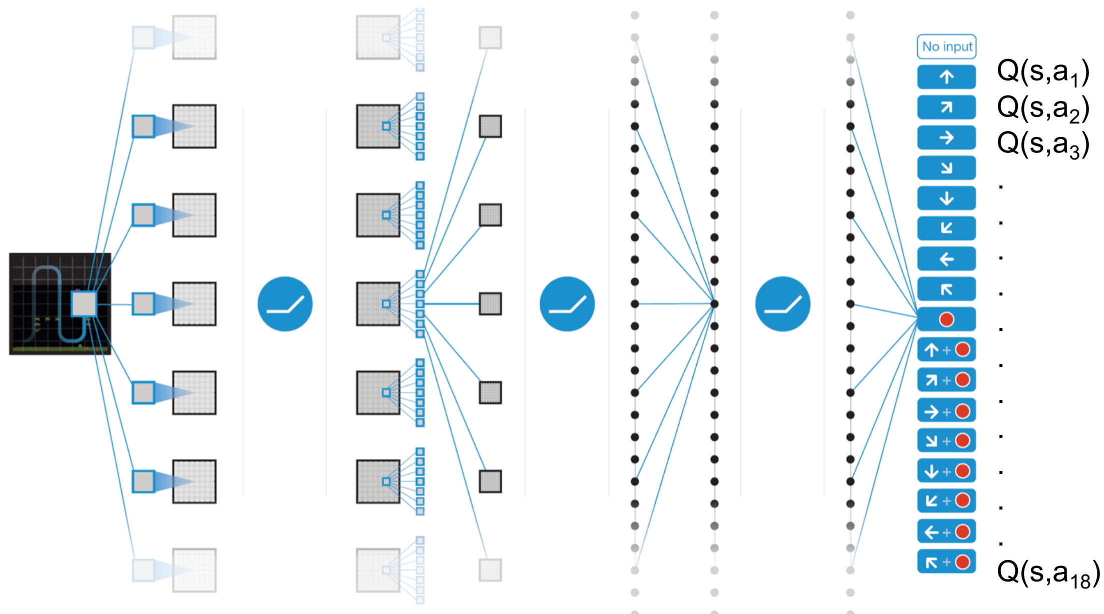
Deep Q learning in Atari



Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Deep Q learning in Atari

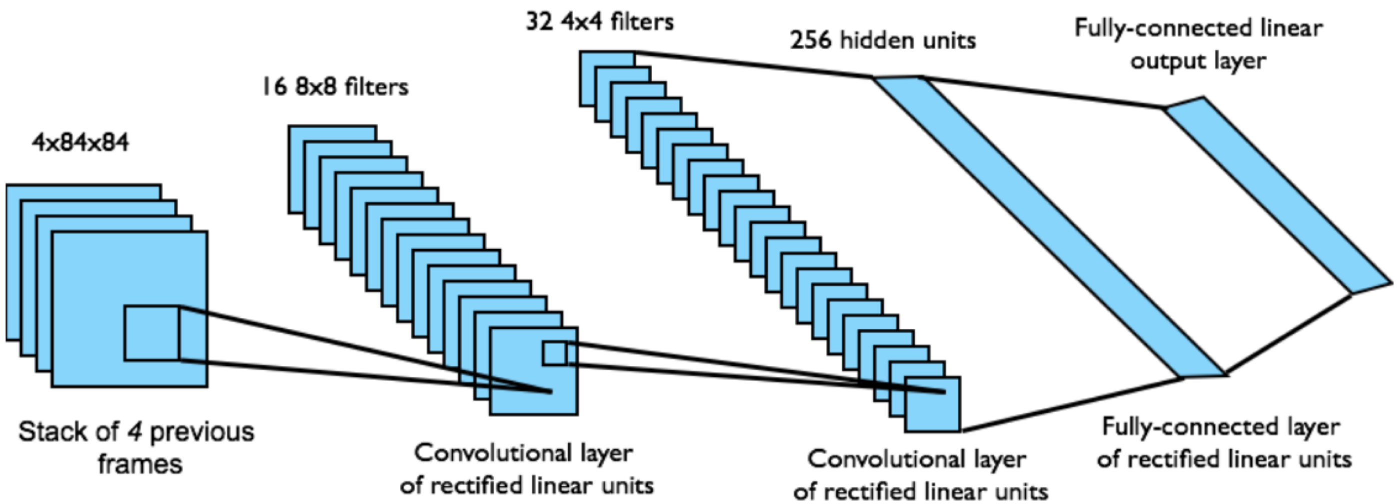
- End-to-end learning of Q(s,a) from pixels s
- Output is Q(s,a) for 18 joystick/button configurations
- Reward is change in score for that step



Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Deep Q learning in Atari

- Input state s is stack of raw pixels from last 4 frames
- Network architecture and hyperparameters fixed for all games



Mnih et al. Human-level control through deep reinforcement learning, *Nature* 2015

Outline

- One-layer neural net
- Multi-layer neural net
- Training a neural net
- On-line Q-learning
- Policy learning
- Imitation learning

Policy gradient methods

- Learning the policy directly can be much simpler than learning Q values
- We can train a neural network to output *stochastic policies*, or probabilities of taking each action in a given state
- *Softmax* policy:

$$\pi(s,a;u) = \frac{\exp\big(f(s,a;u)\big)}{\sum_{a'} \exp\big(f(s,a';u)\big)}$$

Policy gradient: the softmax function

- Notice that the softmax is normalized so that

$$\pi(s, a; u) \geq 0, \text{ and } \sum_a \pi(s, a; u) = 1$$

- So we can interpret $\pi(s, a; w)$ as some kind of probability. Something like "the probability that a is the best action to take from state s."
- In reality, there is no such probability. There is just one correct action. But the agent doesn't know what it is! So $\pi(s, a; u)$ is kind of like the agent's "degree of belief" that a is the best action (determined by parameters u).

Actor-critic algorithm

- Remember the relationship between the utility of a state, and the quality of an action:

$$U(s) = \max_a Q(s,a)$$

- If we don't know which action is best, then we could say that

$$U(s) \approx \sum_a \pi(s,a;u)Q(s,a;w)$$

- $\pi(s,a;u)$ is the "actor:" a neural net that tells the agent how to act.
- $Q(s,a;w)$ is the "critic:" a neural net that tells the agent how good or bad that action was.

Actor-critic algorithm

- Define objective function as total discounted reward:

$$J(u) = \mathbf{E}\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots\right]$$

- The gradient for a stochastic policy is given by

$$\nabla_u J = \mathbf{E}\left[\nabla_u \log \boxed{\pi(s,a;u)}\, \boxed{Q^\pi(s,a;w)}\right]$$

Actor network

Critic network

- Actor network update: $u \leftarrow u + \alpha \nabla_u J$
- Critic network update: use Q learning (following actor's policy)

Advantage actor-critic

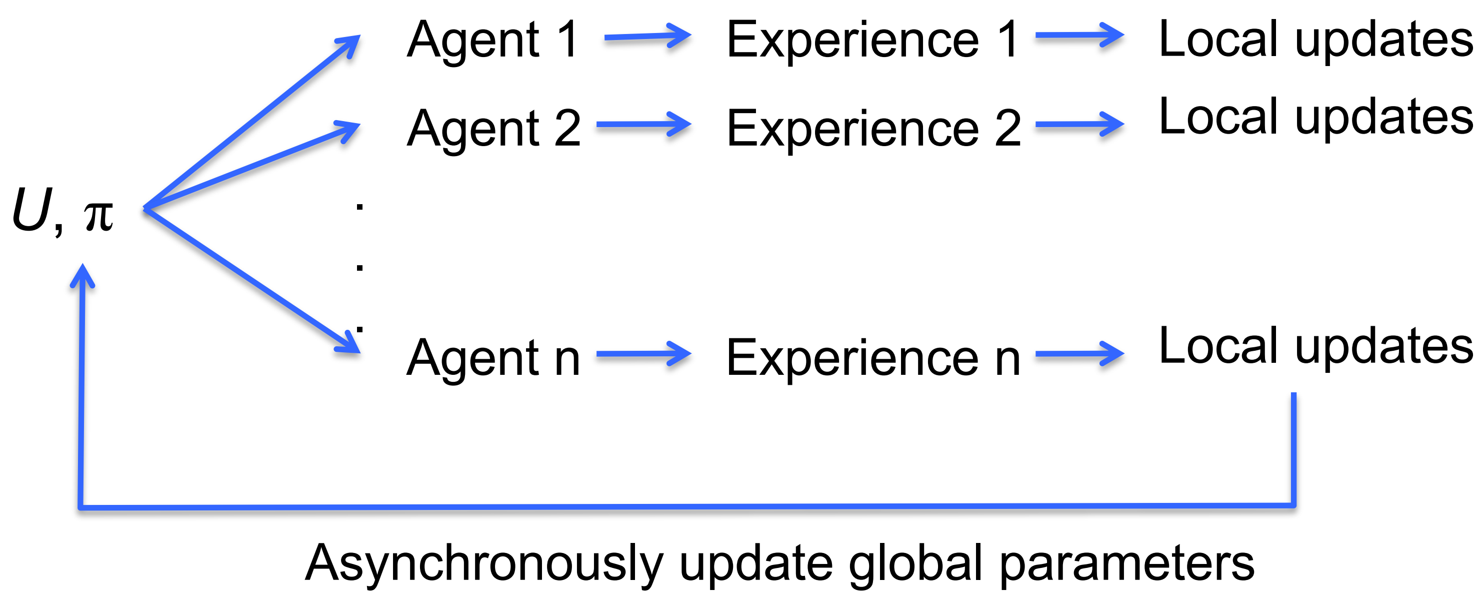
- The raw Q value is less meaningful than whether the reward is better or worse than what you expect to get
- Introduce an *advantage function* that subtracts a baseline number from all Q values

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

  - Estimate *V* using a *value network*
- Advantage actor-critic:

$$\nabla_u J = \mathbf{E}\left[\nabla_u \log \pi(s,a;u)\, A^{\pi}(s,a;w)\right]$$

Asynchronous advantage actor-critic (A3C)



Mnih et al. Asynchronous Methods for Deep Reinforcement Learning. ICML 2016

Asynchronous advantage actor-critic (A3C)

[TORCS car racing simulation video]

Mnih et al. Asynchronous Methods for Deep Reinforcement Learning. ICML 2016

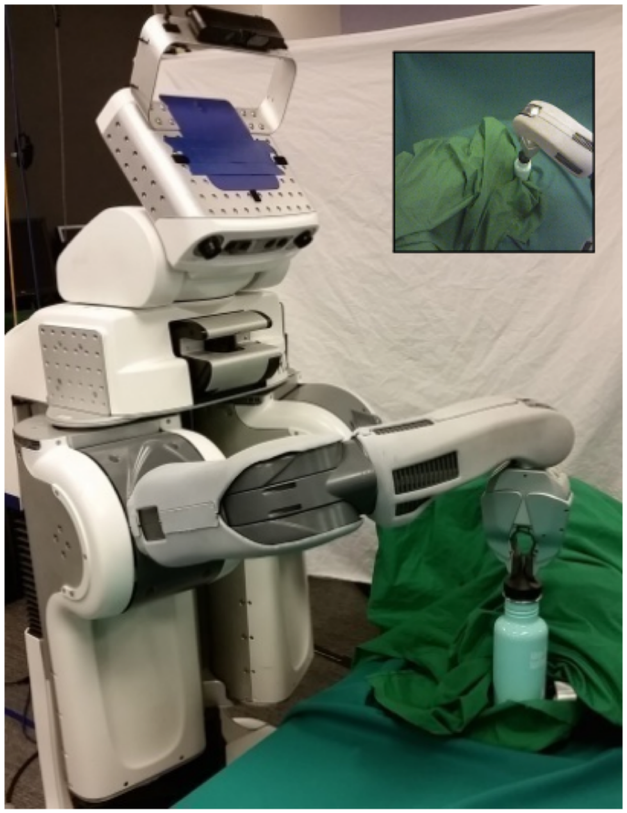
Outline

- One-layer neural net
- Multi-layer neural net
- Training a neural net
- On-line Q-learning
- Policy learning
- Imitation learning

Imitation learning

- In some applications, you cannot bootstrap yourself from random policies
  - High-dimensional state and action spaces where most random trajectories fail miserably
  - Expensive to evaluate policies in the physical world, especially in cases of failure
- **Solution:** learn to imitate sample trajectories or demonstrations
  - This is also helpful when there is no natural reward formulation

Learning visuomotor policies

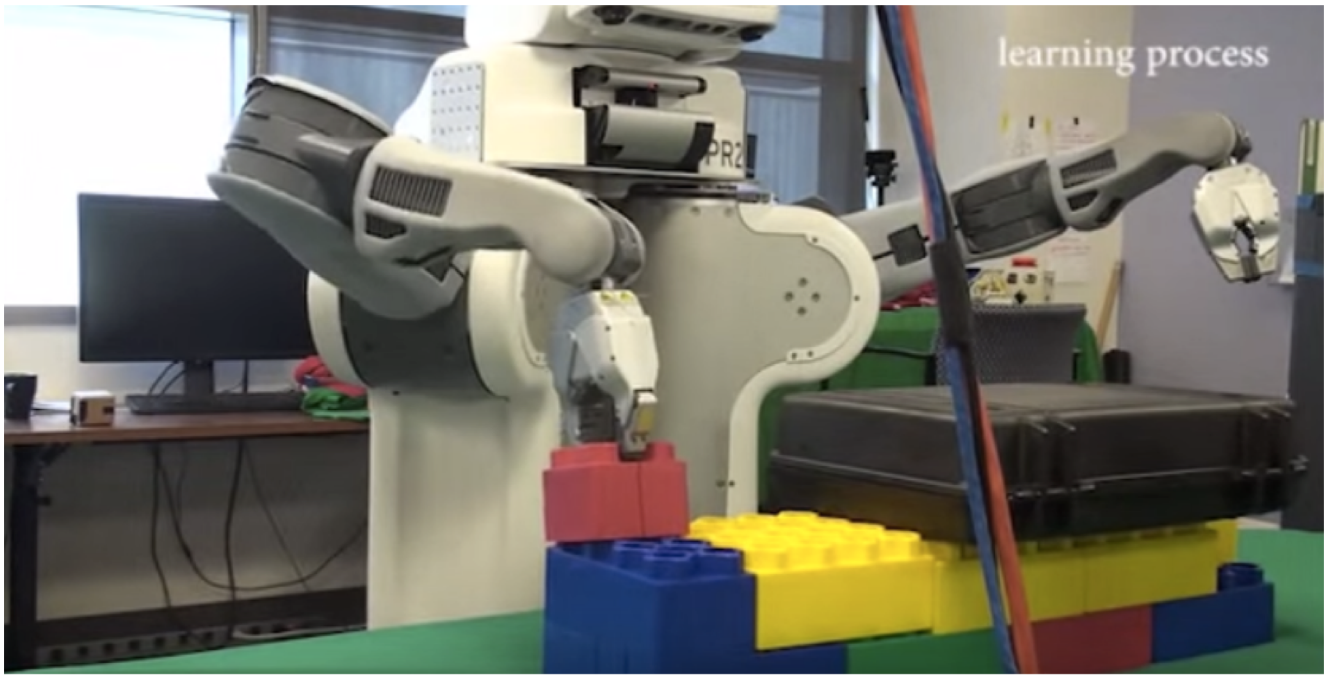


- *Underlying state* x: true object position, robot configuration
- *Observations* o: image pixels

- Two-part approach:
  – Learn *guiding policy* π(a|x) using trajectory-centric RL and control techniques
  – Learn *visuomotor policy* π(a|o) by imitating π(a|x)

S. Levine et al. End-to-end training of deep visuomotor policies. JMLR 2016

Learning visuomotor policies



Overview video, training video

S. Levine et al. End-to-end training of deep visuomotor policies. JMLR 2016