

VALUE

FUNCTION

APPROXIMATION

VALUE FUNCTION APPROXIMATION

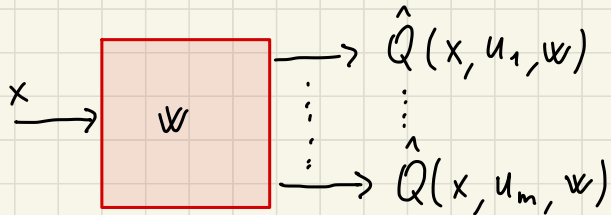
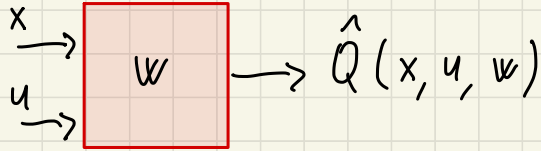
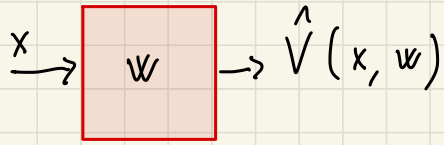
- Discrete-space algorithms don't scale to large systems:
 - Backgammon: 10^{20} states
 - Go: 10^{170} states
 - Robots: continuous state space
- Too many states/control to store and learn
- IDEA: Use function approximation to represent $V(x)/Q(x,u)$

$$\hat{V}(x, w) \approx V(x)$$

$$\hat{Q}(x, u, w) \approx Q(x, u)$$

- w = parameters defining the function

TYPES OF VALUE FUNCTION APPROXIMATION

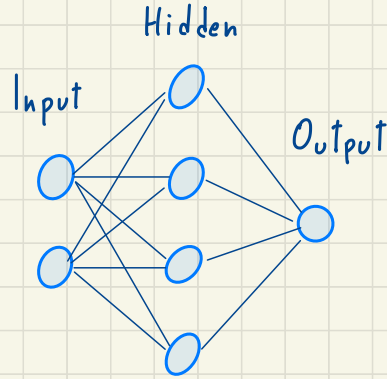


- **Neural networks** are often used as function approximation
- Any **differentiable** approximator can be used:
 - linear combination of **features**
 - Fourier basis
 - Polynomials
- Problems:
 - non-stationary target V
 - non i.i.d. data

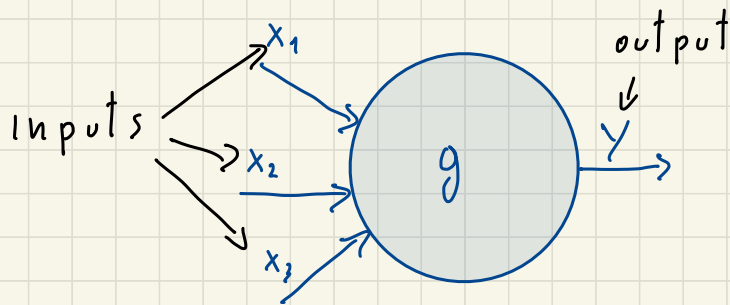
Arbitrary
user-defined
functions of x

NEURAL NETWORKS

- Parametric function structured in "layers":
 - 1 input layer (size = # of inputs)
 - ≥ 1 hidden layers
 - 1 output layer (size = # of outputs)
- Each layer is composed of "neurons"
- In fully-connected networks each neurons receives as input the outputs of all neurons of the previous layer
- A neuron outputs a nonlinear function of a linear combination of its inputs

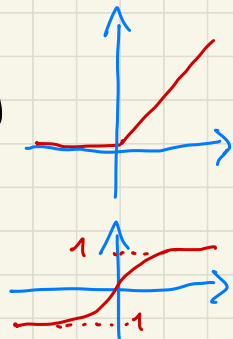


NEURONS



$$y = g\left(\sum_i w_i x_i + w_0\right)$$

- $g(\cdot)$ is the nonlinear "activation" function, e.g. :
 - ReLU, Rectified Linear Unit : $g(p) = \max(0, p)$
 - Hyperbolic tangent : $g(p) = \tanh(p) = \frac{e^p - e^{-p}}{e^p + e^{-p}}$
- w is the vector of weights associated to the inputs x
- w_0 is called the "bias"



TRAINING A NEURAL NETWORK

- Training = Finding the values of all weights w
- Typically formulated as unconstrained optimization:

$$\underset{w}{\text{minimize}} \quad \sum_i \|\phi(x_i; w) - y_i\|^2 \triangleq J(w)$$

← LOSS FUNCTION

- Given a list of input-output pairs (x_i, y_i) , find w such that the network's output $\phi(x; w)$ is close to y .
- Since $\text{size}(w)$ can be very large, typically solved with some variants of gradient descent:

$$w := w - \alpha \nabla J(w)$$

α = step size

STOCHASTIC GRADIENT DESCENT

- If the number of input-output pairs is large, computing ∇J can be computationally expensive

$$\nabla_{\mathbf{w}} J = \sum_{i \in \mathcal{D}} 2 (\phi(x_i; \mathbf{w}) - y_i) \nabla_{\mathbf{w}} \phi(x_i; \mathbf{w})$$

- IDEA: Compute ∇J using only a subset of the data
- In the limit the subset (minibatch) can be just one sample
- Gradient is approximated (i.e. noisy) so:
 - more iterations needed to converge
 - each iteration is much cheaper to compute

INCREMENTAL PREDICTION ALGORITHMS

- How can we get target $y = V(x)$?

- For MC, target is cost-to-go $J_t(x)$:

$$\Delta w = \alpha (J_t - \phi(x_t, w)) \nabla_w \phi(x_t, w)$$

- MC converges to local optimum even with nonlinear approximators

- For TD(0), target is TD target:

$$\Delta w = \alpha (r_t + \gamma \phi(x_{t+1}, w) - \phi(x_t, w)) \nabla_w \phi(x_t, w)$$

- TD converges (close) to global optimum with linear approxim.

INCREMENTAL CONTROL WITH FUNCTION APPROXIM.

IDEA: Generalized Policy Iteration with approximate Action-Value function:

$$\min_w E_{\pi}[(\phi(x, u, w) - Q_{\pi}(x, u))^2]$$

- Use SGD to find local minimum
 - For MC, target is cost-to-go $J_T(x_t, u_t)$
 - For TD(0), target is $l_t(x_t, u_t) + \gamma \phi(x_{t+1}, u_{t+1}, w)$
- Use ϵ -greedy policy to ensure exploration
 - Greedy control is easily computed for discrete control space

CONVERGENCE OF CONTROL ALGORITHMS

	VALUE ENCODING		
	Table Lookup	Linear	Nonlinear
MC control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-Learning	✓	✗	✗

(✓) = chatter around optimum

Linear $\Rightarrow \phi(x, w) = h(x)^T w$ $h(x) = \text{nonlinear function}$

LIMITATIONS OF INCREMENTAL CONTROL

- Each transition (x, u, l, x') is used to perform a single gradient step on w and then thrown away
=> POOR DATA EFFICIENCY
- Consecutive transitions are strongly correlated because the state evolves continuously
=> DATA IS NOT INDEPENDENTLY & IDENTICALLY DISTRIBUTED

Can we overcome these issues?

BATCH LEARNING

- IDEA: Instead of learning incrementally, learn after having collected a batch of training data

- Least-squares prediction:

$$\min_w \sum_{t=1}^T (V_t - \phi(x_t, w))^2$$

- How to achieve it? Experience replay:
 - store data $\langle x_t, V_t \rangle$ in replay buffer
 - sample minibatch from buffer
 - apply SGD to update w
 - converge to LS solution

DEEP Q-NETWORK (2015, Mnih et al.)

- Extension of Q-Learning to continuous state space
- Discrete control space
- Use neural network to represent $Q(x, u, w)$
- ϵ -greedy policy with ϵ decreasing over time
- Use experience replay:
 - Store transitions (x_t, u_t, l_t, x_{t+1}) in replay buffer
 - Every few steps (e.g. 4) sampled randomly from replay buffer a mini-batch of transitions (size 32)
 - Use stochastic gradient descent to optimize w

DQN uses fixed Q targets to stabilize training:

- compute Q targets using old fixed parameters w^-
- optimize MSE btw Q-network and Q-learning targets:

$$\underset{w}{\text{minimize}} \quad \mathbb{E} \left[l + \gamma \min_{u'} \phi(x', u', w^-) - \phi(x, u, w) \right]$$

- every C steps update target weights:

$$w^- := w$$

- Store w resulting in best performance as performance may get worse during training

DQN on ATARI

<https://www.youtube.com/watch?v=W2CAghUiofY>

- 49 ATARI video games (e.g. Pong, Space Invaders)
- Policy maps pixels to actions
- Cost = negative change in score
- Same algorithm, hyperparameters on all games
- Reached human-level performance on 29 out of 49 games
- Convolutional neural network:
 - 3 million parameters
 - History of last 4 frames as input
 - 2 convolutional layers + 1 fully-connected layer
 - ReLU activation functions

DQN DETAILS

- Use **Huber loss** instead of squared loss on TD error:

$$L_{\delta}(e) = \begin{cases} \frac{1}{2} e^2 & \text{if } |e| \leq \delta \quad \Leftarrow \text{Quadratic around zero} \\ \delta(|e| - \frac{1}{2}\delta) & \text{otherwise} \quad \Leftarrow \text{Linear far from zero} \end{cases}$$

- Use **RMS Prop** instead of standard SGD
- Start ϵ from 1, anneal it to 0.1/0.05 over the first million samples

CONCLUSIONS

- Introduce function approximation to scale RL algorithms
- Use stochastic gradient descent for training neural networks
- Incremental alg. update w at every step with current transition
- Batch alg. introduce replay buffer to improve sample efficiency and break data dependency
- Action space is still assumed to be discrete
- Can replace state with sensor data w/o changing algorithms