

Topics in Reinforcement Learning: Rollout and Approximate Policy Iteration

ASU, CSE 691, Spring 2021

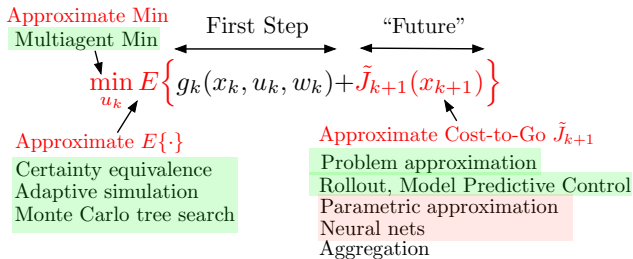
Links to Class Notes, Videolectures, and Slides at
<http://web.mit.edu/dimitrib/www/RLbook.html>

Dimitri P. Bertsekas
dbertsek@asu.edu

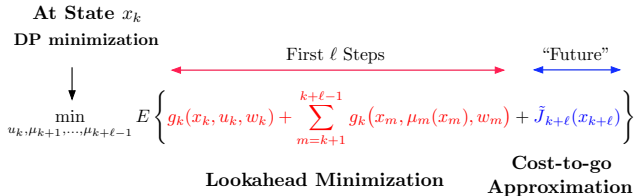
Lecture 8
Off-Line Training, Neural Nets, and Other Parametric Architectures

- 1 Parametric Approximation Architectures
- 2 Training of Architectures - Switch to Video-Lecture 6 from 2019
- 3 Incremental Optimization of Sums of Differentiable Functions
- 4 Neural Networks
- 5 Neural Nets and Finite Horizon DP
- 6 Back from Video-Lecture 6 from 2019 - Miscellaneous Issues in Training

Recall Approximation in Value Space (Mostly Used for On-Line Control Selection)



ONE-STEP LOOKAHEAD



MULTISTEP LOOKAHEAD

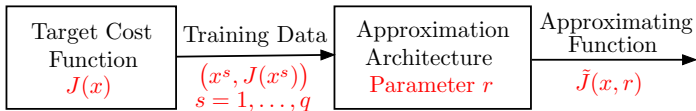
There are two types of off-line approximations in RL:

- **Cost approximation** in finite and infinite horizon problems
 - ▶ Optimal cost function $J_k^*(x_k)$ or $J^*(x)$, optimal Q-function $Q_k^*(x_k, u_k)$ or $Q^*(x, u)$
 - ▶ Cost function of a policy $J_{\pi,k}(x_k)$ or $J_{\mu}(x)$, Q-function of a policy $Q_{\pi,k}(x_k, u_k)$ or $Q_{\mu}(x, u)$
- **Policy approximation** in finite and infinite horizon problems
 - ▶ Optimal policy $\mu_k^*(x_k)$ or $\mu^*(x)$
 - ▶ A given policy $\mu_k(x_k)$ or $\mu(x)$

We will focus on **parametric** approximations $\tilde{J}(x, r)$ and $\tilde{\mu}(x, r)$

- These are functions of x that depend on a parameter vector r
- An example is neural networks (r is the set of weights)

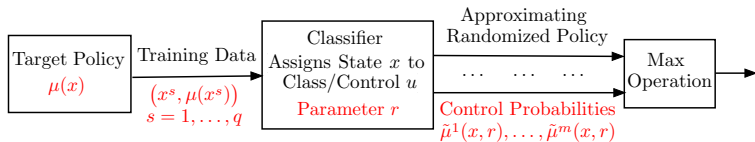
General Parametric Cost Approximation



TRAINING CAN BE DONE WITH SPECIALIZED OPTIMIZATION SOFTWARE
SUCH AS
GRADIENT-LIKE METHODS OR OTHER LEAST SQUARES METHODS

Parametric Policy Approximation - Finite Control Space

- If the control has continuous/real-valued components, the training is similar to the cost function case
- If the control comes from a finite control space $\{u^1, \dots, u^m\}$, a modified approach is needed
- View a policy μ as a **classifier**: A function that maps x into a “category” $\mu(x)$



TRAINING CAN BE DONE WITH CLASSIFICATION SOFTWARE
IF THE NUMBER OF CONTROLS IS FINITE

Randomized policies have continuous components
This helps algorithmically

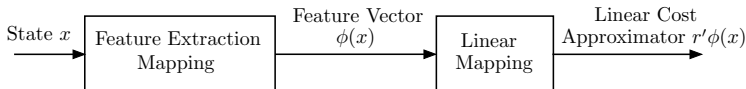
Cost Function Parametric Approximation Generalities

- We select a class of functions $\tilde{J}(x, r)$ that depend on x and a **vector** $r = (r_1, \dots, r_m)$ of m “tunable” scalar parameters.
- We adjust r to change \tilde{J} and “match” the training data from the target function.
- **Training the architecture**: The algorithm to choose r (typically **regression-type**).
- **Local vs global**: Change in a single parameter affects \tilde{J} locally vs globally.
- Architectures are called **linear or nonlinear**, if $\tilde{J}(x, r)$ is linear or nonlinear in r .
- Architectures are **feature-based** if they depend on x via a feature vector $\phi(x)$ that captures “major characteristics” of x ,

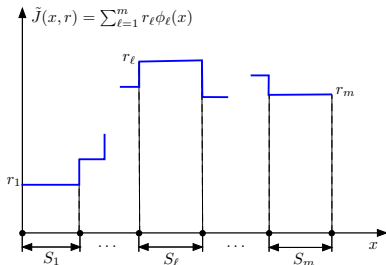
$$\tilde{J}(x, r) = \hat{J}(\phi(x), r),$$

where \hat{J} is some function. Intuitive idea: **Features capture dominant nonlinearities.**

- A **linear feature-based architecture**: $\tilde{J}(x, r) = \sum_{\ell=1}^m r_{\ell} \phi_{\ell}(x) = r' \phi(x)$, where r_{ℓ} and $\phi_{\ell}(x)$ are the ℓ th components of r and $\phi(x)$.



A Simple Example of a Linear Feature-Based (Local) Architecture



Piecewise constant approximation

- Partition the state space into subsets S_1, \dots, S_m . Let the ℓ th feature be defined by membership in the set S_{ℓ} , i.e., **the indicator function of S_{ℓ}** ,

$$\phi_{\ell}(x) = \begin{cases} 1 & \text{if } x \in S_{\ell} \\ 0 & \text{if } x \notin S_{\ell} \end{cases}$$

- The architecture

$$\tilde{J}(x, r) = \sum_{\ell=1}^m r_{\ell} \phi_{\ell}(x),$$

is piecewise constant with value r_{ℓ} for all x within the set S_{ℓ} .

Quadratic polynomial approximation

- Let $x = (x^1, \dots, x^n)$
- Consider features

$$\phi_0(x) = 1, \quad \phi_i(x) = x^i, \quad \phi_{ij}(x) = x^i x^j, \quad i, j = 1, \dots, n,$$

and the linear feature-based approximation architecture

$$\tilde{J}(x, r) = r_0 + \sum_{i=1}^n r_i x^i + \sum_{i=1}^n \sum_{j=i}^n r_{ij} x^i x^j$$

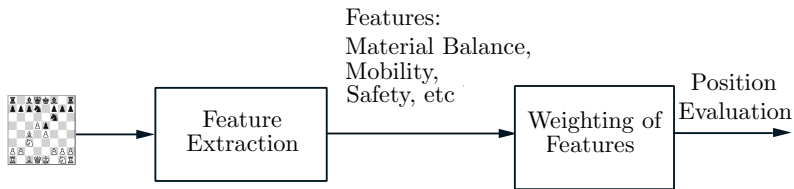
- Here the parameter vector r has components r_0 , r_i , and r_{ij} .

General polynomial architectures: Polynomials in the components x^1, \dots, x^n

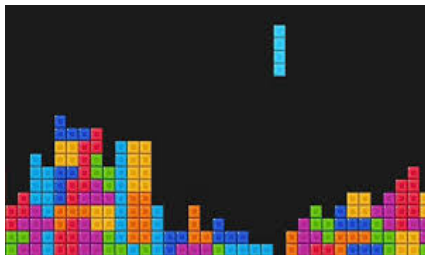
An even more general architecture: Polynomials of features of x

A linear feature-based architecture is a special case

Examples of Problem-Specific Feature-Based Architectures

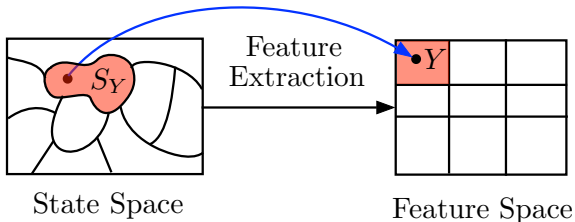


Chess



Tetris

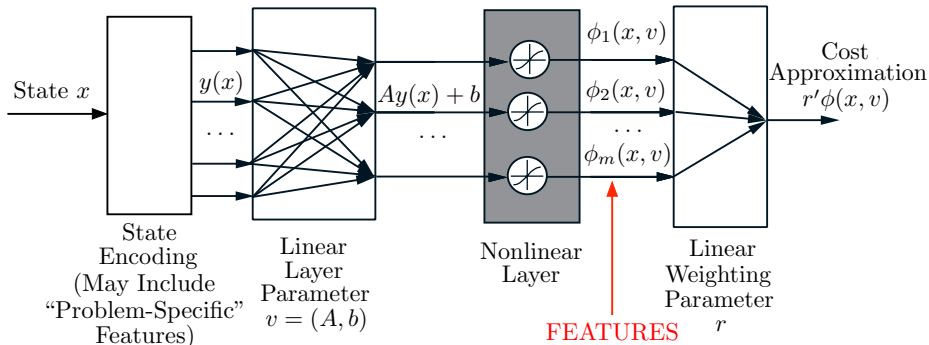
Architectures with Partitioned State Space



A simple method to construct complex approximation architectures:

- Partition the state space into several subsets and **construct a separate cost approximation in each subset**.
- Can use a **separate architecture on each set** of the partition.
- It is often a good idea to **use features to generate the partition**. Rationale:
 - ▶ We want to group together states with similar costs
 - ▶ We hypothesize that states with similar features should have similar costs

Neural Networks: An Architecture that Works with No Knowledge of Features



A SINGLE LAYER NEURAL NETWORK

Least squares regression

- Collect a set of state-cost training pairs (x^s, β^s) , $s = 1, \dots, q$, where β^s is equal to the target cost $J(x^s)$ plus some “noise”.
- r is determined by solving the problem

$$\min_r \sum_{s=1}^q (\tilde{J}(x^s, r) - \beta^s)^2$$

- Sometimes a quadratic regularization term $\gamma \|r\|^2$ is added to the least squares objective, to facilitate the minimization (among other reasons).

Training of linear feature-based architectures can be done exactly

- If $\tilde{J}(x, r) = r' \phi(x)$, where $\phi(x)$ is the m -dimensional feature vector, the training problem is quadratic and can be solved in closed form.
- The exact solution of the training problem is given by

$$\hat{r} = \left(\sum_{s=1}^q \phi(x^s) \phi(x^s)' \right)^{-1} \sum_{s=1}^q \phi(x^s) \beta^s$$

- This requires a lot of computation for a large m and data set; may not be best.

The main training issue

How to exploit the structure of the training problem

$$\min_r \sum_{s=1}^q (\tilde{J}(x^s, r) - \beta^s)^2$$

to solve it efficiently.

Key characteristics of the training problem

- **Possibly nonconvex with many local minima**, horribly complicated graph of the cost function (true when a neural net is used).
- **Many terms in the least squares sum**; standard gradient and Newton-like methods are essentially inapplicable.
- **Incremental** iterative methods that operate on **a single term** $(\tilde{J}(x^s, r) - \beta^s)^2$ **at each iteration** have worked well enough (for many problems).

Generic sum of terms optimization problem

Minimize

$$f(y) = \sum_{i=1}^m f_i(y)$$

where each f_i is a differentiable scalar function of the n -dimensional vector y (this is the parameter vector in the context of parametric training).

The ordinary gradient method generates y^{k+1} from y^k according to

$$y^{k+1} = y^k - \gamma^k \nabla f(y^k) = y^k - \gamma^k \sum_{i=1}^m \nabla f_i(y^k)$$

where $\gamma^k > 0$ is a stepsize parameter.

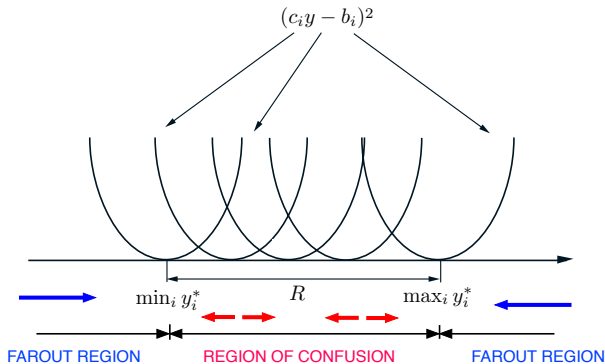
The incremental gradient counterpart

Choose an index i_k and iterate according to

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k)$$

where $\gamma^k > 0$ is a stepsize parameter.

The Advantage of Incrementalism: An Interpretation from the NDP Book



$$\text{Minimize } f(y) = \frac{1}{2} \sum_{i=1}^m (c_i y - b_i)^2$$

Compare the ordinary and the incremental gradient methods in two cases

- When far from convergence: **Incremental gradient is as fast as ordinary gradient with $1/m$ amount of work.**
- When close to convergence: **Incremental gradient gets confused** and requires a diminishing stepsize for convergence.

Incremental **aggregated** method aims at acceleration

- Evaluates gradient of a single term at each iteration.
- Uses previously calculated gradients as if they were up to date

$$y^{k+1} = y^k - \gamma^k \sum_{\ell=0}^{m-1} \nabla f_{i_{k-\ell}}(y^{k-\ell})$$

- Has theoretical and empirical support, and it is often preferable.

Stochastic gradient method (also called stochastic gradient descent or **SGD**)

- Applies to **minimization** of $f(y) = E\{F(y, w)\}$ where w is a random variable
- Has the form

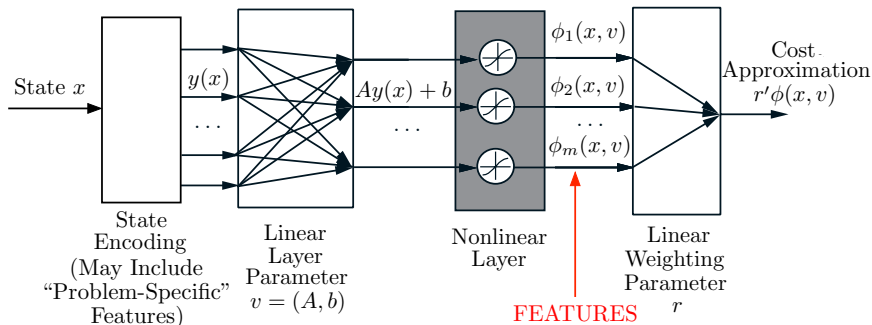
$$y^{k+1} = y^k - \gamma^k \nabla_y F(y^k, w^k)$$

where w^k is a sample of w and $\nabla_y F$ denotes gradient of F with respect to y .

- **The incremental gradient method with random index selection is the same as SGD** [convert the sum $\sum_{i=1}^m f_i(y)$ to an expected value, where i is random with uniform distribution].

- How to pick the **stepsize** γ^k (usually $\gamma^k = \frac{\gamma}{k+1}$ or similar).
- How to deal (if at all) with **region of confusion** issues (detect being in the region of confusion and reduce the stepsize).
- How to select the **order of terms to iterate** (cyclic, random, other).
- **Diagonal scaling** (a different stepsize for each component of y).
- **Alternative methods** (more ambitious): Incremental Newton method, extended Kalman filter (see the textbook and references).

Neural Nets: An Architecture that Automatically Constructs Features

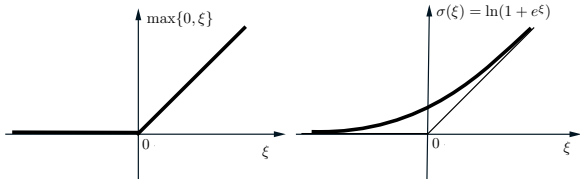


Given a set of state-cost training pairs (x^s, β^s) , $s = 1, \dots, q$, the parameters of the neural network (A, b, r) are obtained by solving the training problem

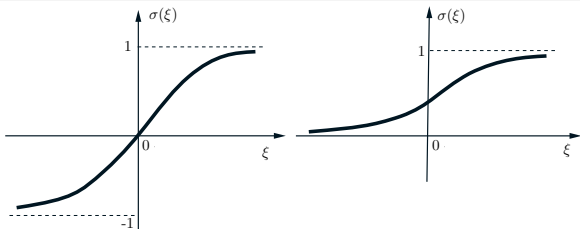
$$\min_{A, b, r} \sum_{s=1}^q \left(\sum_{\ell=1}^m r_{\ell} \sigma \left((Ay(x^s) + b)_{\ell} \right) - \beta^s \right)^2$$

- Incremental gradient is typically used for training.
- **Universal approximation property.**

Rectifier and Sigmoidal Nonlinearities

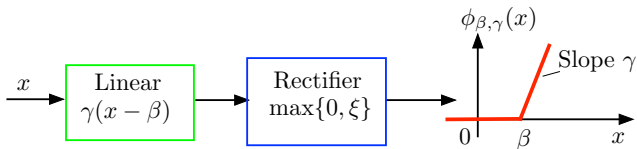


The **rectified linear unit** $\sigma(\xi) = \ln(1 + e^\xi)$. It is the rectifier function $\max\{0, \xi\}$ with its corner “smoothed out.”

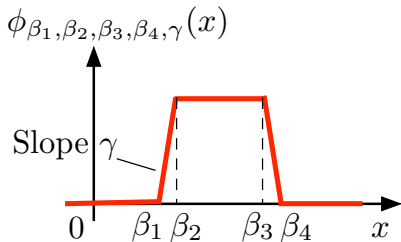


Sigmoidal units: The **hyperbolic tangent** function $\sigma(\xi) = \tanh(\xi) = \frac{e^\xi - e^{-\xi}}{e^\xi + e^{-\xi}}$ is on the left. The **logistic** function $\sigma(\xi) = \frac{1}{1 + e^{-\xi}}$ is on the right.

A Working Break: Challenge Question

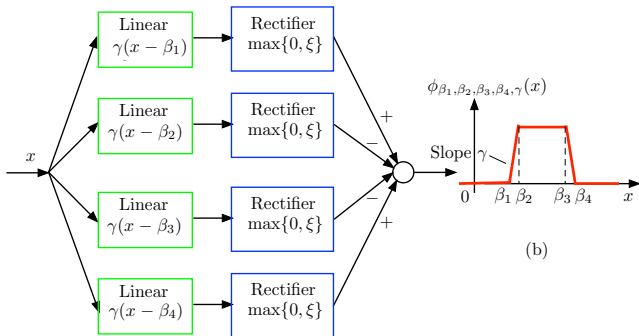
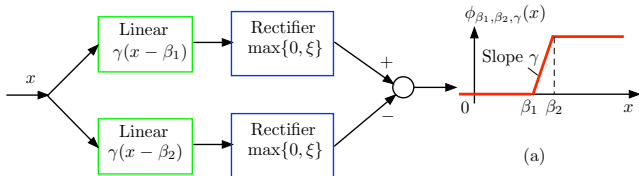


How can we use linear and rectifier units to **construct the "pulse" feature below?**



- What are the features that can be produced by neural nets?
- Why do neural nets have a "universal approximation" property?

Answer



Using the pulse feature as a building block, any feature can be approximated

Sequential DP Approximation - A Parametric Approximation at Every Stage (Also Called **Fitted Value Iteration**)

Start with $\tilde{J}_N = g_N$ and **sequentially train going backwards**, until $k = 0$

- Given a cost-to-go approximation \tilde{J}_{k+1} , we **use one-step lookahead to construct a large number of state-cost pairs** (x_k^s, β_k^s) , $s = 1, \dots, q$, where

$$\beta_k^s = \min_{u \in U_k(x_k^s)} E \left\{ g(x_k^s, u, w_k) + \tilde{J}_{k+1}(f_k(x_k^s, u, w_k), r_{k+1}) \right\}, \quad s = 1, \dots, q$$

- We “train” an architecture \tilde{J}_k on the training set (x_k^s, β_k^s) , $s = 1, \dots, q$.

Typical approach: Train by least squares/regression and possibly using a neural net

We minimize over r_k

$$\sum_{s=1}^q (\tilde{J}_k(x_k^s, r_k) - \beta_k^s)^2$$

Sequential Q-Factor Approximation

- Consider sequential DP approximation of Q-factor parametric approximations

$$\tilde{Q}_k(x_k, u_k, r_k) = E \left\{ g_k(x_k, u_k, w_k) + \min_{u \in U_{k+1}(x_{k+1})} \tilde{Q}_{k+1}(x_{k+1}, u, r_{k+1}) \right\}$$

(Note a mathematical magic: **The order of $E\{\cdot\}$ and min have been reversed.**)

- We obtain $\tilde{Q}_k(x_k, u_k, r_k)$ by training with many pairs $((x_k^s, u_k^s), \beta_k^s)$, where β_k^s is a **sample of the approximate Q-factor of (x_k^s, u_k^s)** . [No need to compute $E\{\cdot\}$.]
- Note: **No need for a model to obtain β_k^s** . Sufficient to have a simulator that generates state-control-cost-next state random samples

$$((x_k, u_k), (g_k(x_k, u_k, w_k), x_{k+1}))$$

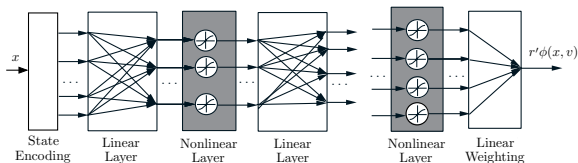
- Having computed r_k , the one-step lookahead control is obtained on-line as

$$\bar{\mu}_k(x_k) \in \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u, r_k)$$

without the need of a model or expected value calculations.

- Important advantage: The **on-line calculation of the control is simplified.**

On The Mystery of Deep Neural Networks



- Extensive research has gone into explaining why they are more effective than shallow neural nets for some problems.
- Recent research strongly suggests that **overparametrization** (many more parameters than data) is the main reason.
- Generally the ratio

$$R = \frac{\text{Number of weights}}{\text{Number of data points}}$$

affects the quality of the trained architecture.

- If $R \approx 1$, the architecture tends to fit very well the training data (**overfitting**), but do poorly at states outside the data set. This is well-known in machine learning.
- **For R considerably larger than 1 this problem can be overcome.**
- See the research literature and the recent text by Hardt and Recht, 2021, "Patterns, Predictions, and Actions", arXiv preprint arXiv:2102.05242

Should we Approximate Q-Factors or Q-Factor Differences?

To compare controls at x , we only need Q-factor differences $\tilde{Q}(x, u) - \tilde{Q}(x, u')$

An example of what can happen if we do not use Q-factor differences:

- Scalar system and cost per stage:

$$x_{k+1} = x_k + \delta u_k, \quad g(x, u) = \delta(x^2 + u^2), \quad \delta > 0 \text{ is very small;}$$

think of discretization of continuous-time problem involving $dx(t)/dt = u(t)$

- Consider policy $\mu(x) = -2x$. Its cost function can be calculated to be

$$J_\mu(x) = \frac{5x^2}{4}(1 + \delta) + O(\delta^2), \quad \text{HUGE relative to } g(x, u)$$

Its Q-factor can be calculated to be

$$Q_\mu(x, u) = \frac{5x^2}{4} + \delta \left(\frac{9x^2}{4} + u^2 + \frac{5}{2}xu \right) + O(\delta^2)$$

- The important part for policy improvement is $\delta(u^2 + \frac{5}{2}xu)$. When $Q_\mu(x, u)$ is approximated by $\tilde{Q}_\mu(x, u; r)$, it will be dominated by $5x^2/4$ and will be "lost"

A More General Issue: Disproportionate Terms in Q-Factor Calculations

Remedy: Subtract state-dependent constants from Q-factors (“**baselines**”)

The constants subtracted should affect the offending terms (such as \tilde{J})

Example: Consider rollout with cost function approximation $\tilde{J} \approx J_\mu$

- At x , we minimize over u

$$E\{g(x, u, w) + \tilde{J}(f(x, u, w))\}$$

- Question: **How to deal with $g(x, u, w)$ being tiny relative to $\tilde{J}(f(x, u, w))$?** An important case where this happens: Time discretization of continuous-time systems.
- A remedy: **Subtract $\tilde{J}(x)$ from $\tilde{J}(f(x, u, w))$** (see Section 2.3 of the class notes).

Other possibilities:

- Learn directly the cost function differences $D_\mu(x, x') = J_\mu(x) - J_\mu(x')$ with an approximation architecture. This is known as **differential training**.
- Methods known as **advantage updating**. [Work with relative Q-factors, i.e., subtract the state-dependent baseline $\min_{u'} Q(x, u')$ from $Q(x, u)$.]

We will cover:

- Infinite horizon theory and algorithms
- Discounted and stochastic shortest path problems

PLEASE REVIEW THE INFINITE HORIZON MATERIAL OF THE CLASS NOTES
WATCH VIDEO LECTURE 7 OF 2019 COURSE OFFERING AT MY WEB SITE