

Topics in Reinforcement Learning:
Lessons from AlphaZero for
(Sub)Optimal Control and Discrete Optimization

Arizona State University
Course CSE 691, Spring 2023

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

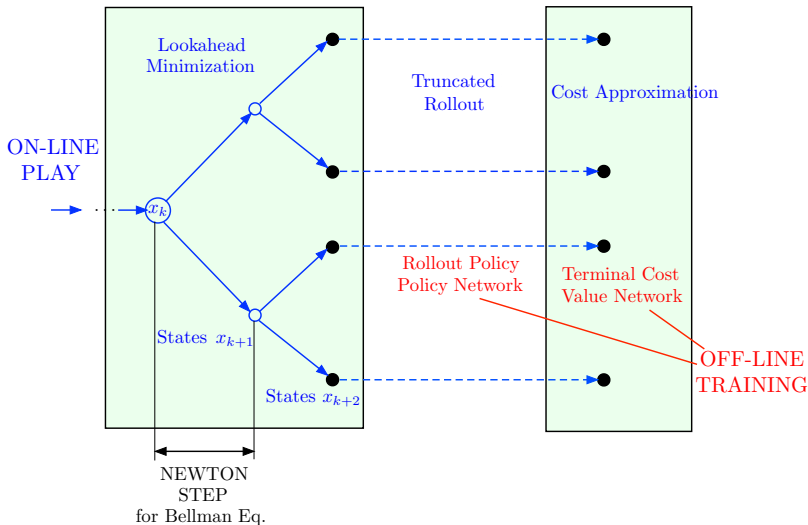
Dimitri P. Bertsekas
dbertsek@asu.edu

Lecture 10

We transition from on-line play to off-line training algorithms
Neural Nets, and Other Parametric Architectures

- 1 Review of What we Have Done and Where we are Going
- 2 Parametric Approximation Architectures for Off-Line Training
- 3 Training of Architectures
- 4 Incremental Optimization of Sums of Differentiable Functions
- 5 Neural Networks

The AlphaZero/MPC Model: Our Starting Point



What We Have Done So Far

We started with four overview/big picture lectures (Chapter 1 of class notes)

- Off-line training, on-line play, Newton step interpretations
- Exact DP, deterministic, stochastic, finite and infinite horizon
- Approximation in value space and rollout
- Problem relations and transformations: State augmentations, termination state problems (e.g., stochastic shortest path), multiagent, POMDP
- Adaptive and model predictive control

Then focused at on-line play algorithms (Chapter 2 of class notes)

- Rollout algorithms for deterministic and stochastic problems; variations (fortified, simplified, constrained, model-free, variance reduction ideas)
- Multistep lookahead search for deterministic problems (pruning, double rollout)
- Multistep lookahead for stochastic problems (certainty equivalence approximations, Monte Carlo tree search)
- Multiagent/multicomponent control problems; variations (autonomous w/ signaling)
- Bayesian optimization, sequential estimation, adaptive control, and rollout

Where We Are Going and What is Left Out

Our plan for the next two lectures (Chapter 3 of class notes)

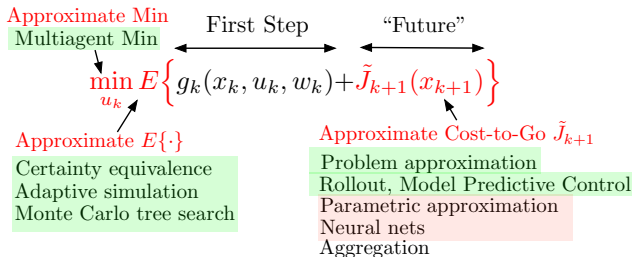
We will cover in some depth and detail

- **Approximation of values and policies** using neural nets and other architectures
- **Training of approximation architectures** with incremental gradient methods
- **Approximate value and policy iteration** with approximation architectures
- **Aggregation**

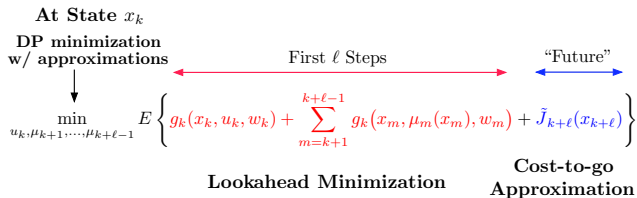
The 2019-2021 course videolectures, the 2019 RL book, and the 2012/2017 DP book deal with additional topics:

- **More on the theory of infinite horizon problems**
- **Stochastic training methods for approximation in value space**: TD(Lambda), other TD methods for policy evaluation, Q-learning
- **Specialized methods for approximation in policy space**: policy gradient methods, random search methods

Recall Approximation in Value Space for On-Line Control Selection



ONE-STEP LOOKAHEAD



MULTISTEP LOOKAHEAD

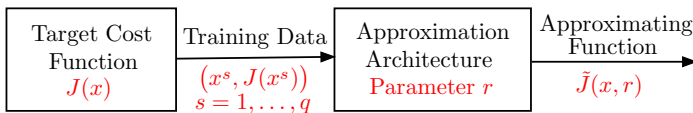
There are two types of off-line approximations in RL:

- **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
 - ▶ Approximation of an optimal policy $\mu_k^*(x_k)$ or $\mu^*(x)$
 - ▶ Approximation of 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)

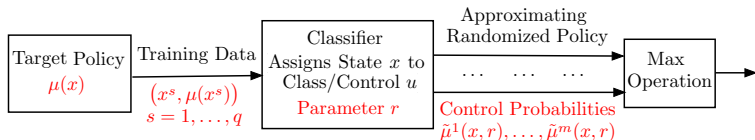
Parametric Approximation of a Target Cost Function



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 $\{u^1, \dots, u^m\}$, **an alternative approach is possible and is commonly used**
- View a policy μ as a **classifier**: A function that maps x into a “category” $\mu(x)$
- Some classifiers introduce **randomized policies**
- Then the output of the classifier is **“control probabilities”**



TRAINING CAN BE DONE WITH CLASSIFICATION SOFTWARE

Randomized policies have continuous components
This helps algorithmically

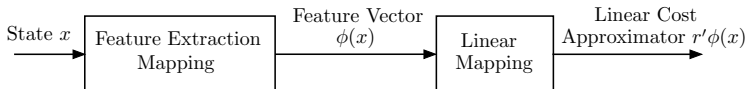
Cost Function Parametric Approximation Generalities

- We start with a class of functions $\tilde{J}(x, r)$ that depend on x and on 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.
- **The training algorithm** is the algorithm that chooses r (typically **regression-type**).
- 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 ,

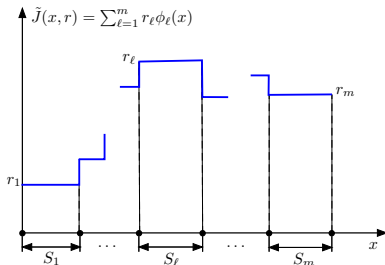
$$\tilde{J}(x, r) = \hat{J}(\phi(x), r),$$

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 Architecture



Piecewise constant approximation

- Partition the state space into subsets $S_1, \dots, S_m$. The ℓ th feature is 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

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

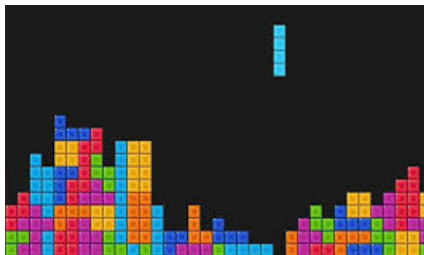
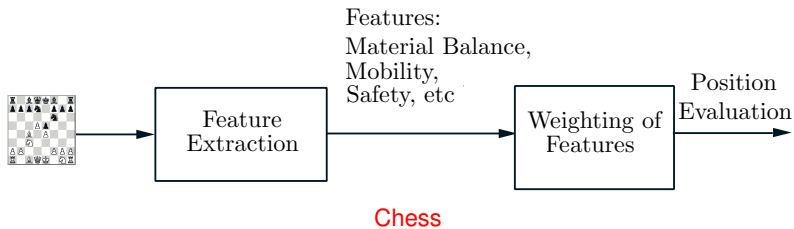
- Here the parameter vector r has components r_0 , r_i , and r_{ij} .

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

An even more general architecture: Polynomials of features of x

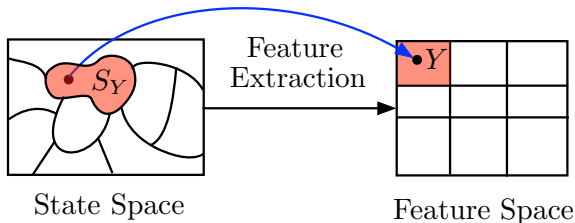
A linear feature-based architecture is a special case

Examples of Problem-Specific Feature-Based Architectures



Tetris

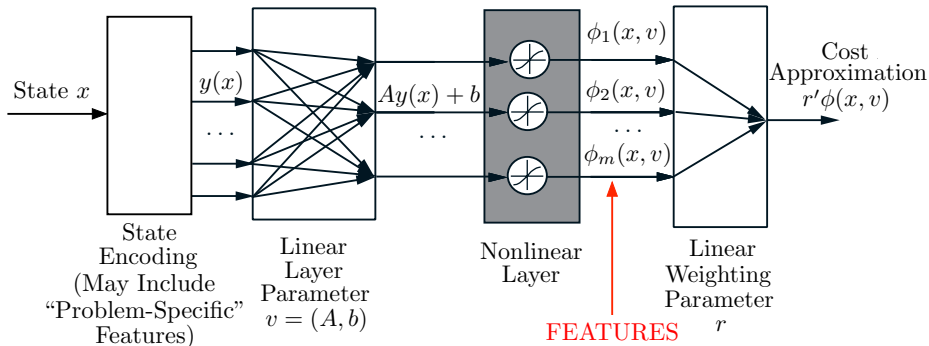
Architectures with Partitioned State Space



A simple method to construct complex approximation architectures:

- Partition the state space into several subsets and **construct a separate cost approximation in each subset**.
- 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
 - ▶ A manifestation of this idea arises in feature-based aggregation (next lecture)

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 - issue of overfitting).

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 involves quadratic minimization 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).

Incremental Gradient Methods (Invented in the 80s, and Analyzed/Extended in the 90s to the Present)

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

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

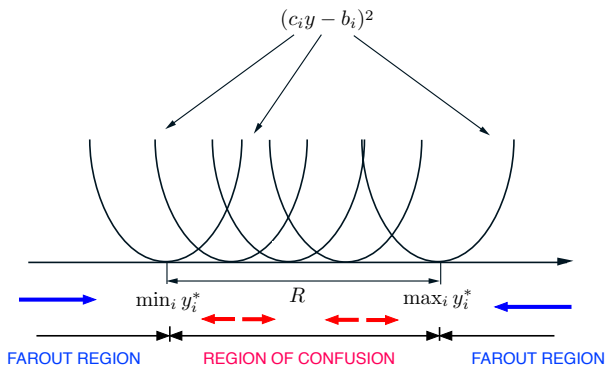
where $\gamma^k > 0$ is a stepsize parameter.

The incremental gradient counterpart

Choose an index i_k and iterate according to

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

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

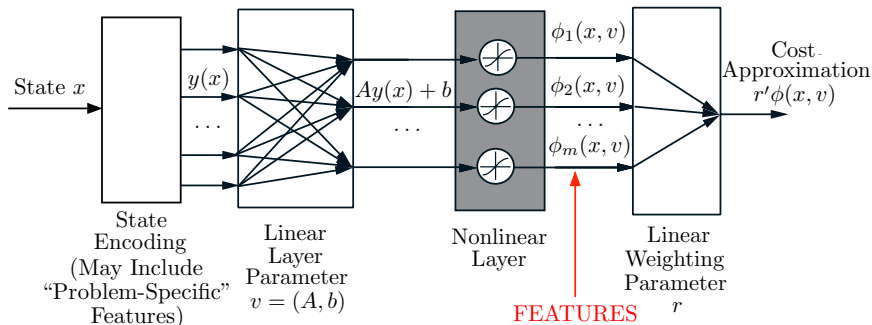
$$y^{k+1} = y^k - \gamma^k \nabla_y F(y^k, w^k)$$

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 class notes and references therein).

Neural Nets: An Architecture that Automatically Constructs Features

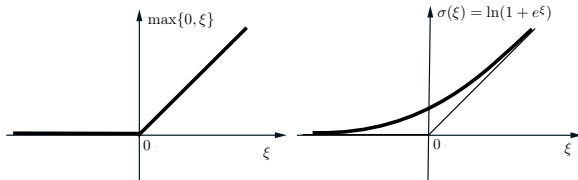


Given a set of state-cost training pairs (x^s, β^s) , $s = 1, \dots, q$, the parameters of the neural network (A, b, r) are obtained by solving the training problem

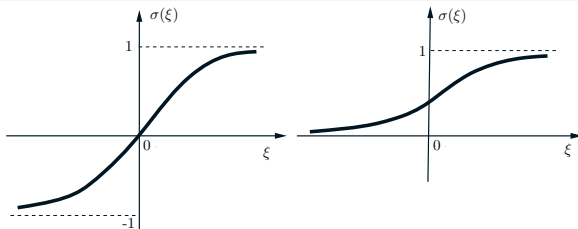
$$\min_{A, b, r} \sum_{s=1}^q \left(\sum_{\ell=1}^m r_{\ell} \sigma((Ay(x^s) + b)_{\ell}) - \beta^s \right)^2$$

- Incremental gradient is typically used for training.
- **Universal approximation property** (can approximate "any" target function, arbitrarily well, with sufficiently large network size).

Rectifier and Sigmoidal Nonlinearities

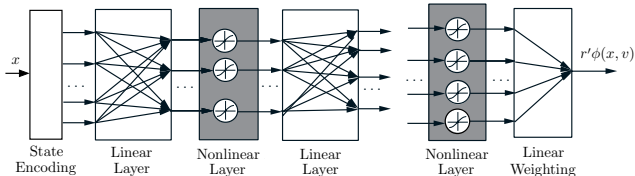


The **rectified linear unit (ReLU)** $\sigma(\xi) = \ln(1 + e^\xi)$. It is the function $\max\{0, \xi\}$ with its corner “smoothed out.”



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.

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

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

affects the quality of the trained architecture (**overparametrization** if $R > 1$).

- 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 extensive research literature.

We will cover:

- Value and policy iteration using neural networks
- Introduction to aggregation