Reinforcement Learning and Optimal Control

ASU, CSE 691, Winter 2019

Dimitri P. Bertsekas
dimitrib@mit.edu

Lecture 6

Outline

1. Parametric Approximation Architectures

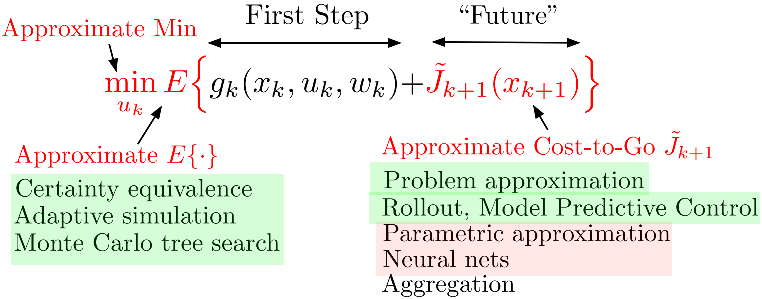
2. Training of Approximation Architectures

3. Incremental Optimization of Sums of Differentiable Functions

4. Neural Networks

5. Neural Nets and Finite Horizon DP

Recall the Approximation in Value Space Framework for Finite Horizon Problems



First Step

"Future"

Approximate Min

$$\min_{u_k} E\Big\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(x_{k+1}) \Big\}$$

Approximate $E\{\cdot\}$

Certainty equivalence
Adaptive simulation
Monte Carlo tree search

Approximate Cost-to-Go $\tilde{J}_{k+1}$

Problem approximation
Rollout, Model Predictive Control
Parametric approximation
Neural nets
Aggregation

Parametric Approximation in Value Space

An approximation architecture is a class of functions $\tilde{J}(x, r)$ that depend on x and a vector $r = (r_1, \ldots, r_m)$ of m “tunable" scalar parameters (or weights).

Issues and terminology

- Aim: Choose r to make $\tilde{J}(x, r)$ close to some target cost function $J(x)$.
- Training algorithm chooses r. It typically uses least squares optimization (regression) to fit $\tilde{J}(x, r)$ to a data set of state-cost pairs.
- An architecture is called linear if $\tilde{J}(x, r)$ is linear in r.
- It is called feature-based if it depend on x via a feature vector $\phi(x)$,

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

  where $\hat{J}$ is some function. 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 ℓth components of r and $\phi(x)$.

Training of Architectures

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

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

- 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.

Training of Nonlinear Architectures

The main training issue

How to exploit the structure of the training problem

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

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 least squares sum; standard gradient and Newton-like methods are essentially inapplicable.
- Incremental iterative methods that operate on a single term $\big(\tilde J(x^s, r) - \beta^s\big)^2$ at each iteration have worked well enough (for many problems).

Incremental Gradient Methods

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.

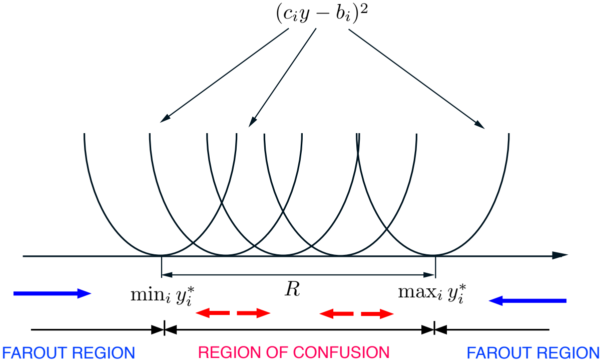
The incremental gradient counterpart

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



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

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 and Stochastic Gradient Methods

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

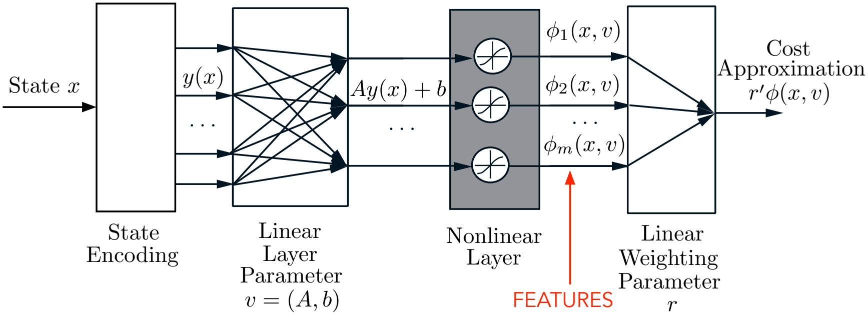
$$y^{k+1} = y^k - \gamma^k \nabla_y F(y^k, w^k)$$

  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].

Implementation Issues of Incremental Methods - Alternative Methods

- 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

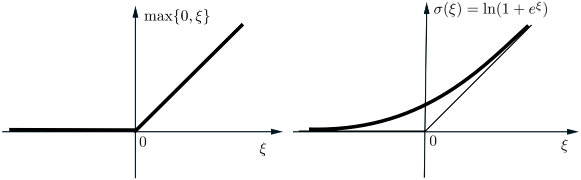


Given a set of state-cost training pairs (x^s,β^s), $s=1,\dots,q$, the parameters of the neural network (A,b,r) are obtained by solving the training problem

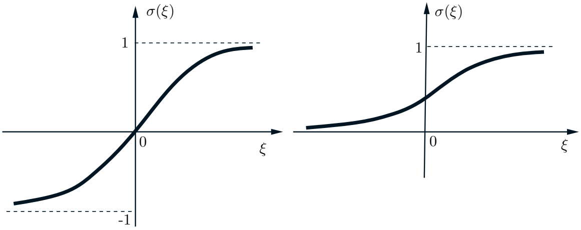
$$\min_{A,b,r}\sum_{s=1}^{q}\left(\sum_{\ell=1}^{m} r_\ell\sigma\left(\left(Ay(x^s)+b\right)_\ell\right)-\beta^s\right)^2$$

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

Rectifier and Sigmoidal Nonlinearities

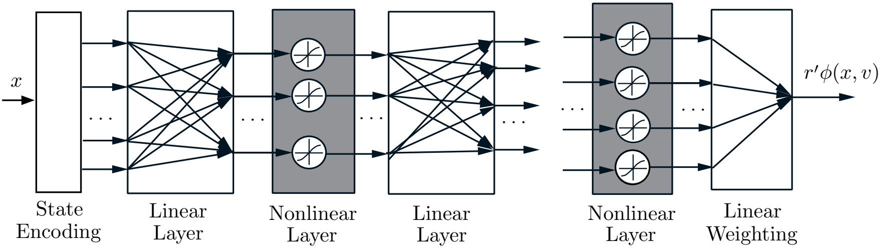


The rectified linear unit $\sigma(\xi) = \ln(1 + e^{\xi})$. It is the rectifier function $\max\{0, \xi\}$ with its corner “smoothed out."



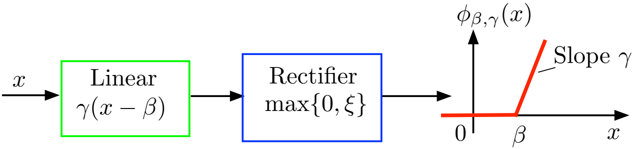
Sigmoidal units: The hyperbolic tangent function $\sigma(\xi) = \tanh(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}}$ is on the left. The logistic function $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$ is on the right.

Deep Neural Networks

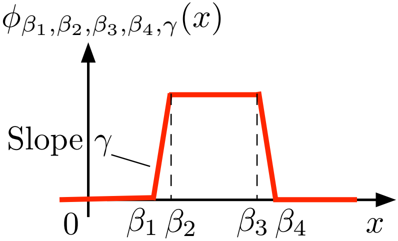


- The multilayer network provides a hierarchy of features (each set of features being a function of the preceding set of features).
- We may use matrices *A* with a special structure that encodes special linear operations such as convolution.
- When such structures are used, the training problem may become easier, because the number of parameters in the linear layers is drastically decreased.
- They have been found more effective than shallow neural nets for some problems.
- Incremental gradient is still used for training. The algorithm is based on an intelligent way of using the chain rule to calculate the incremental gradient at each iteration.

A Working Break: Challenge Question

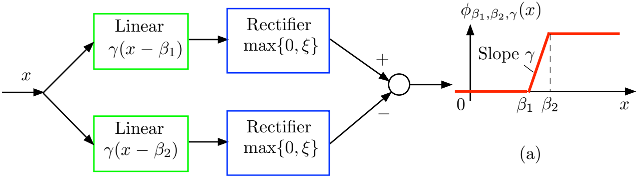


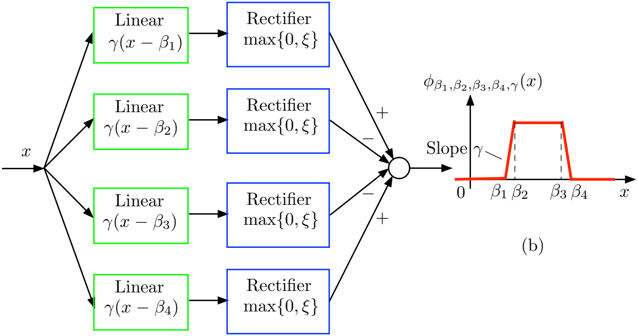
How can we use linear and rectifier units to construct the “pulse" feature below?



- 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), $s = 1, \ldots, q$, where

$$\beta_k^s = \min_{u \in U_k(x_k^s)} E\Big\{ g(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k), r_{k+1}\big)\Big\}, \qquad s = 1, \ldots, q$$

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

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

We minimize over r_k

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

Sequential Q-Factor Approximation

- Consider sequential DP approximation of Q-factor parametric approximations

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

  (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 $\big((x_k^s, u_k^s), \beta_k^s\big)$, 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

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

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

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

  without the need of a model or expected value calculations.
- Important advantage: The on-line calculation of the control is simplified.

About the Next Lecture

We will cover:

- Infinite horizon DP problems: Stochastic shortest path and discounted problems
- Analysis, Bellman's equation, optimality conditions
- Algorithms: Value iteration, policy iteration
- We will likely need more than one lecture

PLEASE READ AS MUCH OF SECTIONS 4.1-4.5 AS YOU CAN

APPENDIX OF CHAPTER 4 CONTAINS PROOFS; TAKE A CRACK AT THEM

PLEASE DOWNLOAD THE LATEST VERSIONS FROM MY WEBSITE