Part II: Approximate Solution Methods

...have been instrumental in recent RL successes
## Evaluation Time!

<table>
<thead>
<tr>
<th>Veranstaltung:</th>
<th>Reinforcement Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lehrperson:</td>
<td>Prof. Dr. Jochen Triesch</td>
</tr>
<tr>
<td>Evaluationstermin:</td>
<td>28.06.2017, 09:00 - 11:00 Uhr</td>
</tr>
<tr>
<td>URL:</td>
<td><a href="http://r.sd.uni-frankfurt.de/3839c835">http://r.sd.uni-frankfurt.de/3839c835</a></td>
</tr>
<tr>
<td>QR-Code:</td>
<td><img src="url" alt="QR-Code" /></td>
</tr>
</tbody>
</table>
Chapter 9:
On-Policy Prediction with Approximation

Motivation and Objectives:

- Look at how experience with a limited part of the state set can be used to produce good behavior over a much larger part: generalization
- What we need is function approximation
- Idea: borrow methods from supervised machine learning, neural networks, pattern recognition, statistical learning
- Overview of function approximation (FA) methods and how they can be adapted to RL
9.1 Value-function Approximation

As usual: Policy Evaluation (the prediction problem):
for a given policy $\pi$, compute the state-value function $v_{\pi}$

In earlier chapters, value function estimates were stored in lookup tables. Now, the value function estimate at time $t$ depends on a parameter $\theta$, that could represent, e.g., the weights of a neural network.

$$\hat{v}(s, \theta) \approx v_{\pi}(s) \quad \theta \in \mathbb{R}^n$$

When we want to update our state value estimates, we make changes to the parameter vector $\theta$. $n$ is typically much smaller than the number of states.
Idea: Adapt Supervised Learning Algorithms

Training Info = desired (target) outputs

Training example = \{input, target output\}

Error = (target output – actual output)
Backups as Training Examples

e.g., the TD(0) backup:

\[ V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \]

As a training example:

\{\text{description of } s_t, \ r_{t+1} + \gamma V(s_{t+1})\} 

input \quad \quad \text{target output}
Any FA Method?

- In principle, yes:
  - artificial neural networks
  - decision trees
  - multivariate regression methods
  - etc.

- But RL has some special requirements:
  - usually want to learn while interacting (online)
  - ability to handle non-stationarity
  - other?
9.2 The Prediction Objective (MSVE)

- Many are applicable but…
- a common and simple one is the Mean-Squared Value Error (MSVE) over a distribution \(d\):

\[
MSVE(\theta) = \sum_{s \in S} d(s) \left[ v_\pi(s) - \hat{v}(s, \theta) \right]^2
\]

- Why \(d\) ?
- Why minimize MSVE?
- Looking for the global vs. a local optimum?
The on-policy distribution

\[ \text{MSVE}(\theta) = \sum_{s \in S} d(s) \left[ v_\pi(s) - \hat{v}(s, \theta) \right]^2 \]

- How should we choose \( d \)?
- Let us assume that \( d \) is always the distribution of states at which backups are done.
- The **on-policy distribution**: the distribution created while following the policy being evaluated. Stronger results are available for this distribution.
9.3 Stochastic-gradient and Semi-gradient Methods

\[ \theta \doteq (\theta_1, \theta_2, \ldots, \theta_n)^\top \]

- Assume \( v_t \) is a smooth differentiable function of \( \theta \) for all states \( s \)
- Since we update the weights every time step we label them with the current time \( t \) as \( \theta_t \)
- Let's assume we're given training examples from the true value function:
  \[ S_t \mapsto v_\pi(S_t) \]
- We still face a difficult function approximation problem, because we have only \( n \) parameters to adjust and we want to generalize to unseen states
Reminder: Gradient Descent

Let $f$ be any function of the parameter space.
Its gradient at any point $\vec{\theta}_t$ in this space is:

$$\nabla f(\theta) = \left( \frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, \ldots, \frac{\partial f(\theta)}{\partial \theta_n} \right)^T$$

Iteratively move down the gradient:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \alpha \nabla_{\vec{\theta}} f(\vec{\theta}_t)$$
Stochastic Gradient Descent

Idea: use the **sample gradient**, i.e., at every step move a bit in the direction that most reduces the error on the current sample:

$$\theta_{t+1} = \theta_t - \frac{1}{2} \alpha \nabla \left[ v_\pi(S_t) - \hat{v}(S_t, \theta_t) \right]^2$$

$$= \theta_t + \alpha \left[ v_\pi(S_t) - \hat{v}(S_t, \theta_t) \right] \nabla \hat{v}(S_t, \theta_t)$$

where

$$\nabla f(\theta) = \left( \frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, \ldots, \frac{\partial f(\theta)}{\partial \theta_n} \right)^T$$

If step size is reduced according to standard stochastic optimization criteria, then this is guaranteed to converge to a local optimum.
Stochastic Gradient Descent Cont.

Usually, we won’t have access to the true values \( v_\pi \). Let’s just replace them with estimates \( U_t \) as we have done in our various backup operations:

\[
\theta_{t+1} = \theta_t + \alpha \left[ U_t - \hat{v}(S_t, \theta_t) \right] \nabla \hat{v}(S_t, \theta_t)
\]

If \( U_t \) is an unbiased estimate of the true values, i.e., \( \mathbb{E}[U_t] = v_\pi(S_t) \) and if the step size is reduced according to standard stochastic optimization criteria, then this is also guaranteed to converge to a local optimum.
Gradient Monte Carlo

Idea: Use the Monte Carlo target $G_t$, which provides an unbiased estimate of the true value.

Gradient Monte Carlo Algorithm for Approximating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : S \times \mathbb{R}^n \rightarrow \mathbb{R}$

Initialize value-function weights $\theta$ as appropriate (e.g., $\theta = 0$)
Repeat forever:
   Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using $\pi$
   For $t = 0, 1, \ldots, T - 1$:
      $\theta \leftarrow \theta + \alpha [G_t - \hat{v}(S_t, \theta)] \nabla \hat{v}(S_t, \theta)$
Semi-Gradient Methods

**Question:** what if we use an estimate of the target value which is not unbiased, in particular because of bootstrapping?

**Answer:** Convergence may no longer be guaranteed.

Bootstrapping targets are not unbiased, because they depend on the current value of $\theta$. 

---

R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction
Example: DP Target

DP Target: \[
\sum_{a,s',r} \pi(a|S_t)p(s', r|S_t, a)[r + \gamma \hat{v}(s', \theta_t)]
\]

\[
\theta_{t+1} = \theta_t - \frac{1}{2} \alpha \nabla \left[ v_\pi(S_t) - \hat{v}(S_t, \theta_t) \right]^2
\]

= \theta_t + \alpha \left[ v_\pi(S_t) - \hat{v}(S_t, \theta_t) \right] \nabla \hat{v}(S_t, \theta_t)

As to satisfy the standard stochastic approximation conditions (2.7), then the SGD for SGD methods assume that

- It may not be immediately apparent why SGD takes only a small step in the direction of the gradient. Could we not move all the way in this direction and minimize an average performance measure such as the MSVE.
- A prototypical semi-gradient method is semi-gradient TD(0), which uses a bootstrapping estimate of the value of the weight vector \( \theta \) as its target. For example, \( \hat{v}(S_t, \theta_t) \) is used as the target value of the weight vector.
- We can approximate it by substituting a small amount in the direction that would most reduce the error on that example:

\[
\theta_{t+1} = \theta_t + \alpha \nabla \hat{v}(S_t, \theta_t)
\]
Semi-Gradient TD(0)

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v}: S^+ \times \mathbb{R}^n \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Initialize value-function weights $\theta$ arbitrarily (e.g., $\theta = 0$)
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      Choose $A \sim \pi(.|S)$
      Take action $A$, observe $R, S'$
      $\theta \leftarrow \theta + \alpha \left[ R + \gamma \hat{v}(S', \theta) - \hat{v}(S, \theta) \right] \nabla \hat{v}(S, \theta)$
      $S \leftarrow S'$
   until $S'$ is terminal

Depending on the function approximator used, the gradient will take different forms. For a (deep) multilayer neural network, this gradient is calculated by the famous backpropagation algorithm.
Example: State Aggregation for 1000-State Random Walk

State aggregation: is special case. States are grouped together with one value estimate for the whole group. Gradient for a group is 1 if current state belongs to the group and 0 otherwise.

![Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 196).](image)

Some of the details of the approximate values are best appreciated by reference to the state distribution $d$ for this task, shown in the lower portion of the figure with a right-side scale. State 500, in the center, is the first state of every episode, but it is rarely visited again. On average, about 1.37% of the time steps are spent in the start state. The states reachable in one step from the start state are the second most visited, with about 0.17% of the time steps being spent in each of them. From there, $d$ falls off almost linearly, reaching about 0.0147% at the extreme states 1 and 1000. The most visible effect of the distribution is on the leftmost groups, whose values are clearly shifted higher than the unweighted average of the true values of states within the group, and on the rightmost groups, whose values are clearly shifted lower. This is due to the states in these areas having the greatest asymmetry in their weightings by $d$. For example, in the leftmost group, state 99 is weighted more than 3 times more strongly than state 0. Thus the estimate for the group is biased toward the true value of state 99, which is higher than the true value of state 0.

### 9.4 Linear Methods

One of the most important special cases of function approximation is that in which the approximate function, $\hat{v}(\cdot, \theta)$, is a linear function of the weight vector, $\theta$. Corresponding to every state $s$, there is a real-valued vector of features $(\phi(s)) = (\phi_1(s), \phi_2(s), \ldots, \phi_n(s))$. The features may be constructed from the states in many different ways; we cover a few possibilities in the next sections. However the features are constructed, the approximate state-value function is given...
9.4 Linear Methods

- Simplest and most popular technique
- For every state there’s a vector of “features”:
  \[ \phi(s) = (\phi_1(s), \phi_2(s), \ldots, \phi_n(s))^\top \]
- Value estimate is linear function of these features:
  \[ \hat{v}(s, \theta) = \theta^\top \phi(s) = \sum_{i=1}^{n} \theta_i \phi_i(s) \]
- What is the gradient in this case?
  \[ \nabla \hat{v}(s, \theta) = \phi(s) \]
Nice Properties of Linear FA Methods

- For RMSE, the error surface is simple: quadratic surface with a single minimum.

- Linear gradient descent TD(\(\lambda\)) converges if:
  - Step size decreases appropriately
  - On-line sampling (states sampled from the on-policy distribution)
  - Converges to parameter vector \(\theta\) with property (see book):

\[
\text{MSVE}(\theta_{TD}) \leq \frac{1}{1 - \gamma} \min_{\theta} \text{MSVE}(\theta)
\]

(Tsitsiklis & Van Roy, 1997)

best parameter vector
Example: Random Walk with State Aggregation and n-step TD methods

Bootstrapping is still beneficial in this case (see book for details)
9.5 Feature Construction for Linear Methods

- What features to use? Depends on the problem!
- Can be generic: polynomial, Fourier based
- Or allows adding domain knowledge to the RL system
- Intuition: features should be “natural” for the task
- Since value function will depend linearly on the features, we may want to add some features that are non-linear combinations of other (simpler) features, so the function approximator can express non-linear relationships of the simpler features.

Example: pole balancing.

Whether a state is a “good” state or not depends on the combination of the pole’s angle and its angular velocity.
Coarse Coding

Assume a continuous 2-dim. state represented with circular features. For every circle there is one weight:

\[ \Sigma \theta_t \]

original representation \rightarrow expanded representation, many features \rightarrow \theta_t \rightarrow \Sigma \rightarrow \text{approximation}

Figure 9.6: Coarse coding. Generalization from state \( s \) to state \( s' \) depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

Figure 9.7: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.
Shaping Generalization in Coarse Coding

a) Narrow generalization  
b) Broad generalization  
c) Asymmetric generalization
Learning and Coarse Coding

\[ \alpha = \frac{0.2}{m} \]

\( m \): number of active features; total of 50 features

Example 9.1: Coarseness of Coarse Coding

This example illustrates the effect on learning of the size of the receptive fields in coarse coding. Linear function approximation based on coarse coding and (9.3) was used to learn a one-dimensional square-wave function (shown at the top of Figure 9.4). The values of this function were used as the targets, \( V_t \). With just one dimension, the receptive fields were intervals rather than circles. Learning was repeated with three different sizes of the intervals: narrow, medium, and broad, as shown at the bottom of the figure. All three cases had the same density of features, about 50 over the extent of the function being learned. Training examples were generated uniformly at random over this extent. The step-size parameter was \( \alpha = \frac{0.2}{m} \), where \( m \) is the number of features that were present at one time. Figure 9.4 shows the functions learned in all three cases over the course of learning. Note that the width of the features had a strong effect early in learning. With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy. However, the final function

\[ m: \text{number of active features; total of 50 features} \]
Tile Coding

- Binary feature for each tile
- Number of features present at any one time is constant
- Binary features means weighted sum easy to compute
- Easy to compute indices of the features present
Tile Coding Cont.

Irregular tilings

- a) Irregular
- b) Log stripes
- c) Diagonal stripes

Hashing
Radial Basis Functions (RBFs)

e.g., Gaussians

\[
\phi_i(s) = \exp \left( - \frac{||s - c_i||^2}{2\sigma_i^2} \right)
\]

Figure 9.13: One-dimensional radial basis functions.
Artificial neural networks (ANNs) are widely used for nonlinear function approximation. A 1991 paper by An et al. and a 1991 paper by Miller et al. published in the Journal of Artificial Intelligence Research explore the use of ANNs in reinforcement learning. In Chapter 16, we describe several examples of reinforcement learning systems that use ANNs.

ANNs have a long history, with early work on artificial neurons, which are the main components of the nervous system. ANNs have been used in reinforcement learning, particularly in the feedforward case. If an ANN has at least one loop in its connections, it is a recurrent network.

The units (the circles in Figure 9.14) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a different activation function. The most common activated function is the rectifier nonlinearity, which is used in the feedforward case. The rectifier nonlinearity is defined as:

\[ f(x) = \max(0, x) \]

where \( x \) is the weighted sum of the input signals. The activation function is used to produce the unit's output, or the net output. The RBF network is a widely used artificial neural network that uses a radial basis function as its activation function. The RBF network has an output layer consisting of two output units.

The RBF network is used in reinforcement learning systems. In Chapter 16, we describe several examples of reinforcement learning systems that use ANNs.

Figure 9.14: Creating these kinds of hierarchical representations without relying exclusively on different activation functions is difficult to obtain well-controlled graded tile activations near the edges of tiles. In high dimensions, the edges of tiles are much more important, and it is difficult to obtain well-controlled graded tile activations. This is especially true when there are more than two state dimensions. In high dimensions, the edges of tiles are much more important, and it is difficult to obtain well-controlled graded tile activations.

The functions are parametrized. The successive layers of a deep learning system are hierarchical compositions of many layers of lower-level abstractions. In high dimensions, the edges of tiles are much more important, and it is difficult to obtain well-controlled graded tile activations. This is especially true when there are more than two state dimensions. In high dimensions, the edges of tiles are much more important, and it is difficult to obtain well-controlled graded tile activations.
Key Concepts for such Feedforward Neural Networks

- Training: Backpropagation algorithm
  - forward pass: given input, calculate output
  - backward pass: how to change weights to make output close to desired output
- Problem of overfitting: good performance on training set, but poor generalization to new data
- Illustration with polynomials (see black board)
- Much research trying to combat overfitting (see next)
Much research trying to combat overfitting:

- **Regularization** (e.g. simple weight decay to keep weights small)
- **Cross validation**: only train as long as performance on an independent validation set keeps improving
- **Smart initialization** of the network weights
- **Drop-out**: randomly deactivate neurons and their connections during training, effectively training with large numbers of randomly “thinned” networks and then combining their responses together
- **Weight sharing** (as in convolutional neural networks, see next)
Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

- weight sharing (helps to avoid overfitting): local filters (receptive fields) “shifted” across the image
- pooling layers (subsampling, spatial averaging, sometimes also maximum operation) help to incorporate shift invariance
9.7 Least-Squares TD

- skipped, see book
Function approximation is useful for generalizing from one state to the next, especially if there are many states.

Connection to supervised learning: treat backup as supervised training example.

Objective often: minimize MSVE for on-policy distribution.

Many supervised learning techniques can be used.

Linear function approximation theoretically best understood.

Nonlinear function approximation with deep neural networks ("deep reinforcement learning").