8 Approximate Value Functions

Up to this point, we have assumed that the value function can be represented as a table. Tables are only useful representations for small discrete problems. Problems with larger state spaces may require an infeasible amount of memory, and the exact methods discussed in the previous chapter may require an infeasible amount of computation. For such problems, we often have to resort to *approximate dynamic programming*, where the solution may not be exact.[1] One way to approximate solutions is to use *value function approximation*, which is the subject of this chapter. We will discuss different approaches to approximating the value function and how to incorporate dynamic programming to derive approximately optimal policies.

[1] A deeper treatment of this field is provided by W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, 2nd ed. Wiley, 2011.

8.1 Parametric Representations

We will use $U_\theta(s)$ to denote our *parametric representation* of the value function, where θ is the vector of *parameters*. There are many different ways to represent $U_\theta(s)$, several of which will be mentioned later in this chapter. Assuming we have such an approximation, we can extract an action according to:

$$\pi(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s, a) U_\theta(s') \right) \tag{8.1}$$

Value function approximations are often used in problems with continuous state spaces, in which case the summation above may be replaced with an integral. The integral can be approximated using transition model samples.

An alternative to the computation in equation (8.1) is to approximate the action value function $Q(s,a)$. If we use $Q_\theta(s,a)$ to represent our parametric

approximation, we can obtain an action according to:

$$\pi(s) = \arg\max_a Q_\theta(s, a) \tag{8.2}$$

This chapter discusses how we can apply dynamic programming at a finite set of states $S = s_{1:m}$ to arrive at a parametric approximation of the value function over the full state space. Different schemes can be used to generate this set. If the state space is relatively low-dimensional, we can define a grid. Another approach is to use random sampling from the state space. However, some states are more likely to be encountered than others and are therefore more important in constructing the value function. We can bias the sampling towards more important states by running simulations with some policy, perhaps initially random, from a plausible set of initial states.

An iterative approach can be used to enhance our approximation of the value function at the states in S. We alternate between improving our value estimates at S through dynamic programming and refitting our approximation at those states. Algorithm 8.1 provides an implementation where the dynamic programming step consists of Bellman backups as done in value iteration (section 7.5). A similar algorithm can be created for action value approximations Q_θ.

```
struct ApproximateValueIteration
    U0    # initial parameterized value function that supports fit!
    S     # set of discrete states for performing backups
    k_max # maximum number of iterations
end

function solve(M::ApproximateValueIteration, 𝒫::MDP)
    U0, S, k_max = M.U0, M.S, M.k_max
    for k in 1:k_max
        U = [backup(𝒫, U0, s) for s in S]
        fit!(U0, S, U)
    end
    return ValueFunctionPolicy(𝒫, U0)
end
```

Algorithm 8.1. Approximate value iteration for an MDP with parameterized value function approximation `U0`. We perform backups (defined in algorithm 7.7) at the states in `S` to obtain a vector of utilities `U`. We then call `fit!(U0, S, U)`, which modifies the parametric representation `U0` to better match the value of the states in `S` to the utilities in `U`. Different parametric approximations have different implementations for `fit!`.

All of the parametric representations discussed in this chapter can be used with algorithm 8.1. To be used with that algorithm, a representation needs to support the evaluation of U_θ and the fitting of U_θ to estimates of the utilities at the points in S.

We can group the parametric representations into two categories. The first category includes *local approximation* methods where θ corresponds to the values at the states in S. To evaluate $U_\theta(s)$ at an arbitrary state s, we take a weighted sum of the values stored in S. The second category includes *global approximation* methods where θ is not directly related to the values at the states in S. In fact, θ may have far fewer or even far more components than there are states in S.

Both local approximation as well as many global approximations can be viewed as a *linear function approximation* $U_\theta(s) = \theta^\top \beta(s)$, where methods differ in how they define the vector function β. In local approximation methods, $\beta(s)$ determines how to weight the utilities of the states in S to approximate the utility at state s. In many global approximation methods, $\beta(s)$ is viewed as a set of basis functions that are combined together in a linear fashion to obtain an approximation for an arbitrary s.

We can also approximate the action value function using a linear function, $Q_\theta(s, a) = \theta^\top \beta(s, a)$. In the context of local approximations, we can provide approximations over continuous action spaces by choosing a finite set of actions $A \subset \mathcal{A}$. Our parameter vector θ would then consist of $|S| \times |A|$ components, each corresponding to a state-action value. Our function $\beta(s, a)$ would return a vector with the same number of components that specifies how to weight together our finite set of state-action values to obtain an estimate of the utility associated with state s and action a.

8.2 Nearest Neighbor

A simple approach to local approximation is to use the value of the state in S that is the *nearest neighbor* of s. In order to use this approach, we need a *distance metric* (see appendix A.3). We use $d(s, s')$ to denote the distance between two states s and s'. The approximate value function is then $U_\theta(s) = \theta_i$, where $i = \arg\min_{j \in 1:m} d(s_j, s)$. Figure 8.1 shows an example of a value function represented using the nearest neighbor scheme.

We can generalize this approach to average together the values of the *k-nearest neighbors*. This approach still results in piecewise constant value functions, but different values for k can result in better approximations. Figure 8.1 shows examples of value functions approximated with different values for k. Algorithm 8.2 provides an implementation.

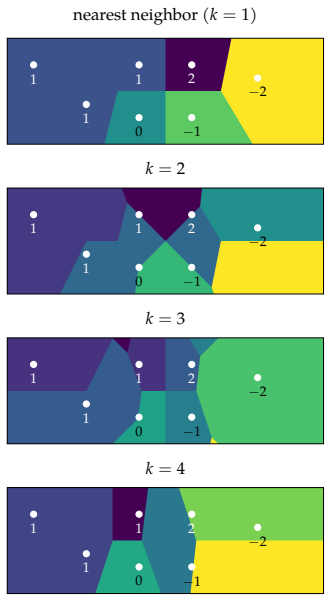


Figure 8.1. Approximating the values of states in a two-dimensional continuous state space using the mean of the utility values of their k-nearest neighbors according to Euclidean distance. The resulting value function is piecewise constant.

```
mutable struct NearestNeighborValueFunction
    k # number of neighbors
    d # distance function d(s, s')
    S # set of discrete states
    θ # vector of values at states in S
end

function (U0::NearestNeighborValueFunction)(s)
    dists = [U0.d(s,s') for s' in U0.S]
    ind = sortperm(dists)[1:U0.k]
    return mean(U0.θ[i] for i in ind)
end

function fit!(U0::NearestNeighborValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.2. The k-nearest neighbors method, which approximates the value of a state `s` based on the `k` closest states in `S` as determined by a distance function `d`. The vector `θ` contains the values of the states in `S`. Greater efficiency can be achieved by using specialized data structures, such as kd-trees, implemented in `NearestNeighbors.jl`.

8.3 Kernel Smoothing

Another local approximation method is *kernel smoothing*, where the utilities of the states in S are smoothed over the entire state space. This method requires defining a *kernel function* $k(s, s')$ that relates pairs of states s and s'. We generally want $k(s, s')$ to be higher for states that are closer together because those values tell us how to weight together the utilities associated with the states in S. This method results in the linear approximation:

$$U_\theta(s) = \sum_{i=1}^{m} \theta_i \beta_i(s) = \theta^\top \beta(s) \tag{8.3}$$

where

$$\beta_i(s) = \frac{k(s, s_i)}{\sum_{j=1}^{m} k(s, s_j)} \tag{8.4}$$

Algorithm 8.3 provides an implementation.

There are many different ways we can define a kernel function. We can define our kernel to simply be the inverse of the distance between states:

$$k(s, s') = \max(d(s, s'), \epsilon)^{-1} \tag{8.5}$$

where ϵ is a small positive constant to avoid dividing by zero when $s = s'$. Figure 8.2 shows value approximations using several different distance functions. As we can see, kernel smoothing can result in smooth value function approximations, in contrast with k-nearest neighbors. Figure 8.3 applies this kernel to a discrete hex world problem and shows the outcome of a few iterations of approximate value iteration (algorithm 8.1). Figure 8.4 shows a value function and policy learned for the mountain car problem (appendix F.4) with a continuous state space.

Another common kernel is the *Gaussian kernel*:

$$k(s, s') = \exp\left(-\frac{d(s, s')^2}{2\sigma^2}\right) \tag{8.6}$$

where σ controls the degree of smoothing.

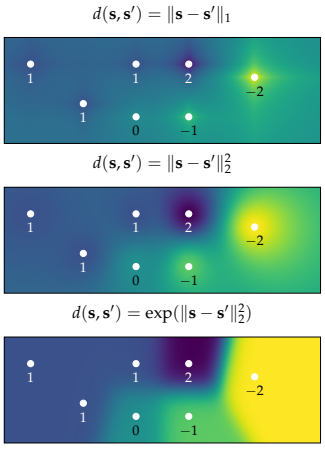


Figure 8.2. Approximating the values of states in a two-dimensional continuous state space by assigning values based on proximity to several states with known values. Approximations are constructed using several different distance functions.

```
mutable struct LocallyWeightedValueFunction
    k # kernel function k(s, s′)
    S # set of discrete states
    θ # vector of values at states in S
end

function (U0::LocallyWeightedValueFunction)(s)
    w = normalize([U0.k(s,s′) for s′ in U0.S], 1)
    return U0.θ ⋅ w
end

function fit!(U0::LocallyWeightedValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.3. Locally weighted value function approximation defined by a kernel function `k` and a vector of utilities `θ` at states in `S`.

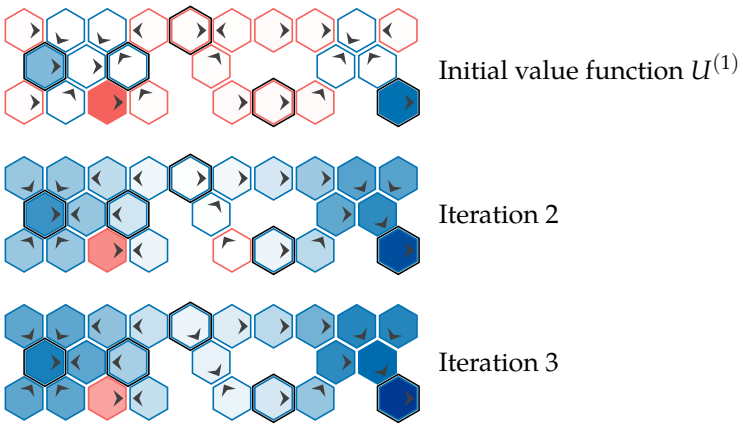


Figure 8.3. Local approximation value iteration used to iteratively improve an approximate value function on the hex world problem. The five outlined states are used to approximate the value function. The value of the remaining states are approximated using the distance function $\|\mathbf{s} - \mathbf{s}'\|_2^2$. The resulting policy is reasonable but nevertheless sub-optimal.

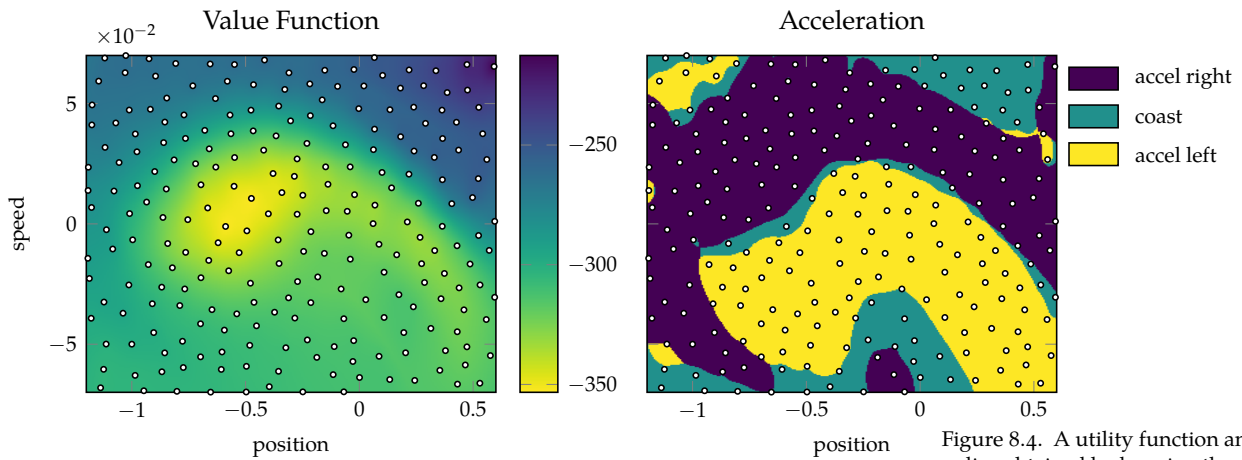


Figure 8.4. A utility function and policy obtained by learning the action values for a finite set of states (white) in the mountain car problem using the distance function $\|\mathbf{s} - \mathbf{s}'\|_2 + 0.1$.

8.4 Linear Interpolation

Linear interpolation is another common approach to local approximation. The one-dimensional case is straightforward, where the approximated value for a state s between two states s_1 and s_2 is:

$$U_\theta(s) = \alpha\theta_1 + (1-\alpha)\theta_2 \tag{8.7}$$

with $\alpha = (s_2 - s)/(s_2 - s_1)$. This case is shown in both figure 8.5 and figure 8.6.

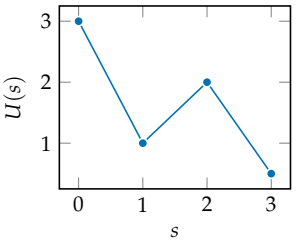


Figure 8.5. One-dimensional linear interpolation produces interpolated values along the line segment connecting two points.

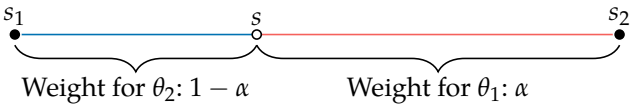
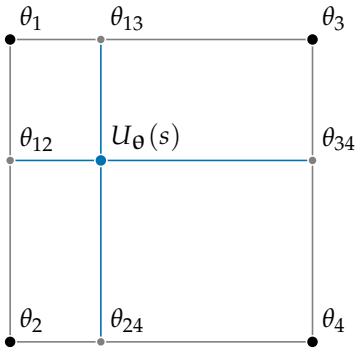


Figure 8.6. The weight assigned to each point in one dimension is proportional to the length of the segment on the opposite side of the interpolation state.

Linear interpolation can be extended to a multidimensional grid. In the two-dimensional case, called *bilinear interpolation*, we interpolate between four vertices. Bilinear interpolation is done through single-dimensional linear interpolation, once in each axis, requiring the utility of four states at the grid vertices. This interpolation is shown in figure 8.7.



θ_{12} = 1D interpolation between θ_1 and θ_2 along vertical axis.
θ_{24} = 1D interpolation between θ_2 and θ_4 along horizontal axis.
θ_{13} = 1D interpolation between θ_1 and θ_3 along horizontal axis.
θ_{34} = 1D interpolation between θ_3 and θ_4 along vertical axis.

$$U_\theta(s) = \begin{cases} \text{1D interpolation between } \theta_{12} \text{ and } \theta_{34} \text{ along horizontal axis.} \\ \text{1D interpolation between } \theta_{13} \text{ and } \theta_{24} \text{ along vertical axis.} \end{cases}$$

Figure 8.7. Linear interpolation on a two-dimensional grid is achieved through linear interpolation on each axis in turn, in either order.

Given four vertices with coordinates $s_1 = [x_1, y_1]$, $s_2 = [x_1, y_2]$, $s_3 = [x_2, y_1]$, and $s_4 = [x_2, y_2]$, and a sample state $s = [x, y]$, the interpolated value is:

$$
\begin{aligned}
U_\theta(s) &= \alpha\theta_{12} + (1-\alpha)\theta_{34} && (8.8)\\
&= \frac{x_2 - x}{x_2 - x_1}\theta_{12} + \frac{x - x_1}{x_2 - x_1}\theta_{34} && (8.9)\\
&= \frac{x_2 - x}{x_2 - x_1}(\alpha\theta_1 + (1-\alpha)\theta_2) + \frac{x - x_1}{x_2 - x_1}(\alpha\theta_3 + (1-\alpha)\theta_4) && (8.10)\\
&= \frac{x_2 - x}{x_2 - x_1}\left(\frac{y_2 - y}{y_2 - y_1}\theta_1 + \frac{y - y_1}{y_2 - y_1}\theta_2\right) + \frac{x - x_1}{x_2 - x_1}\left(\frac{y_2 - y}{y_2 - y_1}\theta_3 + \frac{y - y_1}{y_2 - y_1}\theta_4\right) && (8.11)\\
&= \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4 && (8.12)
\end{aligned}
$$

The resulting interpolation weighs each vertex according to area of its opposing quadrant, as shown in figure 8.8.

Multilinear interpolation in d dimensions is similarly achieved by linearly interpolating along each axis, requiring 2^d vertices. Here too, the utility of each vertex is weighted according to the volume of the opposing hyperrectangle. Multilinear interpolation is implemented in algorithm 8.4. Figure 8.9 demonstrates this approach on a two-dimensional state space.

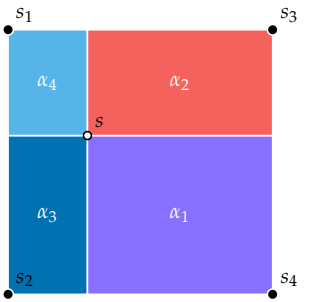


Figure 8.8. Linear interpolation on a 2D grid results in a contribution of each vertex equal to the relative area of its opposing quadrant: $U_\theta(s) = \alpha_1\theta_1 + \alpha_2\theta_2 + \alpha_3\theta_3 + \alpha_4\theta_4$

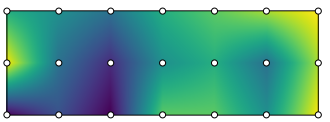


Figure 8.9. Two-dimensional linear interpolation over a 3×7 grid.

8.5 Simplex Interpolation

Multilinear interpolation can be inefficient in high dimensions. Rather than weighting the contributions of 2^d points, *simplex interpolation* considers only $d + 1$ points in the neighborhood of a given state to produce a continuous surface that matches the known sample points.

We start with a multidimensional grid and divide each cell into $d!$ *simplexes*, which are multidimensional generalizations of triangles defined by the *convex hull* of $d + 1$ vertices. This process is known as *Coxeter-Freudenthal-Kuhn triangulation*,[2] and it ensures that any two simplexes that share a face will produce equivalent values across the face, thus producing continuity when interpolating, as shown in figure 8.10.

[2] A. W. Moore, "Simplicial Mesh Generation with Applications," Ph.D. dissertation, Cornell University, 1992.

```
mutable struct MultilinearValueFunction
    o # position of lower-left corner
    δ # vector of widths
    θ # vector of values at states in S
end

function (U0::MultilinearValueFunction)(s)
    o, δ, θ = U0.o, U0.δ, U0.θ
    Δ = (s - o)./δ
    # Multidimensional index of lower-left cell
    i = min.(floor.(Int, Δ) .+ 1, size(θ) .- 1)
    vertex_index = similar(i)
    d = length(s)
    u = 0.0
    for vertex in 0:2^d-1
        weight = 1.0
        for j in 1:d
            # Check whether jth bit is set
            if vertex & (1 << (j-1)) > 0
                vertex_index[j] = i[j] + 1
                weight *= Δ[j] - i[j] + 1
            else
                vertex_index[j] = i[j]
                weight *= i[j] - Δ[j]
            end
        end
        u += θ[vertex_index...]*weight
    end
    return u
end

function fit!(U0::MultilinearValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.4. A method for conducting multilinear interpolation to estimate the value of state vector `s` for known state values `θ` over a grid defined by a lower-left vertex `o` and vector of widths `δ`. Vertices of the grid can all be written `o + δ.*i` for some non-negative integral vector `i`. The package `Interpolations.jl` also provides multilinear and other interpolation methods.

To illustrate, suppose we have translated and scaled the cell containing a state such that the lowest vertex is **0** and the diagonally opposite vertex is **1**. There is a simplex for each permutation of $1:d$. The simplex given by permutation $\mathbf{p}$ is the set of points $\mathbf{x}$ satisfying:

$$0 \leq x_{p_1} \leq x_{p_2} \leq \cdots \leq x_{p_d} \leq 1 \tag{8.13}$$

Figure 8.11 shows the simplexes obtained for the unit cube.

Simplex interpolation first translates and scales a state vector $\mathbf{s}$ to the unit hypercube of its corresponding cell to obtain $\mathbf{s}'$. It then sorts the entries in $\mathbf{s}'$ to determine which simplex contains $\mathbf{s}'$. The utility at $\mathbf{s}'$ can then be expressed by a unique linear combination of the vertices of that simplex.

Example 8.1 provides an example of simplex interpolation. The process is implemented in algorithm 8.5.

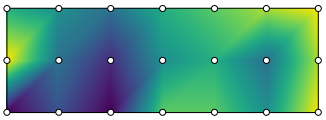


Figure 8.10. Two-dimensional simplex interpolation over a 3×7 grid.

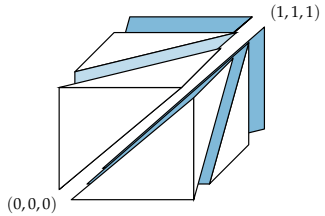


Figure 8.11. A triangulation of a unit cube. Based on Figure 2.1 of A. W. Moore, "Simplicial Mesh Generation with Applications," Ph.D. dissertation, Cornell University, 1992.

Example 8.1. An example of simplex interpolation in three dimensions.

Consider a three-dimensional simplex given by the permutation $\mathbf{p} = [3, 1, 2]$ such that points within the simplex satisfy $0 \leq x_3 \leq x_1 \leq x_2 \leq 1$. This simplex has vertices $(0, 0, 0)$, $(0, 1, 0)$, $(1, 1, 0)$, and $(1, 1, 1)$.

Any point $\mathbf{s}$ belonging to the simplex can thus be expressed by a weighting of the vertices:

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = w_1 \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + w_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + w_3 \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + w_4 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

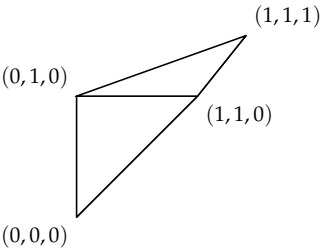
We can determine the values of the last three weights in succession:

$$w_4 = s_3 \qquad w_3 = s_1 - w_4 \qquad w_2 = s_2 - w_3 - w_4$$

We obtain w_1 by enforcing that the weights sum to one.

If $\mathbf{s} = [0.3, 0.7, 0.2]$, then the weights are:

$$w_4 = 0.2 \qquad w_3 = 0.1 \qquad w_2 = 0.4 \qquad w_1 = 0.3$$



```
mutable struct SimplexValueFunction
    o # position of lower-left corner
    δ # vector of widths
    θ # vector of values at states in S
end

function (U0::SimplexValueFunction)(s)
    Δ = (s - U0.o)./U0.δ
    # Multidimensional index of upper-right cell
    i = min.(floor.(Int, Δ) .+ 1, size(U0.θ) .- 1) .+ 1
    u = 0.0
    s′ = (s - (U0.o + U0.δ.*(i.-2))) ./ U0.δ
    p = sortperm(s′) # increasing order
    w_tot = 0.0
    for j in p
        w = s′[j] - w_tot
        u += w*U0.θ[i...]
        i[j] -= 1
        w_tot += w
    end
    u += (1 - w_tot)*U0.θ[i...]
    return u
end

function fit!(U0::SimplexValueFunction, S, U)
    U0.θ = U
    return U0
end
```

Algorithm 8.5. A method for conducting simplex interpolation to estimate the value of state vector `s` for known state values `θ` over a grid defined by a lower-left vertex `o` and vector of widths `δ`. Vertices of the grid can all be written `o + δ.*i` for some non-negative integral vector `i`. Simplex interpolation is also implemented in the general `GridInterpolations.jl` package.

8.6 Linear Regression

A simple global approximation approach is *linear regression*, where $U_\theta(s)$ is a linear combination of *basis functions*.[3] These basis functions are generally a nonlinear function of the state s and are combined together into a vector function $\beta(s)$ or $\beta(s, a)$, resulting in the approximations:

$$U_\theta(s) = \theta^\top \beta(s) \qquad Q_\theta(s, a) = \theta^\top \beta(s, a) \tag{8.14}$$

[3] Also commonly referred to as *features*.

Although our approximation is linear with respect to the basis functions, the resulting approximation may be nonlinear with respect to the underlying state variables. Figure 8.12 illustrates this concept. Example 8.2 provides an example of global linear value approximation using polynomial basis functions for the continuous mountain car problem, resulting in a nonlinear value function approximation with respect to the state variables.

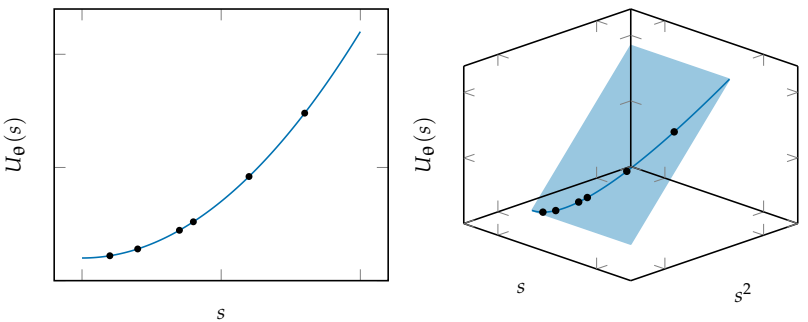


Figure 8.12. Linear regression with nonlinear basis functions is linear in higher dimensions. Here, polynomial regression can be seen as linear in a three-dimensional space. The function exists in the plane formed from its bases, but it does not occupy the entire plane because the terms are not independent.

Adding more basis functions generally improves the ability to match the target utilities at the states in S, but too many basis functions can lead to poor approximations at other states. Principled methods exist for choosing an appropriate set of basis functions for our regression model.[4]

[4] See Chapter 14 of M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019. or Chapter 7 of T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001.

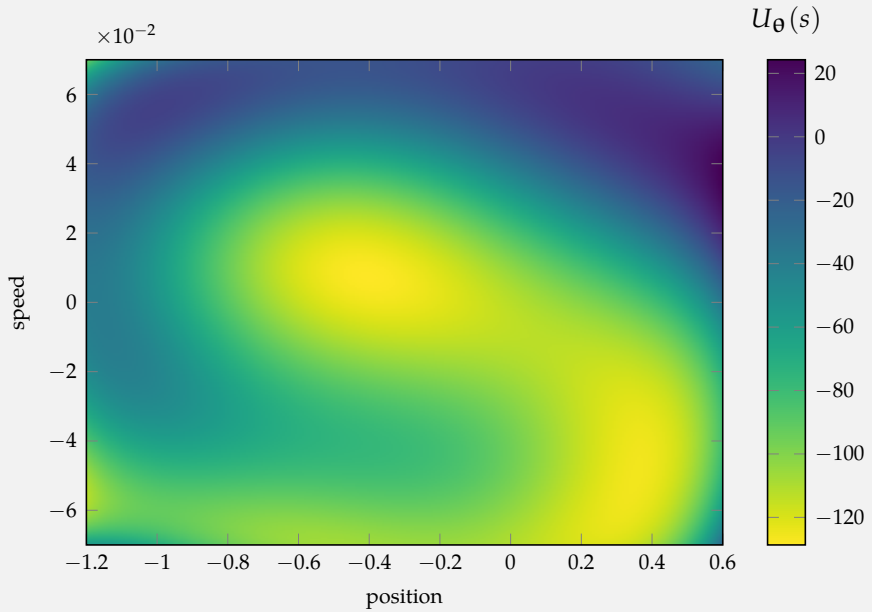
Fitting linear models involves determining the vector θ that minimizes the squared error of the predictions at the states in $S = s_{1:m}$. If the utilities associated with those states are denoted $u_{1:m}$, then we want to find the θ that minimizes

$$\sum_{i=1}^{m} (\hat{U}_\theta(s_i) - u_i)^2 = \sum_{i=1}^{m} (\theta^\top \beta(s_i) - u_i)^2 \tag{8.15}$$

We can approximate the value function for the mountain car problem using a linear approximation. The problem has a continuous state space with two dimensions consisting of position x and speed v. Here are the basis functions up to degree six:

$$\boldsymbol{\beta}(s) = \begin{bmatrix} 1, & & & & & & \\ x, & v, & & & & & \\ x^2, & xv, & v^2, & & & & \\ x^3, & x^2v, & xv^2, & v^3, & & & \\ x^4, & x^3v, & x^2v^2, & xv^3, & v^4, & & \\ x^5, & x^4v, & x^3v^2, & x^2v^3, & xv^4, & v^5, & \\ x^6, & x^5v, & x^4v^2, & x^3v^3, & x^2v^4, & xv^5, & v^6 \end{bmatrix}$$

Below is a plot of an approximate value function fit to state-value pairs from an expert policy:



Example 8.2. An example of using a linear approximation to the mountain car value function. The choice of basis functions makes a big difference. The optimal value function for the mountain car is nonlinear, with a spiral shape and discontinuities. Even sixth-degree polynomials do not produce a perfect fit.

The optimal θ can be computed through some simple matrix operations. We first construct a matrix $\mathbf{X}$ where each of the m rows $\mathbf{X}_{i,:}$ contains $\beta(s_i)^\top$.[5] It can be shown that the value of θ that minimizes the squared error is:

$$\theta = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top u_{1:m} = \mathbf{X}^+ u_{1:m} \tag{8.16}$$

where $\mathbf{X}^+$ is the Moore-Penrose *pseudoinverse* of matrix $\mathbf{X}$. The pseudoinverse is often implemented by first computing the *singular value decomposition*, $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}^*$. We then have

$$\mathbf{X}^+ = \mathbf{U}\Sigma^+\mathbf{U}^* \tag{8.17}$$

The pseudoinverse of the diagonal matrix $\mathbf{\Sigma}$ is obtained by taking the reciprocal of each nonzero element of the diagonal and then transposing the result.

Figure 8.13 shows how the utilities of states in S are fit with several different basis function families. Different choices of basis functions result in different errors.

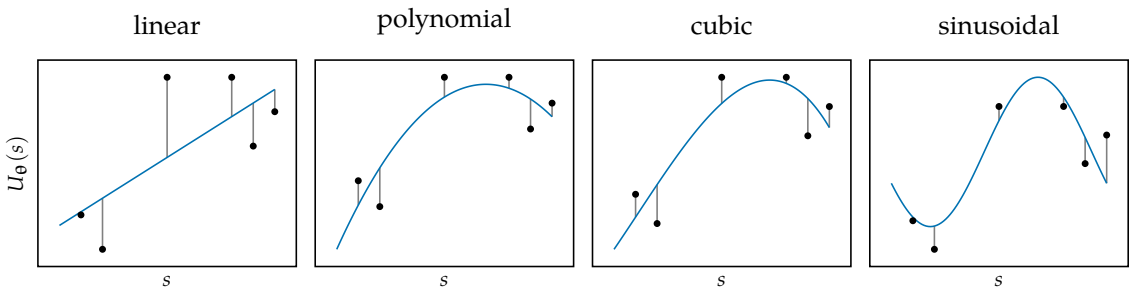


Figure 8.13. Linear regression with different basis function families.

Algorithm 8.6 provides an implementation for evaluating and fitting linear regression models of the value function. Example 8.3 demonstrates this approach on the mountain car problem.

8.7 Neural Network Regression

Neural network regression relieves us of having to construct an appropriate set of basis functions as required in linear regression. Instead, a *neural network* is used to represent our value function. For a review of neural networks, see appendix D. The input to the neural network would be the state variables, and the output

[5] For an overview of the mathematics involved in linear regression as well as more advanced techniques, see T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001.

```
mutable struct LinearRegressionValueFunction
    β # basis vector function
    θ # vector of paramters
end

function (U0::LinearRegressionValueFunction)(s)
    return U0.β(s) ⋅ U0.θ
end

function fit!(U0::LinearRegressionValueFunction, S, U)
    X = hcat([U0.β(s) for s in S]...)'
    U0.θ = pinv(X)*U
    return U0
end
```

Algorithm 8.6. Linear regression value function approximation, defined by a basis vector function `β` and a vector of parameters `θ`.

would be the utility estimate. The parameters θ would correspond to the *weights* in the neural network.

As discussed in appendix D, we can optimize the network weights to achieve some objective. In the context of approximate dynamic programming, we would want to minimize the error of our predictions, just as we did in the previous section. However, minimizing the squared error cannot be done through simple matrix operations. Instead, we generally have to rely upon optimization techniques, such as gradient descent. Fortunately, computing the gradient of neural networks can be done exactly through straightforward application of the derivative chain rule.

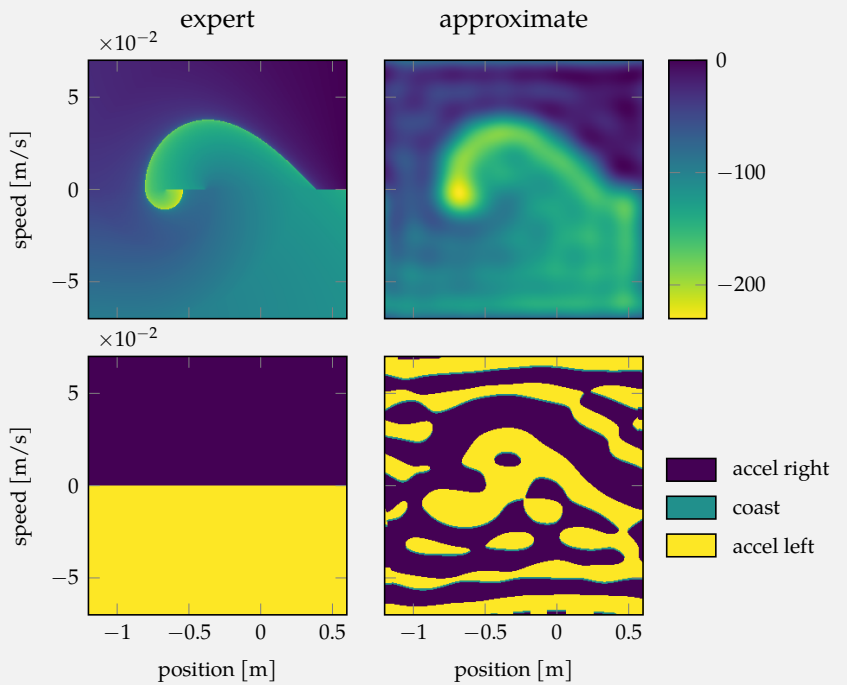
8.8 Summary

- For large or continuous problems, we can attempt to find approximate policies represented by parameterized models of the value function.
- The approaches taken in this chapter involve iteratively applying steps of dynamic programming at a finite set of states and refining our parametric approximation.
- Local approximation techniques approximate the value function based on the values of nearby states with known values.
- A variety of local approximation techniques include nearest neighbor, kernel smoothing, linear interpolation, and simplex interpolation.

We can apply linear regression to learn a value function for the mountain car problem. The optimal value function has the form of a spiral, which can be difficult to approximate with polynomial basis functions (see example 8.2). We use Fourier basis functions whose components have the form:

$$
\begin{aligned}
b_0(x) &= 1/2 \\
b_{s,i}(x) &= \sin(2\pi i x/T) \text{ for } i = 1, 2, \ldots \\
b_{c,i}(x) &= \cos(2\pi i x/T) \text{ for } i = 1, 2, \ldots
\end{aligned}
$$

where T is the width of the component's domain. The multidimensional Fourier basis functions are all combinations of the one-dimensional components across the state-space axes. Below we use a 8th order approximation, so i ranges up to 8. The expert policy is to accelerate in the direction of motion.



Example 8.3. Linear regression using Fourier bases used to approximate the value function for the mountain car problem (appendix F.4). Value functions (top row) and resulting policies (bottom row) are shown. The globally approximated value function is a poor fit despite using 8th order Fourier basis functions. The resulting approximate policy is not a close approximation to the expert policy. The small timestep in the mountain car problem causes even small changes in the value function landscape to affect the policy. Optimal utility functions often have complex geometries that can be difficult to capture with global basis functions.

- Global approximation techniques include linear regression and neural network regression.
- Nonlinear utility functions can be obtained when using linear regression when combined with an appropriate selection of nonlinear basis functions.
- Neural network regression relieves us of having to specify basis functions, but fitting them is more complex and generally requires us to use gradient descent to tune our parametric approximation of the value function.

8.9 Exercises

Exercise 8.1. A tabular representation is a special case of linear approximate value functions. Show how, for any discrete problem, a tabular representation can be framed as a linear approximate value function.

Solution: Consider a discrete MDP with m states $s_{1:m}$ and n actions $a_{1:n}$. A tabular representation associates a value with each state or state-action pair. We can recover the same behavior using a linear approximate value function. We associate an indicator function with each state or state-action pair, whose value is 1 when the input is the given state or state-action pair and 0 otherwise:

$$\beta_i(s) = (s = s_i) = \begin{cases} 1 & \text{if } s = s_i \\ 0 & \text{otherwise} \end{cases}$$

or

$$\beta_{ij}(s, a) = ((s, a) = (s_i, a_j)) = \begin{cases} 1 & \text{if } (s, a) = (s_i, a_j) \\ 0 & \text{otherwise} \end{cases}$$

Exercise 8.2. Suppose we have a problem with continuous state and action spaces and that we would like to construct both a linear approximation and a global approximation of the action value function $Q(s, a) = \theta^\top \beta(s, a)$. For global approximation, we choose the basis functions

$$\beta(s, a) = \left[1, s, a, s^2, sa, a^2\right]$$

Given a set of one-hundred states $S = s_{1:100}$ and a set of five actions $A = a_{1:5}$, how many parameters are in θ for a local approximation method? How many parameters are in θ for the specified global approximation method?

Solution: In local approximation methods, the state-action values are the parameters. We will have $|S| \times |A| = 100 \times 5 = 500$ parameters in θ. In global approximation methods, the coefficients of the basis functions are the parameters. Since there are six components in $\beta(s, a)$, we will have six parameters in θ.

Exercise 8.3. We are given the following states $s_1 = (4, 5)$, $s_2 = (2, 6)$, and $s_3 = (-1, -1)$, and their corresponding values, $U(s_1) = 2$, $U(s_2) = 10$, and $U(s_3) = 30$. Compute the value at state $s = (1, 2)$ using 2-nearest neighbor local approximation with an L_1 distance metric, with an L_2 distance metric, and with an L_∞ distance metric.

Solution: We tabulate the distances from s to the points $s' \in S$ below:

$s' \in S$	L_1	L_2	L_∞
$s_1 = (4, 5)$	6	$\sqrt{18}$	3
$s_2 = (2, 6)$	5	$\sqrt{17}$	4
$s_3 = (-1, -1)$	5	$\sqrt{13}$	3

Using the L_1 norm, we estimate $U(s) = (10 + 30)/2 = 20$. Using the L_2 norm, we estimate $U(s) = (10 + 30)/2 = 20$. Using the L_∞ norm, we estimate $U(s) = (2 + 30)/2 = 16$.

Exercise 8.4. We would like to estimate the value at a state s given the values at a set of two states $S = \{s_1, s_2\}$. If we want to use local approximation value iteration, which of the following weighting functions are valid? If they are invalid, how could the weighting functions be modified to make them valid?

- $\boldsymbol{\beta}(s) = [1, 1]$
- $\boldsymbol{\beta}(s) = [1 - \lambda, \lambda]$ where $\lambda \in [0, 1]$
- $\boldsymbol{\beta}(s) = \left[e^{(s-s_1)^2}, e^{(s-s_2)^2}\right]$

Solution: The first set of weighting functions is not valid, as it violates the constraint $\sum_i \beta_i(s) = 1$. We can modify the weighting functions by normalizing them by their sum

$$\boldsymbol{\beta}(s) = \left[\frac{1}{1+1}, \frac{1}{1+1}\right] = \left[\frac{1}{2}, \frac{1}{2}\right]$$

The second set of weighting functions is valid. The third set of weighting functions is not valid, as it violates the constraint $\sum_i \beta_i(s) = 1$. We can modify the weighting functions by normalizing them by their sum

$$\boldsymbol{\beta}(s) = \left[\frac{e^{(s-s_1)^2}}{e^{(s-s_1)^2} + e^{(s-s_2)^2}}, \frac{e^{(s-s_2)^2}}{e^{(s-s_1)^2} + e^{(s-s_2)^2}}\right]$$

Exercise 8.5. Prove that bilinear interpolation is invariant under (nonzero) linear grid scaling.

Solution: It is straightforward to show that the interpolated value is invariant to a linear scaling on one or both axes, e.g. $\tilde{U}_\theta(\tilde{s}) = U_\theta(s)$. We show this by substituting all x- and y-values by their scaled versions $\tilde{x} = \beta x$ and $\tilde{y} = \gamma y$, and showing that the grid scalings cancel out:

$$\tilde{U}_\theta(\tilde{s}) = \frac{(\tilde{x}_2 - \tilde{x})(\tilde{y}_2 - \tilde{y})}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_1 + \frac{(\tilde{x}_2 - \tilde{x})(\tilde{y} - \tilde{y}_1)}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_2 + \frac{(\tilde{x} - \tilde{x}_1)(\tilde{y}_2 - \tilde{y})}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_3 + \frac{(\tilde{x} - \tilde{x}_1)(\tilde{y} - \tilde{y}_1)}{(\tilde{x}_2 - \tilde{x}_1)(\tilde{y}_2 - \tilde{y}_1)}\theta_4$$

$$\tilde{U}_\theta(\tilde{s}) = \frac{\beta(x_2 - x)\gamma(y_2 - y)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_1 + \frac{\beta(x_2 - x)\gamma(y - y_1)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_2 + \frac{\beta(x - x_1)\gamma(y_2 - y)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_3 + \frac{\beta(x - x_1)\gamma(y - y_1)}{\beta(x_2 - x_1)\gamma(y_2 - y_1)}\theta_4$$

$$\tilde{U}_\theta(\tilde{s}) = \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4$$

$$\tilde{U}_\theta(\tilde{s}) = U_\theta(s)$$

Exercise 8.6. Given the following four states $s_1 = [0, 5]$, $s_2 = [0, 25]$, $s_3 = [1, 5]$, and $s_4 = [1, 25]$, and a sample state $s = [0.7, 10]$, generate the interpolant equation $U_\theta(s)$ for arbitrary θ.

Solution: The general form for bilinear interpolation is given in equation (8.12) and reproduced below. To generate the interpolant, we substitute our values into the equation and simplify

$$U_\theta(s) = \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_1 + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_2 + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}\theta_3 + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}\theta_4$$

$$U_\theta(s) = \frac{(1 - 0.7)(25 - 10)}{(1 - 0)(25 - 5)}\theta_1 + \frac{(1 - 0.7)(10 - 5)}{(1 - 0)(25 - 5)}\theta_2 + \frac{(0.7 - 0)(25 - 10)}{(1 - 0)(25 - 5)}\theta_3 + \frac{(0.7 - 0)(10 - 5)}{(1 - 0)(25 - 5)}\theta_4$$

$$U_\theta(s) = \frac{9}{40}\theta_1 + \frac{3}{40}\theta_2 + \frac{21}{40}\theta_3 + \frac{7}{40}\theta_4$$

Exercise 8.7. Following example 8.1, what are the simplex interpolant weights for a state $\mathbf{s} = [0.4, 0.95, 0.6]$?

Solution: For the given state $\mathbf{s}$, we have $0 \leq x_1 \leq x_3 \leq x_2 \leq 1$, and so our permutation vector is $\mathbf{p} = [1, 3, 2]$. The vertices of our simplex can be generated by starting from $(0, 0, 0)$ and changing each 0 to a 1 in reverse order of the permutation vector. Thus, the vertices of the simplex are $(0, 0, 0)$, $(0, 1, 0)$, $(0, 1, 1)$, and $(1, 1, 1)$.

Any point $\mathbf{s}$ belonging to the simplex can thus be expressed by a weighting of the vertices

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = w_1 \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + w_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + w_3 \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} + w_4 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

We can determine the values of the weights in reverse order, finally solving for w_1 by applying the constraint that the weights must sum to one. We can then compute the weights for $\mathbf{s} = [0.4, 0.95, 0.6]$

$$w_4 = s_1 \qquad w_3 = s_3 - w_4 \qquad w_2 = s_2 - w_3 - w_4 \qquad w_1 = 1 - w_2 - w_3 - w_4$$
$$w_4 = 0.4 \qquad w_3 = 0.2 \qquad w_2 = 0.35 \qquad w_1 = 0.05$$