

Part II: Approximate Solution Methods In the second part, we are dealing with arbitrarily large state spaces; in these cases, it is hard to find π_* or v_*, q_* . Our goal instead is to find a good approximate solution using limited computational resources.

The problem with large state spaces is threefold: memory, time and data. In many of our target tasks, almost every state encountered will never have been seen before. The solution is *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

To some extent, we need only combine RL methods with existing generalization methods. One of frequent use is *function approximation*, which takes examples from a desired function (e.g. a value function) and attempts to generalize from them to construct an approximation of the entire function.

Function approximation is an instance of supervised learning.

RL with function approximation involves a number of new issues (usually not in SL), such as nonstationarity, bootstrapping, and delayed targets.

Chap 9 (prediction): on-policy training, policy is given and only its value function is approximated

Chap 10 (control): an approximation to the optimal policy is found

Chap 11: off-policy learning with function approximation

Chap 12: *eligibility traces*, which dramatically improves the computational properties of multi-step RL methods in many cases.

Chap 13 (control): *policy-gradient*, a different approach to control, which approximates the optimal policy directly and need never form an approximate value function.

1 On-policy Prediction with Approximation

We start consider using function approximation in estimating the state-value function from on-policy data, which approximates v_π from experience generated using a known policy π .

The approximate value function is represented as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$, instead of a table. We write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector \mathbf{w} .

For example,

- \hat{v} = linear function in features of the state, \mathbf{w} = vector of feature weights
- \hat{v} = function computed by a multi-layer ANN, \mathbf{w} = vector of connection weights in all the layers
- \hat{v} = function computed by a decision tree, \mathbf{w} = all the numbers defining the split points and leaf values of the tree

Typically, the number of weights is much less than the number of states ($d \ll |S|$), and one weight takes care of many states.

Extending RL to function approximation also makes it applicable to partially observable problems (in which the full state is not available to the agent).

All theoretical results for methods using function approximation apply equally well to cases of partial observability.

Function approximation cannot augment the state representation with memories of past observations.

1.1 Value-function Approximation

Definition 1.1 *individual update notation:* $s \rightarrow u$ (the book uses $|->$, where s is the state updated and u is the update target. The estimated value for s should be more like u).

- MC: $S_t \rightarrow G_t$
- TD(0): $S_t \rightarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$
- n-step TD: $S_t \rightarrow G_{t:t+n}$
- DP: $s \rightarrow \mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) | S_t = s]$

We want to extend a one-to-one mapping to one-to-many mapping, i.e., updating s will also update many other states.

Supervised learning methods learn to mimic input-output examples, and when the outputs are numbers, like u , the process is often called *function approximation*. We use these methods for value prediction simply by passing to them the $s \rightarrow u$ of each update as a training example.

In principle, we can use any method for supervised learning from examples, including ANN, decision trees, and various kinds of multivariate regression. However, in RL, we require the methods are able to handle nonstationarity cases as RL agents need to do online learning (interact with the environment to obtain data).

1.2 The Prediction Objective (\overline{VE})

By assumption we have far more states than weights, so making one state's estimate more accurate invariably (always) means making others' less accurate. We need to find which states we care most about.

Definition 1.2 state distribution $\mu(s)$: $\mu \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error (square distance between approx value $\hat{v}(s, \mathbf{w})$ and true value $v_\pi(s)$) in each state s .

Since we are doing on-policy policy evaluation, the μ is from policy π . We can also design μ , then it becomes off-policy prediction.

Definition 1.3 Mean Squared Value Error (\overline{VE}): a natural objective function,

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in S} \underbrace{\mu(s)}_{\text{weight}} \underbrace{[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2}_{\text{error}}$$

Often $\mu(s)$ is chosen to be the fraction of time spent in s . Under on-policy training, this is called *on-policy distribution*. In continuing tasks, the on-policy distribution is the stationary distribution under π . The book talks about the episodic tasks (not including here).

!! NOTE that the RL community is not sure if $\overline{V E}$ is the right performance objective for RL. But this is the best we can find so far. Adam White: VE might not be the best because we are only approximating the exact true values. Instead, we can use Average reward + Policy Gradient methods for control.

1.3 Stochastic-gradient and Semi-gradient Methods

In GD, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, \dots, w_d)^T$, and the approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of \mathbf{w} for all $s \in S$. However, no \mathbf{w} can get all the states in real world.

GD vs SGD In both GD and SGD, we update a set of parameters in an iterative manner to minimize an error function.

- GD: we run through ALL the samples to do a single update for a parameter in a particular iteration (which is not feasible)
- SGD: we run through ONLY ONE or a SUBSET of samples to do the update. If we use a SUBSET, it is called Minibatch Stochastic Gradient Descent.

The solution to \mathbf{w} is SGD: we minimize error on the *observed samples*. The weight vector of SGD is:

$$\mathbf{w}_{t+1} \doteq \mathbf{w} - \frac{1}{2} \alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2$$

$$\mathbf{w}_{t+1} = \mathbf{w} + \alpha [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

Now we consider when we do not have the exact true value $v_\pi(S_t)$, but an

approximate value U_t . Then

$$\mathbf{w}_{t+1} = \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\nabla\hat{v}(S_t, \mathbf{w}_t)$$

If U_t is an *unbiased* estimate ($\mathbb{E}[U_t|S_t = s] = v_\pi(s)$), then \mathbf{w}_t is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (chap2). For example, Monte Carlo target $U_t \doteq G_t$ is an unbiased estimate of $v_\pi(S_t)$.

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

Input: π to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \rightarrow \mathbb{R}$

Parameter: $\alpha > 0$

Init the value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

Loop forever (for each episode):

 Generate an episode $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using π

 Loop for each step of episode, $t = 0, 1, \dots, T - 1$:

 Take action A ; observe resultant reward, R , and state, S'

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

On the other hand, Bootstrapping targets, such as n -step returns $G_{t:t+n}$ or the DP target, all depend on the current value of the weight vector \mathbf{w}_t , which implies that they will be biased. These are not the true gradient-descent methods, but *semi*-gradient methods.

Although semi-gradient (bootstrapping) methods do not converge as robustly as true gradient methods, they offer important advantages: (1) enable much faster learning; (2) enable learning to be continual and online; (3) provides computational advantages. A typical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma\hat{v}(S_t, \mathbf{w})$

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

Input: π to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \rightarrow \mathbb{R}$ where $\hat{v}(\text{terminal}, \cdot) = 0$

Parameter: $\alpha > 0$

Init the value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

Loop for each episode:

 Init S

 Loop for each step of episode:

 Choose $A \sim \pi(\cdot|S)$

 Take action A ; observe reward, R , and state, S'

$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{v}(S_t, \mathbf{w}) - \hat{v}(S, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$

$S \leftarrow S'$

 until S is terminal

The target U_t in semi-gradient methods is biased, so the \mathbf{w} may not converge to a local optimum.

state aggregation a simple form of generalizing function approximation in which states are grouped together, with one estimated value for each group.

1.4 Linear Methods

One of the most important special cases of function approximation is that, $\hat{v}(\cdot; \mathbf{w})$ is a linear function of the weight vector \mathbf{w} . Corresponding to every state s , there is a real-valued vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \dots, x_d(s))^T$, with the same number of components as \mathbf{w} .

Linear methods approximate the state-value function by the inner product between \mathbf{w} and $\mathbf{x}(s)$:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s) \doteq \sum_{i=1}^d w_i x_i(s)$$

The vector $\mathbf{x}(s)$ is a *feature vector* representing state s . Each component $x_i(s)$ is a *feature*.

The gradient of the approximation value function w.r.t \mathbf{w} is then $\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$. Thus, we can simplify the general SGD update to

$$\mathbf{w}_{t+1} = \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(s)$$

Understand the fixed point of linear TD learning Let us start with TD update with linear function approximation on the first line, recall the value of a state $\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s)$

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(S_t) \\ &= \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}_t \quad \text{simplify notation} \\ &= \mathbf{w} + \alpha[R_{t+1} + \gamma \mathbf{w}_t^T \mathbf{x}_{t+1} - \mathbf{w}_t^T \mathbf{x}_t]\mathbf{x}_t \quad \text{due to } \hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s) \\ &= \mathbf{w} + \alpha[\underbrace{R_{t+1}\mathbf{x}_t}_{\mathbf{b}} - \underbrace{\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T}_{\mathbf{A}}\mathbf{w}_t] \quad \text{transpose doesn't change scalar} \end{aligned}$$

The TD update can be rewritten as the expected update + a noise term, so it is largely dominated by the behavior of the expected update.

$$\begin{aligned} \mathbb{E}[\Delta \mathbf{w}_t] &= \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t) \\ \mathbf{b} &= \mathbb{E}[R_{t+1}\mathbf{x}_t] \\ \mathbf{A} &= \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T] \quad \text{expct. over features} \end{aligned}$$

Definition 1.4 TD Fixed Point \mathbf{w}_{TD} , the weight is converged when the expected TD update = 0.

$$\mathbb{E}[\Delta \mathbf{w}_{TD}] = \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_{TD}) = 0$$

When \mathbf{A} is invertible, we can rewrite $\mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b}$. We say \mathbf{w}_{TD} is a solution to this linear system; the solution is called TD fixed point.

\mathbf{w}_{TD} minimizes an objective that is based on this \mathbf{A} and \mathbf{b} . This objective extends the connection between TD and Bellman equations to the function approximation setting.

Recall in the tabular setting, TD is described as a sample based method for solving the Bellman equation. Linear TD similarly approximates the solution to the Bellman equation, minimizing what is called the projected Bellman error.

Describe a theoretical guarantee on the mean squared value error at the TD fixed point We want to understand the relationship between the solution found by TD and the minimum value error solution.

$$\overline{VE}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{VE}(\mathbf{w})$$

The difference between these two errors depends on γ and the quality of features.

Why isn't the TD fixed point error equal to the minimum value error solution? This is because bootstrapping under function approximation. If our function approximator is good, then our estimate of the next state will be accurate.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: π to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \rightarrow \mathbb{R}$ where $\hat{v}(\text{terminal}, \cdot) = 0$

Parameter: $\alpha > 0, n > 0$

Init the value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily

All store and access operations (S_t and R_t) can their index mod $n + 1$

Loop for each episode:

 Init and store $S_0 \neq \text{terminal}$

$T \leftarrow \text{inf}$

 Loop for $t = 0, 1, 2, \dots$:

 If $t < T$, then:

 Take an action according to $\pi(\cdot | S_t)$

 Observe and store R_{t+1} and S_{t+1}

 If S_{t+1} is terminal, then $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)

 If $\tau \geq 0$:

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$

 If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_\tau, \mathbf{w})] \nabla \hat{v}(S_\tau, \mathbf{w})$

 until $\tau = T - 1$

1.5 Feature Construction for Linear Methods

Choosing features appropriate to the task is an important way of adding *prior domain knowledge* to reinforcement learning systems.

Linear methods,

- pro: guarantee convergence, data efficient, computation efficient
- con: does not consider interactions between features

1.5.1 Polynomials

1.5.2 Fourier Basis

1.5.3 Coarse Coding

The features used to construct value estimates, are one of the most important parts of a RL agent.

Recall linear value function approximation $v_{\pi}(s) \approx \hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$. Recall that a tabular representation can be expressed as a binary (0 or 1) feature vector. Each state is associated with a different feature. Agent-presented state is 1, all other features are 0. The tabular case is a special case of linear function approximation, where the feature vector is an indicator/one-hot encoding of the state.

This becomes a problem when the feature space is large. Recall that we can use state aggregation to associate nearby states with the same features.

From state aggregation to coarse coding: although state aggregation can have arbitrary shapes, but it does not allow overlapping. We can remove this restriction, and we have coarse coding. Therefore, coarse coding is a generalization of state aggregation.

Definition 1.5 *binary feature*: 1-0 valued feature, 1 for present, 0 for absent.

Definition 1.6 *coarse coding*: representing a state with binary features that overlap.

Changing the shapes and sizes of features impacts generalization and discrimination, and so affects the speed of learning and the value functions we can represent.

Coarse coding parameters affect discrimination and generalization, which in turn affect learning accuracy.

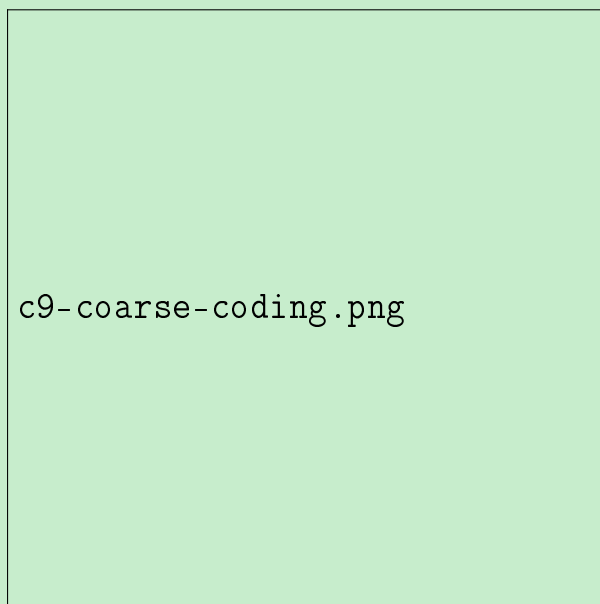


Figure 1: Coarse coding. Generalization from s to s' depend on the number of their features whose receptive fields overlap

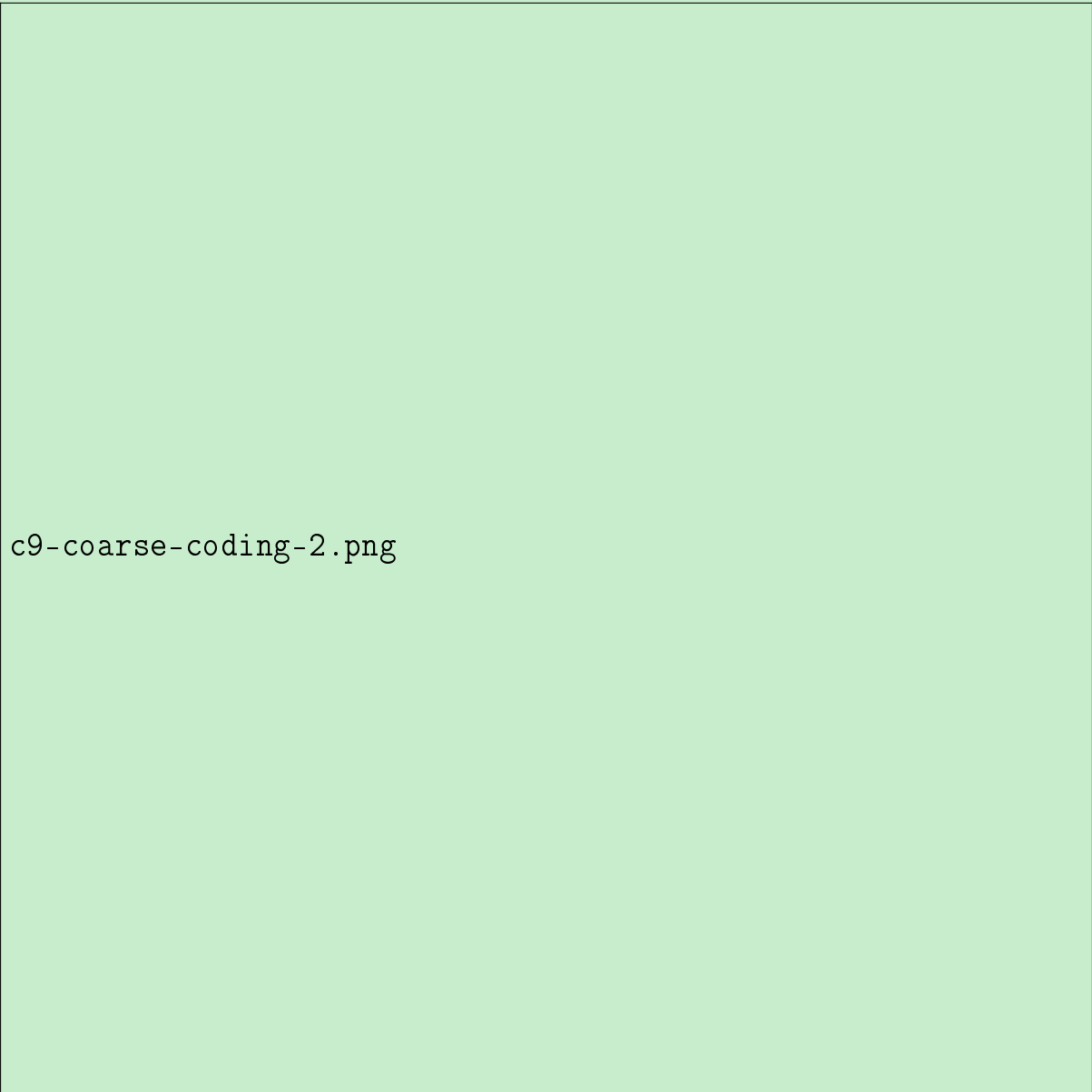


Figure 2: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

1.5.4 Tile Coding

Definition 1.7 *Tile coding*, a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient.

This may be the most practical feature representation for modern sequential digital computers.

In the coding, the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. The tiles of a partition do not overlap.

Recall one tiling is just like state aggregation. Tile coding = multiple overlapped state aggregation. To improve discrimination ability, we use multiple tilings so that we have many small intersections.


To get the strength of coarse coding, we need overlapping receptive fields. The advantages of tile coding

- the total active features at one time is the same for any state
- computational advantages from its use of binary feature vectors (just sum all active features)
- save memory

The choice of how to offset the tilings from each other affects generalization. Tile asymmetrical offsets are preferred. If the tilings are uniformly offset, then there are diagonal artifacts and substantial variations in the generalization, whereas with asymmetrically offset tilings the generalization is more spherical and homogeneous.

Extensive studies have been made of the effect of different displacement vectors on the generalization of tile coding. In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles.

Another useful trick of reducing memory requirements is *hashing* (Like one-to-many mapping).



c9-tile-coding.png

Figure 3: Multiple, overlapping grid-tilings on a limited 2D space. The tilings are offset from one another by a uniform amount in each dimension.

1.6 Selecting Step-Size Parameters Manually

1.7 Nonlinear Function Approximation: Artificial Neural Networks

!! Note that neural network, like coarse coding/tile coding, is also a way of feature representation. State aggregation, coarse coding and tile coding are fixed representation. NN are flexible representation (due to learning)

This section is like a summary of the NN course.

ANN is a network of interconnected units that have some of the properties of neurons.

A feedforward ANN has no loops in its network, i.e., there are no paths within the network by which a unit's output can influence its input. If an ANN has at least one loop in its connections, it is a recurrent rather than a feedforward ANN.

Hidden layers: layers that are neither input/output layers.

A real-valued weight is associated with each link.

The units in ANN are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function (activation function). **Nonlinearity is essential:** if all the units have linear activation functions, the entire network is equivalent to a network without hidden layers (linear+linear = linear).

An ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy.

Training the hidden layers of an ANN is a way to auto create features appropriate for a given problem so that hierarchical representations can be produced w/o relying exclusively on hand-crafted features.

ANN typically learn by a stochastic gradient method. Each weight is adjusted in a direction aimed at improving the network's overall performance as measured by an objective function to be either min/maximized (that is, to estimate the partial derivative of an objective function w.r.t each weight, given the current values of all network's weights).

Backpropagation algorithm consists of alternating forward and backward passes through the network. (In later chapter, we discuss train ANN with RL instead of backpropagation).

Backpropagation may not work well for deep ANN. Because (1) the large number of weights can lead to overfitting; (2) the partial derivatives computed by its backward passes either decay rapidly (slow learning) or grow rapidly (unstable learning).

Overfitting is a problem for any function approximation method that adjusts functions with many degrees of freedom on the basis of limited training data.

Methods for reducing overfitting / improve training

- cross validation: stop training when performance begins to decrease on validation data different from the training data
- regularization: modifying the objective function to discourage complexity of the approximation
- weight sharing: introducing dependencies among the weights to reduce the number of degrees of freedom
- dropout: during training, units are randomly removed from the network along with their connections
- deep belief networks: deepest layers are trained first using unsupervised learning
- batch normalization: normalizes the output of dep layers before they feed into the following layer
- deep residual learning: sometimes it is easier to learn a function differs from the identify function than to learn the function itself (by adding shortcut, or skip connections)
- deep convolution neural network

Compute the gradient of a single hidden layer neural network

- step 1: define a loss on the parameters of the NN
- step 2: find the parameters which minimize this loss function

Suppose we have input layer s , hidden layer x , output layer y , input-hidden weight A , hidden-output weight B .

- $\mathbf{x} \doteq f_A(\boldsymbol{\psi}), \boldsymbol{\psi} \doteq \mathbf{s}\mathbf{A}$
- $\hat{\mathbf{y}} \doteq f_B(\boldsymbol{\theta}), \boldsymbol{\theta} = \mathbf{x}\mathbf{B}$

Gradient w.r.t \mathbf{B} :

$$\begin{aligned}
\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{B}_{jk}} &= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{B}_{jk}} \\
&= \frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_B(\theta_k)}{\partial \theta_k} \frac{\partial \theta_k}{\partial \mathbf{B}_{jk}} \\
&= \underbrace{\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \hat{\mathbf{y}}_k} \frac{\partial f_B(\theta_k)}{\partial \theta_k}}_{\delta_k^B} \mathbf{x}_j
\end{aligned}$$

Note that the weight \mathbf{A} also affect \mathbf{x} , so a chain rule set up is needed here.

Gradient w.r.t \mathbf{A} :

$$\begin{aligned}
\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{A}_{ij}} &= \delta_k^B \frac{\partial \theta_k}{\partial \mathbf{A}_{ij}} & \frac{\partial \theta_k}{\partial \mathbf{A}_{ij}} &= \mathbf{B}_{jk} \frac{\partial \mathbf{x}_j}{\partial \mathbf{A}_{ij}} \\
&= \delta_k^B \mathbf{B}_{jk} \frac{\partial \mathbf{x}_j}{\partial \mathbf{A}_{ij}} & \frac{\partial \mathbf{x}_j}{\partial \mathbf{A}_{ij}} &= \frac{\partial f_A(\psi_j)}{\partial \psi_j} \frac{\partial \psi_j}{\partial \mathbf{A}_{ij}} \\
&= \delta_k^B \mathbf{B}_{jk} \frac{\partial f_A(\psi_j)}{\partial \psi_j} \frac{\partial \psi_j}{\partial \mathbf{A}_{ij}} \\
&= \underbrace{\delta_k^B \mathbf{B}_{jk} \frac{\partial f_A(\psi_j)}{\partial \psi_j}}_{\delta_j^A} \mathbf{s}_i
\end{aligned}$$

Notice that both gradients can be rewritten in a similar form. They have a term δ that contains an error signal times their input.

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{A}_{ij}} = \delta_j^A \mathbf{s}_i$$

$$\frac{\partial L(\hat{\mathbf{y}}_k, \mathbf{y}_k)}{\partial \mathbf{B}_{jk}} = \delta_k^B \mathbf{x}_j$$

one-hidden layer backprop algorithm

for each i/o pair (s, y) in dataset:

$$\delta_k^B = \frac{\partial L(\hat{y}_k, y_k)}{\partial \hat{y}_k} \frac{\partial f_B(\theta_k)}{\partial \theta_k}$$

$$\nabla_B^{jk} = \delta_k^B x_j$$

$$B = B - \alpha_B \nabla_B$$

$$\delta_j^A = \delta_k^B B_{jk} \frac{\partial f_A(\psi_j)}{\partial \psi_j}$$

$$\nabla_j^A = \delta_j^A s_i$$

$$A = A - \alpha_A \nabla_A$$

First get prediction \hat{y} (forward pass), then compute the gradients backwards from the output. Specifically, we first compute δ_B and the gradient for B , then we use this gradient to update the parameters B , the step size α_B . Next we update the parameters A , we compute δ_A which uses δ_B . Notice that by computing the gradients backwards, we avoid re-computing the same terms; In fact, this is the main idea behind backpropagation: it is simple GD with efficient strategy to compute gradients.

1.8 Learning Objectives (UA RL MOOC)

Lesson 1: Estimating Value Functions as Supervised Learning

1. Understand how we can use parameterized functions to approximate value functions

$$\hat{v}(s, \mathbf{w})$$

2. Explain the meaning of linear value function approximation

$$\hat{v}(s, \mathbf{w}) = \sum w_i x_i(s)$$

3. Recognize that the tabular case is a special case of linear value function approximation.

Consider each state has a corresponding vector (one-hot encoding)

4. Understand that there are many ways to parameterize an approximate

value function

ANN, decision tree etc

5. Understand what is meant by generalization and discrimination

generalization: updates to the value estimate of one state influence the value of other states

discrimination: the ability to make the values for two states different to distinguish between the values for these two state.

6. Understand how generalization can be beneficial

Generalization can speed up learning.

7. Explain why we want both generalization and discrimination from our function approximation

Because we want generalization to speed up learning, and discrimination to distinguish really different states.

8. Understand how value estimation can be framed as a supervised learning problem

Supervised learning involves approximating a function given a dataset of (input, target) pairs. For Monte-Carlo (S_t, G_t) . For TD $(S_t, R_t + \gamma \hat{v}(S_t, \mathbf{w}))$

9. Recognize not all function approximation methods are well suited for reinforcement learning

Some are not suitable because they are designed for a fixed batch of data (offline learning, contrast to RL's online learning), or not designed for temporally correlated data (data in RL is always correlated).

Lesson 2: The Objective for On-policy Prediction

10. Understand the mean-squared value error objective for policy evaluation

We cannot guarantee perfect approximation for every state's value, so we need to define an objective, a measure of the distance between our approximation

and the true values.

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \underbrace{\mu(s)}_{\text{weight}} \underbrace{[v_{\pi}(s) - \hat{v}(s, \mathbf{w})]}_{\text{error}}^2$$

11. Explain the role of the state distribution in the objective

$\mu(s)$ specifies how much we care for each state.

- It is a probability distribution
- It has higher values for states that are visited more often
- It serves as a weighting to minimize the error more in states that we care about

12. Understand the idea behind gradient descent and stochastic gradient descent

We use SGD to optimize the objective \overline{VE} . (1) the gradient indicates the direction to increase/decrease the weight vectors; (2) the gradient gives the direction of steepest ascent

13. Outline the gradient Monte Carlo algorithm for value estimation

see chap 9.3

14. Understand how state aggregation can be used to approximate the value function

Group multiple states into one grouped state. One feature for each grouped state.

15. Apply Gradient Monte-Carlo with state aggregation

see chap 9.3

Lesson 3: The Objective for TD

16. Understand the TD-update for function approximation

17. Highlight the advantages of TD compared to Monte-Carlo

TD enables faster learning, continual and online learning; provides compu-

tational advantages

18. Outline the Semi-gradient TD(0) algorithm for value estimation

see chap 9.3; semi-gradient TD as an approximation to SGD.

19. Understand that TD converges to a biased value estimate

The TD target depends on our estimate of the value in the next state. This means our update could be biased because the estimate in our target may be inaccurate.

20. Understand that TD converges much faster than Gradient Monte Carlo

TD can learn during the episode and has a lower variance update.

Lesson 4: Linear TD

21. Derive the TD-update with linear function approximation

$$\mathbf{w}_{t+1} = \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(s)$$

22. Understand that tabular TD(0) is a special case of linear semi-gradient TD(0)

linear TD is a strict generalization on both tabular TD and TD with state aggregation. Recall the feature vector is like one-hot encoded vector, where the i-th=1, else 0 corresponds to i-th weight.

23. Highlight the advantages of linear value function approximation over nonlinear

- linear methods are simpler to understand and analyze mathematically
- with good features, linear methods can learn quickly and achieve good prediction accuracy

linear semi-gradient TD (aka TD with linear function approximation)

Lesson 1: Feature Construction for Linear Methods

24. Describe the difference between coarse coding and tabular representations

Tabular states can be represented with a binary one-hot encoding. Tabular representations become infeasible when state space is large (a state associated with a feature). We then introduced state aggregation to resolve this, where one feature represents many states. Coarse coding is a further generalization of state aggregation, which allows overlapping of features (receptive fields).

25. Explain the trade-off when designing representations between discrimination and generalization

The size, shape, and number of features (receptive fields) affect the generalization and discrimination, thus affecting learning accuracy. For example, larger circle can generalize better, but discrimination is worse.

26. Understand how different coarse coding schemes affect the functions that can be represented

Recall the step-function approximation in the textbook.

27. Explain how tile coding is a (computationally?) convenient case of coarse coding

Since grid is uniform, it's easy to compute which cell the current state is in.

28. Describe how designing the tilings affects the resultant representation

The symmetry/asymmetry offsets affect tile coding's generalization and discrimination.

29. ...

Lesson 2: Neural Networks

30. Define a neural network

- NN = input layer + hidden layer(s) + output layer
- layer = node(s); layers are linked, each link is a real-valued weight
- node contains non-linear function (active function)

31. Define activation functions

activation function = non-linear function (Sigmoid, ReLU, step function)

32. Define a feed-forward architecture

A network without link from output to input.

33. Understand how neural networks are doing feature construction

Tile coding has fixed parameters before learning, NN has both fixed and learned parameters. So NN can use both prior knowledge and knowledge from data to do feature construction.

34. Understand how neural networks are a non-linear function of state

Non-linear activation function resulting in a non-linear function of the inputs.

35. Understand how deep networks are a composition of layers

We can have many hidden layers (1) to allow composition of features. Composition can produce more specialized features by combining modular components; (2) to obtain abstractions, DNN compose many layers of lower-level of abstractions with each successive layer contributing to increasingly abstract representation.

36. Understand the trade-off between learning capacity and challenges presented by deeper networks

Challenges mostly from overfitting and training efficiency.

Lesson 3: Training Neural Networks

37. Understand the importance of initialization for neural networks

If we choose a bad init parameters, we either have zero gradient or stuck in local optima.

38. Describe strategies for initializing neural networks

Method I: randomly sample the initial weights from a normal distribution with small variance, and normalize the weights.

$$\mathbf{w}_{init} \sim \frac{N(0, 1)}{\sqrt{n_{inputs}}}$$

This way, each neuron has a different output from other neurons within its layer. This provides a more diverse set of potential features. By keeping the variants small, we ensure that the output of each neuron is within the same range as its neighbors. By normalizing the weights, we prevent larger variance as number of neurons grows.

Adam Algorithm are Method II + Method III. **Method II:** Update Momentum M

$$\begin{aligned}\mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} L(\mathbf{w}_t) + \lambda M_t \\ M_{t+1} &\leftarrow \lambda M_t - \alpha \nabla_{\mathbf{w}} L\end{aligned}$$

The momentum term summarizes the history of the gradients using a decaying sum of gradients with decay rate λ . If recent updates in the same direction, then we have large momentum, making larger step; if recent updates in the opposite direction, then we kill the momentum.

Method III: vector step sizes

Instead of global step size, this uses different step sizes for different weights.

39. Describe optimization techniques for training neural networks

cross validation, regularization, dropout, residual learning etc