

A Guided Tour of Chapter 4: Function Approximation and Approximate DP

Ashwin Rao

ICME, Stanford University

From DP to Approximate ADP (abbrev. ADP)

- Dynamic Programming algorithms meant for non-large finite spaces
- DP algorithms typically sweep through all states in each iteration
- Cannot do this for large finite spaces or for infinite spaces
- Requires us to generalize to function approximation of Value Function
 - Sample an appropriate subset of states
 - Calculate the Value Function for those states (Bellman calculation)
 - Create/Update a func approx with the sampled states' calculated values
- Also, can sample transitions to estimate DP algo's Bellman update
- The fundamental structure of the algorithms is still the same
- Fundamental principles (Fixed-Point/Bellman Operators) still same
- These generalizations known as *Approximate Dynamic Programming*

Theory of Function Approximations

- We work with a generic but simple setting for Function Approximation
- Predictor variable $x \in \mathcal{X}$ (generic domain), Response variable $y \in \mathbb{R}$
- We treat x, y as unknown random variables, and want to estimate a function approximation for $\mathbb{P}[y|x]$ from data in the form of (x, y) pairs
- We consider parameterized functions f with parameters denoted w
- Exact data type of w will depend on specific form of function approx
- Denote the estimated probability of $y|x$ as $f(x; w)(y)$
- Assume given data in the form of a sequence of n (x, y) pairs:

$$[(x_i, y_i) | 1 \leq i \leq n]$$

- Estimating $\mathbb{P}[y|x]$ is formalized by solving for $w = w^*$ such that:

$$w^* = \arg \max_w \left\{ \prod_{i=1}^n f(x_i; w)(y_i) \right\} = \arg \max_w \left\{ \sum_{i=1}^n \log f(x_i; w)(y_i) \right\}$$

Maximum Likelihood and Cross-Entropy Loss

- This is the framework of *Maximum Likelihood Estimation* of $y|x$
- Data $[(x_i, y_i) | 1 \leq i \leq n]$ specifies *empirical probability distribution* D
- Parameterized function f specifies *model probability distribution* M
- So we are in the business of reconciling D and M
- So this is minimizing *Cross-Entropy Loss* between D and M

$$\text{Cross-Entropy Loss } \mathcal{H}(D, M) = -\mathbb{E}_D[\log M]$$

- We want to allow for incremental estimation (with data at each t):

$$[(x_{t,i}, y_{t,i}) | 1 \leq i \leq n_t]$$

- Parameters update from w_{t-1} to w_t with say gradient descent
- Allow for full batch, mini-batch or single pair (eg: SGD)
- With an estimate of $f(x; w)$, we can predict $y|x$ as $\mathbb{E}_M[y|x]$:

$$\mathbb{E}_M[y|x] = \mathbb{E}_{f(x;w)}[y] = \int_{-\infty}^{+\infty} y \cdot f(x; w)(y) \cdot dy$$

The @abstractclass FunctionApprox

```
class FunctionApprox(ABC, Generic[X]):  
  
    @abstractmethod  
    def solve(  
        self ,  
        xy_vals_seq: Iterable[Tuple[X, float]] ,  
        error_tolerance: Optional[float] = None  
    ) -> FunctionApprox[X]:  
  
    @abstractmethod  
    def evaluate(  
        self ,  
        x_values_seq: Iterable[X]  
    ) -> np.ndarray:
```

The @abstractclass FunctionApprox

```
def update(  
    self ,  
    xy_vals_seq: Iterable[Tuple[X, float]]  
) -> FunctionApprox[X]:  
    pass
```

```
def iterate_updates(  
    self ,  
    xy_seq_stream: Iterator[Iterable[Tuple[X, float]]]  
) -> Iterator[FunctionApprox[X]]:  
    return iterate.accumulate(  
        xy_seq_stream ,  
        lambda fa , xy: fa.update(xy) ,  
        initial=self  
    )
```

Linear Function Approximation

- Define a sequence of feature functions $\phi_j : \mathcal{X} \rightarrow \mathbb{R}, j = 1, 2, \dots, m$

Feature Vector $\phi(x) = (\phi_1(x), \phi_2(x), \dots, \phi_m(x))$ for all $x \in \mathcal{X}$

- Parameters w is a weights vector $\mathbf{w} = (w_1, w_2, \dots, w_m) \in \mathbb{R}^m$
- Linear function approximation assumes gaussian distribution for $y|x$

with mean $= \sum_{j=1}^m \phi_j(x) \cdot w_j = \phi(x)^T \cdot \mathbf{w}$ and constant variance σ^2

$$\mathbb{P}[y|x] = f(x; \mathbf{w})(y) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(y - \phi(x)^T \cdot \mathbf{w})^2}{2\sigma^2}}$$

- Regularized cross-entropy loss function for data $[x_i, y_i | 1 \leq i \leq n]$:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n} \cdot \sum_{i=1}^n (\phi(x_i)^T \cdot \mathbf{w} - y_i)^2 + \frac{1}{2} \cdot \lambda \cdot \|\mathbf{w}\|^2$$

- This ignores constants involving σ , and λ is regularization coefficient
- So $\mathcal{L}(\mathbf{w})$ is just MSE of linear predictions $\phi(x_i)^T \cdot \mathbf{w}$

Linear Function Approximation with Gradient Descent

- Gradient of $\mathcal{L}(\mathbf{w})$ with respect to \mathbf{w} works out to:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n} \cdot \left(\sum_{i=1}^n \phi(x_i) \cdot (\phi(x_i)^T \cdot \mathbf{w} - y_i) \right) + \lambda \cdot \mathbf{w}$$

- Solve for \mathbf{w}^* by incremental estimation using gradient descent
- Gradient estimate $\mathcal{G}_{(x_t, y_t)}(\mathbf{w}_t)$ for time t -data $[(x_{t,i}, y_{t,i}) | 1 \leq i \leq n_t]$:

$$\mathcal{G}_{(x_t, y_t)}(\mathbf{w}_t) = \frac{1}{n} \cdot \left(\sum_{i=1}^{n_t} \phi(x_{t,i}) \cdot (\phi(x_{t,i})^T \cdot \mathbf{w}_t - y_{t,i}) \right) + \lambda \cdot \mathbf{w}_t$$

- Interpreted as the weighted-mean of the feature vectors $\phi(x_{t,i})$
- Weighted by the (scalar) linear prediction errors $\phi(x_{t,i})^T \cdot \mathbf{w}_t - y_{t,i}$
- So the update to the weights vector \mathbf{w} is given by:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t \cdot \mathcal{G}_{(x_t, y_t)}(\mathbf{w}_t)$$

Direct Solution of Linear Function Approximation

- If feature functions not too large, we can directly solve for \mathbf{w}^*
- Assume the entire provided data is $[(x_i, y_i) | 1 \leq i \leq n]$
- Then the gradient estimate based can be set to 0 to solve for \mathbf{w}^*

$$\frac{1}{n} \cdot \left(\sum_{i=1}^n \phi(x_i) \cdot (\phi(x_i)^T \cdot \mathbf{w}^* - y_i) \right) + \lambda \cdot \mathbf{w}^* = 0$$

- Denote Φ as $n \times m$ matrix: $\Phi_{i,j} = \phi_j(x_i)$
- Denote column vector $\mathbf{Y} \in \mathbb{R}^n$ defined as $\mathbf{Y}_i = y_i$

$$\frac{1}{n} \cdot \Phi^T \cdot (\Phi \cdot \mathbf{w}^* - \mathbf{Y}) + \lambda \cdot \mathbf{w}^* = 0$$

$$\Rightarrow (\Phi^T \cdot \Phi + n\lambda \cdot I_m) \cdot \mathbf{w}^* = \Phi^T \cdot \mathbf{Y}$$

$$\Rightarrow \mathbf{w}^* = (\Phi^T \cdot \Phi + n\lambda \cdot I_m)^{-1} \cdot \Phi^T \cdot \mathbf{Y}$$

Deep Neural Networks (of vanilla flavor)

- Deep Neural Network (DNN) layers numbered $l = 0, 1, \dots, L$
- Denote input and output to layer l as vectors \mathbf{I}_l and \mathbf{O}_l

$$\mathbf{I}_0 = \phi(x) \in \mathbb{R}^m \text{ and } \mathbf{O}_L = \mathbb{E}_M[y|x] \text{ and } \mathbf{I}_{l+1} = \mathbf{O}_l$$

- Denote layer l parameters as $|\mathbf{O}_l| \times |\mathbf{I}_l|$ matrix \mathbf{w}_l
- Layer l neurons define a linear transformation from \mathbf{I}_l to \mathbf{S}_l

$$\mathbf{S}_l = \mathbf{w}_l \cdot \mathbf{I}_l \text{ and } \mathbf{O}_l = g_l(\mathbf{S}_l)$$

- where $g_l : \mathbb{R} \rightarrow \mathbb{R}$ is the activation function for layer l
- Forward-propagation composes layers' linear and activation functions
- Back-propagation calculates cross-entropy loss gradient $\nabla_{\mathbf{w}_l} \mathcal{L}$
- Gradient Descent updates for weights \mathbf{w}_l proportional to $\nabla_{\mathbf{w}_l} \mathcal{L}$

Back-prop as Recursive Gradient Calculation

- Loss gradient can be reduced to calculating $\mathbf{P}_I = \nabla_{\mathbf{S}_I} \mathcal{L}$

$$\nabla_{\mathbf{w}_I} \mathcal{L} = (\nabla_{\mathbf{S}_I} \mathcal{L})^T \cdot \nabla_{\mathbf{w}_I} \mathbf{S}_I = \mathbf{P}_I^T \cdot \nabla_{\mathbf{w}_I} \mathbf{S}_I = \mathbf{P}_I \cdot \mathbf{I}_I^T = \mathbf{P}_I \otimes \mathbf{I}_I$$

- Including L^2 regularization (with regularization coefficients λ_I):

$$\nabla_{\mathbf{w}_I} \mathcal{L} = \mathbf{P}_I \cdot \mathbf{I}_I^T + \lambda_I \cdot \mathbf{w}_I$$

- \cdot is inner-product, \otimes is outer-product, \circ is component-wise product

Theorem

$$\mathbf{P}_I = (\mathbf{w}_{I+1}^T \cdot \mathbf{P}_{I+1}) \circ g'_I(\mathbf{S}_I) \text{ (read the proof in the book)}$$

To calculate $\mathbf{P}_L = \nabla_{\mathbf{S}_L} \mathcal{L}$, assume suitable functional form for $\mathbb{P}[y|S_L]$

Exponential functional form for $\mathbb{P}[y|S_L]$

- Consider the exponential-family functional-form for $\mathbb{P}[y|S_L]$

$$\mathbb{P}[y|S_L] = p(y|S_L, \tau) = h(y, \tau) \cdot e^{\frac{S_L \cdot y - A(S_L)}{d(\tau)}}$$

- Form adopted from framework of Generalized Linear Models (GLM)
- We want the scalar prediction $O_L = g_L(S_L)$ to be equal to $\mathbb{E}_p[y|S_L]$
- What function $g_L : \mathbb{R} \rightarrow \mathbb{R}$ (in terms of $p(y|S_L, \tau)$) would satisfy the requirement of $O_L = g_L(S_L) = \mathbb{E}_p[y|S_L]$?

Lemma

$$\mathbb{P}[y|S_L] = h(y, \tau) \cdot e^{\frac{S_L \cdot y - A(S_L)}{d(\tau)}} \Rightarrow \mathbb{E}_p[y|S_L] = A'(S_L)$$

- To satisfy $O_L = g_L(S_L) = \mathbb{E}_p[y|S_L]$, we need: $O_L = g_L(S_L) = A'(S_L)$
- So $g_L(\cdot)$ must be set to be the derivative of the $A(\cdot)$ function
- In GLM theory, $A'(\cdot)$ serves as *canonical link function* for given $\mathbb{P}[y|x]$

Examples of Distributions and their Canonical Links

With canonical link, P_L reduces to prediction error for each (x, y) data

Theorem

$$P_L = \frac{\partial \mathcal{L}}{\partial S_L} = \frac{O_L - y}{d(\tau)}$$

Some examples of distributions and their canonical link functions:

- Normal distribution $y \sim \mathcal{N}(\mu, \sigma^2)$:

$$S_L = \mu, \tau = \sigma, h(y, \tau) = \frac{-y^2}{\sqrt{2\pi\tau}}, A(S_L) = \frac{S_L^2}{2}, d(\tau) = \tau^2. \quad g_L(S_L) = S_L$$

- Bernoulli distribution for binary-valued y , parameterized by p :

$$S_L = \log\left(\frac{p}{1-p}\right), \tau = h(y, \tau) = d(\tau) = 1, A(S_L) = \log(1 + e^{S_L}).$$
$$g_L(S_L) = \frac{1}{1 + e^{-S_L}}$$

- Poisson distribution for y parameterized by λ :

$$S_L = \log \lambda, \tau = d(\tau) = 1, h(y, \tau) = \frac{1}{y!}, A(S_L) = e^{S_L}. \quad g_L(S_L) = e^{S_L}$$

Tabular as a form of *FunctionApprox*

- “Tabular” is simple setting with finite $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$
- With (x, y) data pairs having all its x -values within this finite \mathcal{X}
- $\mathbb{E}[y|x]$ must be calculated from data y -values associated with single x
- So $\mathbb{E}[y|x]$ prediction must be *some sort of average* of those y -values
- A “table” can store all \mathcal{X} together with all predictions $\mathbb{E}[y|x]$
- This “Tabular” setting is compatible with *FunctionApprox* interface
- Also, “Tabular” is a special case of linear function approximation
- With features ϕ_i as indicator functions for each $x_i \in \mathcal{X}$
- And weights w_i as average of y -values associated with x_i in the data
- Next we cover Approximate DP using *FunctionApprox*
- Where \mathcal{X} is state space and predictions constitute Value Function
- Specializing *FunctionApprox* to Tabular gives Tabular DP

Approximate Policy Evaluation

- Repeatedly apply \mathbf{B}^π on FunctionApprox of $V : \mathcal{N} \rightarrow \mathbb{R}$
- Operates on MarkovRewardProcess (not necessarily Finite)
- So no enumeration of states and no access to transition probabilities
- We specify a sampling probability distribution of “source states”
- From each source sample s , sample pairs of (next state s' , reward r)
- Estimate $\mathbb{E}[r + \gamma \cdot V(s')]$ by averaging over sampled pairs
- $V(s')$ obtained from the instance of FunctionApprox being used
- Sample of source states and their associated Bellman expectation estimates (from transition samples) used to update FunctionApprox

@abstractclass Distribution interface

```
class Distribution(ABC, Generic[A]):
```

```
    @abstractmethod
```

```
    def sample(self) -> A:
```

```
        pass
```

```
    @abstractmethod
```

```
    def expectation(
```

```
        self,
```

```
        f: Callable[[A], float]
```

```
    ) -> float:
```

```
        pass
```


Approximate Policy Evaluation interface

```
ValueFunctionApprox = FunctionApprox[NonTerminal[S]]  
NTStateDistribution = Distribution[NonTerminal[S]]
```

```
def evaluate_mrp(  
    mrp: MarkovRewardProcess[S],  
    gamma: float,  
    approx_0: ValueFunctionApprox[S],  
    nt_states_distribution: NTStateDistribution[S],  
    num_samples: int  
) -> Iterator[ValueFunctionApprox[S]]:
```

Approximate Policy Evaluation code

```
def update(v: ValueFunctionApprox[S]) -> \
    ValueFunctionApprox[S]:
    nt_states: Sequence[NonTerminal[S]] = \
        nt_states_distribution.sample_n(num_samples)

    def return_(s_r: Tuple[State[S], float]) -> float:
        s1, r = s_r
        return r + gamma * extended_vf(v, s1)

    return v.update(
        [(s, mrp.transition_reward(s).expectation(
            return_)) for s in nt_states]
    )

return iterate(update, approx_0)
```

Approximate Value Iteration interface

```
def value_iteration(  
    mdp: MarkovDecisionProcess[S, A],  
    gamma: float,  
    approx_0: ValueFunctionApprox[S],  
    nt_states_distribution: NTStateDistribution[S],  
    num_samples: int  
) -> Iterator[ValueFunctionApprox[S]]:
```

Approximate Value Iteration code

```
def update(v: ValueFunctionApprox[S]) -> \
    ValueFunctionApprox[S]:
    nt_states: Sequence[NonTerminal[S]] = \
        nt_states_distribution.sample_n(num_samples)

    def return_(s_r: Tuple[State[S], float]) -> float:
        s1, r = s_r
        return r + gamma * extended_vf(v, s1)

    return v.update(
        [(s, max(mdp.step(s, a).expectation(return_)
            for a in mdp.actions(s)))
         for s in nt_states]
    )
return iterate(update, approx_0)
```

Finite-Horizon Approximate Dynamic Programming

- Similarly, generalize Backward Induction DP algorithms
- Each time steps' Value Function is a FunctionApprox
- Work with a separate MRP/MDP representation for each time step's transitions, that is responsible for sampling next step's (state, reward)
- x -values come from current time step's states sampling distribution
- y -values come from applying Bellman Operator on next time steps' FunctionApprox for it's Value Function
- Bellman Operator expectation is estimated by averaging over transition samples
- These (x, y) pairs constitute the data-set used to solve the current time step's FunctionApprox for it's Value Function

Constructing the Non-Terminal States Distribution

- Each ADP algorithm works with a distribution of non-terminal states
- Good choice: Stationary Distribution of uniform-policy-implied MRP
- See if you can use some mathematical property of given MDP/MRP
- Or create sampling traces and estimate with occurrence frequency
- Backup choice: Uniform Distribution of all non-terminal states
- Likewise, for backward induction, see if you can utilize some property of the given process to infer distribution of states for a fixed time step
- eg: In finance, continuous-time processes can sometimes be solved
- Or create sampling traces and estimate with occurrence frequency
- Backup choice: Uniform Distribution of all non-terminal states

Key Takeaways from this Chapter

- The FunctionApprox interface involves three key methods:
 - solve: Calculate the “best-fit” parameters that minimizes the cross-entropy loss function for the given fixed data set of (x, y) pairs
 - update: Parameters of FunctionApprox are updated based on each new (x, y) pairs data set from the available data stream
 - evaluate: Calculate the conditional expectation of response variable y , according to the model specified by FunctionApprox
- Tabular is a special case of linear function approximation with feature functions as indicator functions for each of the finite set of \mathcal{X}
- All the Tabular DP algorithms can be generalized to ADP algorithms
 - Tabular VF updates replaced by updates to FunctionApprox parameters
 - Sweep over all states in Tabular case replaced by state samples
 - Bellman Operators' Expectation estimated as average of calculations over transition samples (versus using explicit transition probabilities)