A Guided Tour of Chapter 6: 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:

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

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, floa
) -> 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 $\phi_j : \mathcal{X} \rightarrow \mathbb{R}, j = 1, 2, \ldots, m$

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

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

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

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

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

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

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

Linear Function Approximation with Gradient Descent

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

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

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

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

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

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

Direct Solution of Linear Function Approximation

- If feature functions not too large, we can directly solve for $\boldsymbol{w}^*$
- 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 $\boldsymbol{w}^*$

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

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

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

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

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

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 $\boldsymbol{I_l}$ and $\boldsymbol{O_l}$

$$\boldsymbol{I_0} = \phi(x) \in \mathbb{R}^m \text{ and } \boldsymbol{O_L} = \mathbb{E}_M[y|x] \text{ and } \boldsymbol{I_{l+1}} = \boldsymbol{O_l}$$

- Denote layer l parameters as $|\boldsymbol{O_l}| \times |\boldsymbol{I_l}|$ matrix $\boldsymbol{w_l}$
- Layer l neurons define a linear transformation from $\boldsymbol{I_l}$ to $\boldsymbol{S_l}$

$$\boldsymbol{S_l} = \boldsymbol{w_l} \cdot \boldsymbol{I_l} \text{ and } \boldsymbol{O_l} = g_l(\boldsymbol{S_l})$$

- 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 $\nabla_{\boldsymbol{w_l}} \mathcal{L}$
- Gradient Descent updates for weights $\boldsymbol{w_l}$ proportional to $\nabla_{\boldsymbol{w_l}} \mathcal{L}$

Back-prop as Recursive Gradient Calculation

- Loss gradient can be reduced to calculating $\boldsymbol{P_l} = \nabla_{\boldsymbol{S_l}} \mathcal{L}$

$$\nabla_{\boldsymbol{w_l}} \mathcal{L} = (\nabla_{\boldsymbol{S_l}} \mathcal{L})^T \cdot \nabla_{\boldsymbol{w_l}} \boldsymbol{S_l} = \boldsymbol{P_l}^T \cdot \nabla_{\boldsymbol{w_l}} \boldsymbol{S_l} = \boldsymbol{P_l} \cdot \boldsymbol{I_l}^T = \boldsymbol{P_l} \otimes \boldsymbol{I_l}$$

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

$$\nabla_{\boldsymbol{w_l}} \mathcal{L} = \boldsymbol{P_l} \cdot \boldsymbol{I_l}^T + \lambda_l \cdot \boldsymbol{w_l}$$

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

Theorem

$$\boldsymbol{P_l} = (\boldsymbol{w_{l+1}}^T \cdot \boldsymbol{P_{l+1}}) \circ g_l'(\boldsymbol{S_l}) \text{ \textit{(read the proof in the book)}}$$

To calculate $\boldsymbol{P_L} = \nabla_{\boldsymbol{S_L}} \mathcal{L}$, 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 $y \sim \mathcal{N}(\mu, \sigma^2)$:
  $S_L = \mu, \tau = \sigma, h(y, \tau) = \frac{e^{\frac{-y^2}{2\tau^2}}}{\sqrt{2\pi}\tau}, A(S_L) = \frac{S_L^2}{2}, d(\tau) = \tau^2.$ $g_L(S_L) = S_L$
- Bernoulli distribution for binary-valued y, parameterized by p:
  $S_L = \log(\frac{p}{1-p}), \tau = h(y, \tau) = d(\tau) = 1, A(S_L) = \log(1 + e^{S_L}).$
  $g_L(S_L) = \frac{1}{1+e^{-S_L}}$
- Poisson distribution for y parameterized by λ:
  $S_L = \log \lambda, \tau = d(\tau) = 1, h(y, \tau) = \frac{1}{y!}, A(S_L) = e^{S_L}.$ $g_L(S_L) = e^{S_L}$

Tabular as a form of *FunctionApprox*

- "Tabular" is simple setting with finite $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$
- With (x, y) data pairs having all it's 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 $\boldsymbol{B}^\pi$ 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

```python
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)