IEOR 265 - Lecture 7
Parametric Approximation Architectures

1 Linear Architectures and Features

As always, we start by stating the DP problem with disturbances:

$$J^*(x_0) = \min_{\pi \in \Pi} \mathbb{E}_w \big[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \big]$$
$$x_{k+1} = f_k(x_k, u_k(x_k), w_k), \forall k \in \{0, 1, ..., N-1\} \tag{1}$$

where the expectation on the right-hand side is taken w.r.t. the joint probability distribution of $(w_0, ..., w_{N-1})$.

Next, we state the DP recursion where the cost-to-go is replaced by some approximate value function $\tilde{J}_{k+1}(x_{k+1})$:

$$J_N(x_n) = g_N(x_N)$$
$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \big\{ \mathbb{E}_{w_k} \big[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \big] \big\}, \forall k \in \{0, ..., N-1\} \tag{2}$$

We will proceed to cover parametric approximate value functions, where $\tilde{J}_{k+1}(x_{k+1})$ is obtained by some parametric form based on **features**. The term "features" may be used to represent distinct elements in approximate DP: here we will consider features to be a mapping from the states x_k to a vector $\phi_k(x_k)$, which is taken to represent the important information regarding state x_k:

$$\phi_k(x_k) = [\phi_{k,1}(x_k), ..., \phi_{k,m_k}(x_k)]^\top \tag{3}$$

So each element of the vector $\phi_k(x_k)$ represent some distinct characteristic of the state vector x_k.

Before the rise of Reinforcement Learning techniques, feature-based approximation played a central role (and it is still relevant) in trying to reduce the dimension of the state space, by extracting the relevant information from the state. This extraction was done for the most part, via expert knowledge and practice: For example, in the game of chess, given the board position x_k, an expert can extract features such as: pawn structure, material balance, king exposure, etc.

After this feature vector $\phi_k(x_k)$ is obtained we can use a linear architecture to provide the approximation for the value function at stage k:

$$\tilde{J}_k(x_k) = \hat{J}_k(\phi(x_k), r_k) = \sum_{i=1}^{m_k} r_{k,i}\phi_{k,i}(x_k) = r_k^\top \phi_k(x_k) \tag{4}$$

The intuition behind this approximation is that if the features we use are "good-enough" to capture the complexity of the relationship between different states, then we can use a "simple" linear architecture to approximate the optimal cost-to-go. This is called *linear feature-based architecture*, and the vector $r_k = [r_{k,1}, ..., r_{k,m_k}]^\top$ is called the *weight vector.*

1.1 Scalar-impulses example

As an example, consider a case where the state x_k is a scalar. Suppose we are able to partition the real line into subsets $S_1, ..., S_{m_k}$, and we define the feature as follows:

$$\phi_l(x) = \begin{cases} 1 \text{ if } x \in S_l \\ 0 \text{ if } x \notin S_l \end{cases} , \forall l \in \{1, ..., m_k\} \tag{5}$$

Then using the linear feature-based architecture with a weight-vector r_k, can be illustrated by the figure:

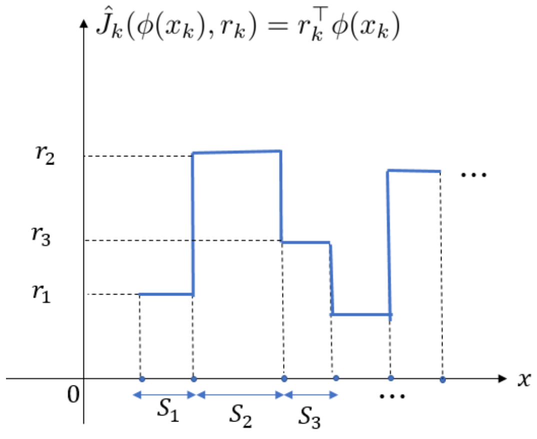

Figure 1: linear feature-based architecture with constant piece-wise features. The different weights r_i correspond to impulses on their respective subset S_i.

1.2 Training Linear Architectures

Given the feature vector $\phi_k(x_k)$ and the cost-to-go parametric approximation:

$$\tilde{J}_k(x_k) = r_k^\top \phi_k(x_k) \tag{6}$$

A key consideration regards the problem of finding the vector r_k, such that the approximate value function $\tilde{J}_k(x_k)$ best matches the optimal cost-to-go $J_k^*(x_k)$ function. For the moment, suppose we are able to generate sample pairs $\{(x_k^s, J_k^s)\}_{s=1}^S$ of states and optimal cost-to-go values. Then we can find the estimate $\hat{r}_k$, by solving the following *Least-Squares Regression*:

$$\hat{r}_k = \arg\min_r \Big\{ \sum_{s=1}^{S} (r_k^\top \phi(x_k^s) - J_k^s)^2 \Big\} \tag{7}$$

This optimization is simple enough, that the optimal solution can be obtained in closed-form by taking the gradient and setting it to zero:

$$2\sum_{s=1}^{S}\phi(x_k^s)(r_k^\top\phi(x_k^s) - J_k^s) = 0 \tag{8}$$

or, equivalently:

$$\sum_{s=1}^{S}\phi(x_k^s)\phi(x_k^s)^\top r_k^\top = \sum_{s=1}^{S}\phi(x_k^s)J_k^s \tag{9}$$

and we can take the matrix inversion and obtain the optimal solution:

$$\hat{r}_k = \Big(\sum_{s=1}^{S}\phi(x_k^s)\phi(x_k^s)^\top\Big)^{-1}\Big(\sum_{s=1}^{S}\phi(x_k^s)J_k^s\Big) \tag{10}$$

Technically, we need to ensure that we are able to perform the inversion, which can be done if the matrix is full-rank or by adding a suitable regularization term (but we will not delve into this here).

2 Neural Network Architectures

2.1 Neural Networks and Training

For many applications, it can be a difficult task to obtain good features such that the linear architecture yields good results (that is that $\tilde{J}_k(x_k, \hat{r}_k)$ is close to $J_k^*(x_k)$). In these settings we may rely on another approximation architecture (a Neural Network) to provide us a (non-linear) feature vector $\phi(x_k, v_k)$, where v_k are this architecture parameter. Then, the approximate value function is given by:

$$\tilde{J}_k(x_k, v_k, r_k) = r_k^\top\phi(x_k, v_k) \tag{11}$$

where the architecture parameters are now both r_k and v_k. Still, we can write the (non-linear) regression problem:

$$(\hat{r}_k, \hat{v}_k) = \arg\min_{r_k, v_k}\Big\{\sum_{s=1}^{S}(r_k^\top\phi(x_k^s, v_k) - J_k^s)^2\Big\} \tag{12}$$

However, due to the potential nonlinearities of the feature vector $\phi(x_k^s, v_k)$, there is no closed-form solution in general for the optimization problem. In addition, the problem can be non-convex and one would only be able to guarantee that the solution is a local minimum. The closed-form solution of linear regression is replaced by iterative methods, such as Gradient Methods.

There is a vast array of different iterative methods available in literature, and most software packages (such as TensorFlow [1]), implement one or another method. In this lecture, we will present the standard **Stochastic Gradient Method**. We start by defining the following function:

$$f(x_k^s, J_k^s, \theta_k) = \Big(r_k^\top\phi(x_k^s, v_k) - J_k^s\Big)^2$$

where $\theta_k = (r_k, v_k)$. Then we can re-write the regression problem Eq(12) as the equivalent problem:

$$\hat{\theta}_k = \arg\min_{r_k, v_k} \Big\{ \frac{1}{S} \sum_{s=1}^{S} f(x_k^s, J_k^s, \theta_k) \Big\} = \arg\min_{r,v} \big\{ F(\theta_k) \big\} \tag{13}$$

where:

$$F(\theta_k) = \frac{1}{S} \sum_{s=1}^{S} f(x_k^s, J_k^s, \theta_k) \tag{14}$$

and we note that by dividing the objective function by the total number of samples S, does not change the set of optimal solutions to the problem. The Stochastic Gradient Method samples (at random in an i.i.d. fashion) and index $j \in \{1, ..., S\}$ and then computes the following gradient:

$$\nabla_\theta f(x_k^j, J_k^j, \theta_k) \approx \nabla_\theta F(\theta_k) \tag{15}$$

where we approximate the gradient of the entire objective function $\nabla_\theta F(\theta_k)$, by the gradient of a single component $\nabla_\theta f(x_k^j, J_k^j, \theta_k)$, associated with the j'th sample. Then, given some scalar $\alpha^{(t)}$ (called the *stepsize*), an iteration t of the method is given by:

$$\theta_k^{(t+1)} = \theta_k^{(t)} - \alpha^{(t)} \nabla_\theta f(x_k^j, J_k^j, \theta_k^{(t)}) \tag{16}$$

This process is repeated after T iterations or until some convergence criterion is met. We summarize the Stochastic Gradient Method below:

Algorithm 1 Stochastic Gradient Method

```
Input: Initial parameter vector $\theta_k^0 = (r_k^{(0)}, v_k^{(0)})$.
1: Collect S samples of state/cost-to-go pairs $\{(x_k^s, J_k^s)\}_{s=1}^S$.
2: for $t = 0, ..., T$ do (stochastic gradient iterations)
3:     draw uniformly at random one sample $j \in \{1, ..., S\}$.
4:     Compute :
           $\theta^{(t+1)} = \theta^{(t)} - \alpha^{(t)} \nabla_\theta f(x_k^j, J_k^j, \theta_k^{(t)})$
5: end for
Output: the final parameter vector $\theta_k^{(T)} = (r_k^{(T)}, v_k^{(T)})$
```

As we mentioned before, this implementation can be enhanced in several ways: by selecting the appropriate stepsize $\alpha^{(t)}$ at each iteration; using a batch of samples instead of a single sample in steps 3-4; changing Eq(16) using scaling and deflecting matrices, etc. These extensions and improvements are not part of the scope of the course and are left as extra reading.

2.2 Example: Single-Layer Perceptron

On this subsection, we illustrate the usage of a single perceptron unit as the approximation architecture. A perceptron is illustrated in the following figure:

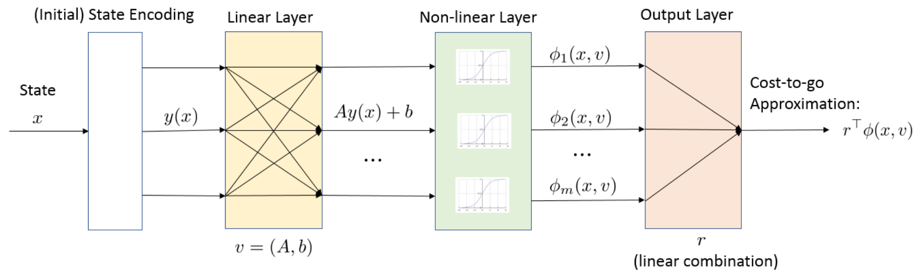

Figure 2: A perceptron unit consisting of the initial numerical encoding transforming the state x into the vector $y(x)$, which is passed to a linear layer as $Ay(x)+b$, followed by a non-linear transformation, which produces the different features $\phi_l(x,v)$, $l \in \{1,...,m\}$. Those features are linearly combined in the output layers producing the cost-to-go approximation $\tilde{J} = r^\top \phi(x,v)$.

As is commonly done in practice, the state x_k is first encoded into a vector of numerical values $y(x)$ (so for example if x_k denotes the board position in chess, then $y(x)$ is some numerical encoding so it can be processed by a computer). Without loss of generality, assume:

$$y(x_k) = [y_1(x_k), ..., y_n(x_k)]^\top \tag{17}$$

Then this numerical vector is passed through a linear transformation, obtaining the vector z:

$$z = Ay(x) + b \tag{18}$$

where A is a $m \times n$-matrix and b is a m-vector. This transformation step is called the linear layer of the perceptron. Then this intermediate vector z is fed, component-wise, into a nonlinear function $\sigma : \mathbb{R} \to \mathbb{R}$:

$$\phi_l(x_k, v_k) = \sigma(z^l) = \sigma((Ay(x)+b)_l), \forall l \in \{1,...,m\} \tag{19}$$

where we use the notation $(Ay(x)+b)_l$ to denote the l'th component of the linear layer output and we let $v_k = (A,b)$. Typical non-linear functions used are given below:

$$\sigma(\epsilon) = \max\{0,\epsilon\} \qquad \text{rectified-linear unit (ReLU)} \tag{20}$$

$$\sigma(\epsilon) = \frac{1}{1+e^{-e^{\epsilon}}} \qquad \text{sigmoid} \tag{21}$$

$$\sigma(\epsilon) = \tanh(\epsilon) = \frac{e^{\epsilon}-e^{\epsilon}}{e^{\epsilon}+e^{\epsilon}} \qquad \text{hyperbolic tangent} \tag{22}$$

$$\tag{23}$$

Lastly each non-linear feature $\phi_l(x_k, v_k)$ is sent to the final output layer which performs a linear combination with the weight vector r_k, obtaining the approximation of the cost-to-go associated with state x_k:

$$\tilde{J}_k(x_k, v_k, r_k) = \sum_{l=1}^{m_k} r_{k,l}\phi_l(x_k, v_k) = r_k^\top \phi(x_k, v_k) \tag{24}$$

2.2.1 Example: 2x2 perceptron

As an illustration let's perform a single Stochastic Gradient iteration for the case of a perceptron unit where $A \in \mathbb{R}^{2\times 2}$ and we will assume no intercept, so $b = 0$ (note that we can always add a vector of "ones" as an extra component of $y(x)$ and treat the problem as having no intercept). We will drop the subscript denoting the stage k, for ease of notation. Suppose we use the sigmoid function as our non-linear function of choice:

$$\sigma(\epsilon) = \frac{1}{1 + e^{-\epsilon}} \tag{25}$$

Then at iteration t, given a random sample (x^s, J^s) we need to compute the gradient of:

$$f(x^s, J^s, \theta^{(t)}) = \left(\sum_{l=1}^{m} r_l^{(t)} \sigma((A^{(t)} y(x^s))_l) - J^s \right)^2 \tag{26}$$

where $m = 2$ and $\theta^{(t)} = (r^{(t)}, A^{(t)})$. Note that we need to differentiate Eq(26) w.r.t. every element of matrix A and w.r.t. every element of the vector r. This is obtained by successive applications of the chain rule:

$$\frac{\partial f(x^s, J^s, \theta^{(t)})}{\partial a_{i,j}} = 2E^{(t)}(x^s, J^s) r_i^{(t)} \sigma(z_i^{(t)})(1 - \sigma(z_i^{(t)})) y(x^s)_j,\ \forall i, j \in \{1, 2\} \tag{27}$$

$$\frac{\partial f(x^s, J^s, \theta^{(t)})}{\partial r_i} = 2E^{(t)}(x^s, J^s)\sigma(z_i^{(t)}),\ \forall i \in \{1, 2\} \tag{28}$$

where we let:

$$z_i^{(t)} = (A^{(t)} y(x^s))_i \ ,\ i \in \{1, 2\} \tag{29}$$

and we define the error term associated with sample s:

$$E^{(t)}(x^s, J^s) = r_l^{(t)} \sigma(z_l^{(t)}) + r_2^{(t)} \sigma(z_2^{(t)}) - J^s \tag{30}$$

and we used the fact that:

$$\frac{\partial \sigma(\epsilon)}{\partial \epsilon} = \sigma(\epsilon)(1 - \sigma(\epsilon)) \tag{31}$$

Then the Stochastic gradient step becomes:

$$a_{i,j}^{(t+1)} = a_{i,j}^{(t)} - \alpha^{(t)} \frac{\partial f(x^s, J^s, \theta^{(t)})}{\partial a_{i,j}},\ \forall i, j \in \{1, 2\} \tag{32}$$

$$r_i^{(t+1)} = r_i^{(t)} - \alpha^{(t)} \frac{\partial f(x^s, J^s, \theta^{(t)})}{\partial r_i},\ \forall i \in \{1, 2\} \tag{33}$$

$$\tag{34}$$

2.3 Deep Neural Networks (DNN's)

Now, let's turn our attention to Deep Neural Networks, where we concatenate the single perceptron, with other perceptrons many many times, obtaining the structure illustrated by the following figure:

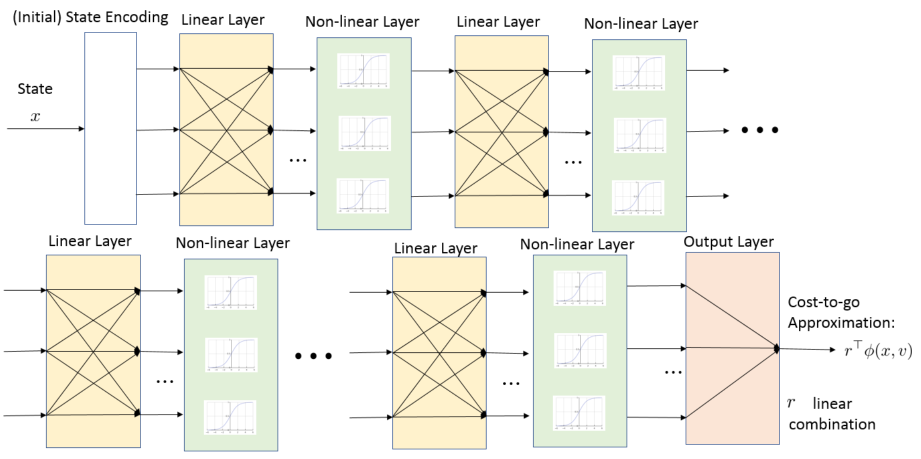

Figure 3: A (feed-forward) Deep Neural Network, composed of a sequence of linear layers and layers with non-linear transformations, where non-linear features are fed into linear transformations sequentially, until the last linear combination in the output layer.

The usage of DNN is based on empirical work and practical experience. It is not obvious *a priori* how many layers and which types of non-linearities yield the best results for a given application: trial and error and constant experiment are necessary in order to define a particular architecture.

In order to give some intuition we can (informally) argue that using many layers provide us "flexibility" in constructing an "hierarchy" of features: after each non-linear layer, we output a feature vector that gets fed as input to the next linear layers, and so on. On more practical grounds, it allow us to use sparse matrices in the linear layers, which provide significant computational speed-ups during training.

Nonetheless, the regression problem is given as before:

$$(\hat{r}_k, \hat{v}_k) = \arg\min_{r_k, v_k} \Big\{ \sum_{s=1}^{S} (r_k^\top \phi(x_k^s, v_k) - J_k^s)^2 \Big\} \tag{35}$$

but now the parameter vector v_k contain information of many linear layers. The regression problem is solved essentially in the same as the single perceptron example: we compute the gradient samples and perform gradient steps, iteratively. Due to the feed-forward nature of the DNN's considered, the gradient computation can be done very efficiently by smart application of the chain-rule. This is the **Backpropagation Algorithm** (also called "backprop") which is done via Automatic Differentiation[2] (AD) in many software packages, such as PyTorch[3].

Algorithm 2 Backpropagation Algorithm

```
Input: Sample pair $(x_k^s, J_k^s)$.
 1: Perform a forward pass in the DNN to calculate sequentially the outputs of
    the linear layers. Let $z_1 = L_1 x_k^s$
 2: for $i = 2, ..., m+1$ do (forward pass)
 3:     Compute: $z_i = L_i \Sigma_{i-1} z_{i-1}$
 4: end for
 5: Compute the error term:
        $$E = M(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1}, x_k^s) - J_k^s$$
 6: Perform a backward pass in the DNN to calculate sequentially the required
    derivatives. Let $y_{m+1} = E^\top$ .
 7: Compute the Derivative of the output layer:
        $$\frac{\partial S(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1})}{\partial L_{m+1}(i,j)} = y_{m+1} I_{ij} \Sigma_m z_m \quad (36)$$
 8: for $t = m, ..., 1$ do (backward pass)
 9:     Evaluate $\bar{\Sigma}_t$ at $z_t$.
10:     Compute: $y_t = y_{t+1} L_{t+1} \bar{\Sigma}_t$
11:     Compute the Derivatives:
        $$\frac{\partial S(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1})}{\partial L_t(i,j)} = y_t I_{ij} \Sigma_{t-1} z_{t-1} \quad (37)$$
12: end for
Output: the gradient of Least Squares objective w.r.t. to the DNN parameters
    on the sample pair $(x_k^s, J_k^s)$.
```

We will limit ourselves in providing an algebraic version of the Backpropagation algorithm, as the AD methods are beyond the scope of the course. Let's consider a DNN with m non-linear layers and $m+1$ linear layers (recall that there is always one extra linear layer as the output layer). For each linear layer let $L_i, i \in \{1, ..., m+1\}$ be the mapping associated with it; Similarly, for each non-linear layer let $\Sigma_i \in \{1, ..., m\}$ be the associated mapping. For simplicity let the initial numerical encoding be the identity, so $y(x) = x$. Then we can define the output of the DNN as the following function:

$$M(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1}, x_k^s) = L_1 \Sigma_1, \cdots L_m \Sigma_m L_{m+1} x_k^s \tag{38}$$

And given some cost-to-go sample J_k^s, we can define the squared error function as:

$$S(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1}) = \Big(M(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1}, x_k^s) - J_k^s\Big)^2 \tag{39}$$

Now, if we apply the chain-rule to compute the derivative of the (i,j)'th of the p'th linear layer, we obtain:

$$\frac{\partial S(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1})}{\partial L_p(i,j)} = E^\top L_{m+1}\bar{\Sigma}_m L_m \cdots L_{p+1} I_{ij} \Sigma_{p-1} \cdots \Sigma_1 L_1 x_k^s \tag{40}$$

where:

$$E = M(L_1, \Sigma_1, ..., L_m, \Sigma_m, L_{m+1}, x_k^s) - J_k^s \tag{41}$$

and $\bar{\Sigma}_i, i \in \{1, ..., m\}$ are diagonal matrices with the diagonal terms equal to the derivatives of the non-linear functions σ_i evaluated at their respective arguments. Lastly, I_{ij} is the matrix obtained from L_p by setting all of its components to 0 except for the ij-component which is set to 1. With Eq(38) the Backpropagation Algorithm is be defined in Algorithm 2.

3 Fitted Value Iteration: finite-horizon case

So far, the presented methods are essentially *Supervised Learning* algorithms: The goal is to find a parametric mapping from states x_k to cost-to-go values $J_k(x_k)$. The mappings themselves can range from simple linear models to complexed deep neural networks with several non-linear layers. On this section we will use those parametric functions inside a DP algorithm in order to generate a sub-optimal policy for the DP problem.

We will study the **Fitted Value Iteration Algorithm** applied to finite horizon DP problems. The goal is to recursively generate the cost-to-go values starting from $J_N(x_N)$ and use them to train a parametric architecture to approximate the cost-to-go $J_{N-1}(x_{N-1})$, and then use that architecture to generate the cost-to-go values for the $(N-2)$'th step, which is then used to train another architecture, and so forth: proceeding in the typical backward fashion until the initial state. In this approach we, then, have N different parametric architectures, defined by their respective parameters $\theta_k, k = \{0, ..., N-1\}$.

Even though this algorithm is presented for the finite-horizon case (where N is fixed and finite), it will lay the foundation for the future algorithms, which were developed for the infinite-horizon case. The Algorithm starts at the last stage, where the exact cost-to-go function is known:

$$J_N(x_N) = g_N(x_N) \tag{42}$$

Then after sampling states $\{x_{N-1}^{(s)}\}_{s=1}^S$, we perform the following Regression step:

$$\theta_{N-1}^* \in \arg\min_\theta \left\{ \sum_{s=1}^S \Bigg(\tilde{J}_{N-1}(x_{N-1}^s, \theta_{N-1}) - \right. \tag{43}$$

$$\left. \min_{u_{N-1} \in U_{N-1}(x_{N-1}^s)} \left\{ \mathbb{E}_{w_{N-1}} \left[g_{N-1}(x_{N-1}^s, u_{N-1}, w_{N-1}) + g_N(f_{N-1}(x_{N-1}^s, u_{N-1}, w_{N-1})) \right] \right\} \Bigg)^2 \right\} \tag{44}$$

where the expectation w.r.t. w_{N-1} can be either computed exactly, or by using sample average approximation via sampling. Note that we can define the cost-to-go "labels" J^s_{N-1} as:

$$J^s_{N-1} = \min_{u_{N-1} \in U_{N-1}(x^s_{N-1})} \left\{ \mathbb{E}_{w_{N-1}} \left[g_{N-1}(x^s_{N-1}, u_{N-1}, w_{N-1}) + g_N(f_{N-1}(x^s_{N-1}, u_{N-1}, w_{N-1})) \right] \right\} \tag{45}$$

and we recover, the usual regression-type minimization:

$$\theta^*_{N-1} \in \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left(\tilde{J}_{N-1}(x^s_{N-1}, \theta_{N-1}) - J^s_{N-1} \right)^2 \right\}$$

Now we proceed backwards in-time such that, at iteration k, by obtaining new samples $\{x_k^{(s)}\}_{s=1}^S$ we compute:

$$\theta^*_k \in \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left(\tilde{J}_k(x^s_k, \theta_k) - J^s_k \right)^2 \right\}$$

where:

$$J^s_k = \min_{u_k \in U_k(x^s_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x^s_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x^s_k, u_k, w_k), \theta^*_{k+1}) \right] \right\}$$

thus going all the way until the initial stage 0. Lastly, the suboptimal policy is given directly by the 1-step lookahead minimization:

$$\tilde{\mu}_k(x_k) = \arg \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k), \theta^*_{k+1}) \right] \right\}$$

We highlight that this approach is highly dependent on how the state samples, for every stage, are generated. Intuitively, the approximation will be better if we are able to sample a "rich" set of states. More formally, the frequency in which the generated sampled states appear should be roughly proportional to the probabilities of their occurrence under an optimal policy. We will discuss more about how to generate "good" samples on future lectures.

Lastly, the Fitted Value Iteration as presented, quickly becomes impractical as the horizon N becomes too large and if the parametric architectures are complex. On the other hand, most practical problems (in robotics, AI gaming, self-driving, etc) are problems with very long horizons. This is the key motivation to treat those DP problems as infinite-horizon problems, which will open new and more efficient ways to incorporate approximation architectures into the DP framework.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[2] A. Griewank *et al.*, "On automatic differentiation," *Mathematical Programming: recent developments and applications*, vol. 6, no. 6, pp. 83–107, 1989.

[3] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.