# Approximation in Value Space

- Let's begin by writing the DP algorithm:

$$J_N(x_n) = g_N(x_N)$$

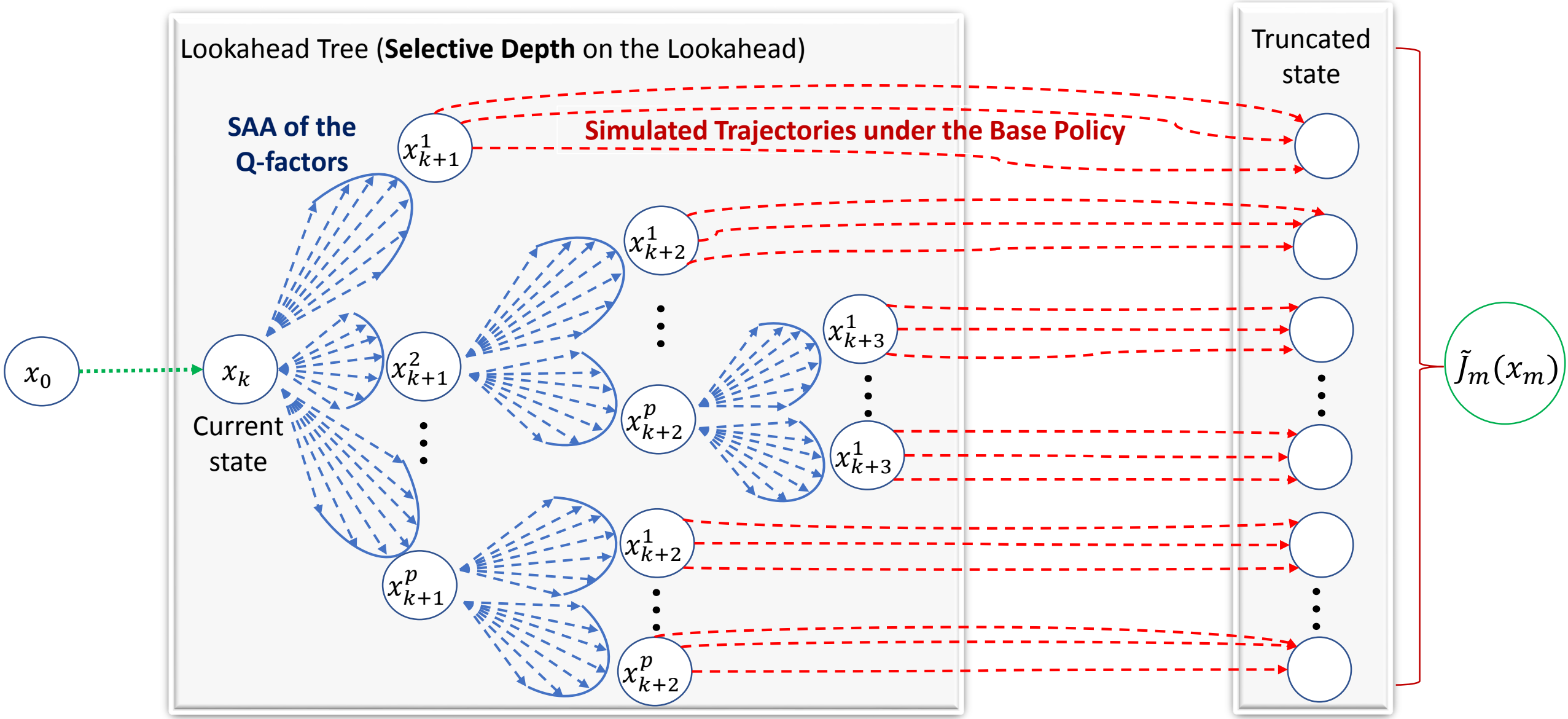
$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[ g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, ..., N-1\}$$

- Where we can compute a sub-optimal admissible policy by using an approximate cost-to-go function $\tilde{J}_{k+1}(x_{k+1})$ :

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, ..., N-1\}$$

- $\tilde{J}_{k+1}(x_{k+1})$ is also called an **approximate value function.**

# Monte-Carlo Tree Search (MCTS)



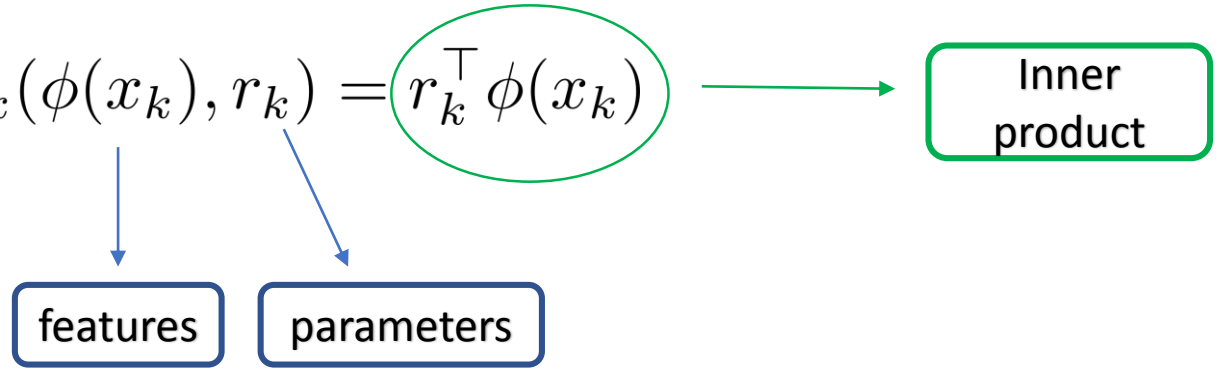
# Approximation Architectures

- We will study how to generate approximation architectures to approximate $\tilde{J}_k(x_k)$ for any state $k$.

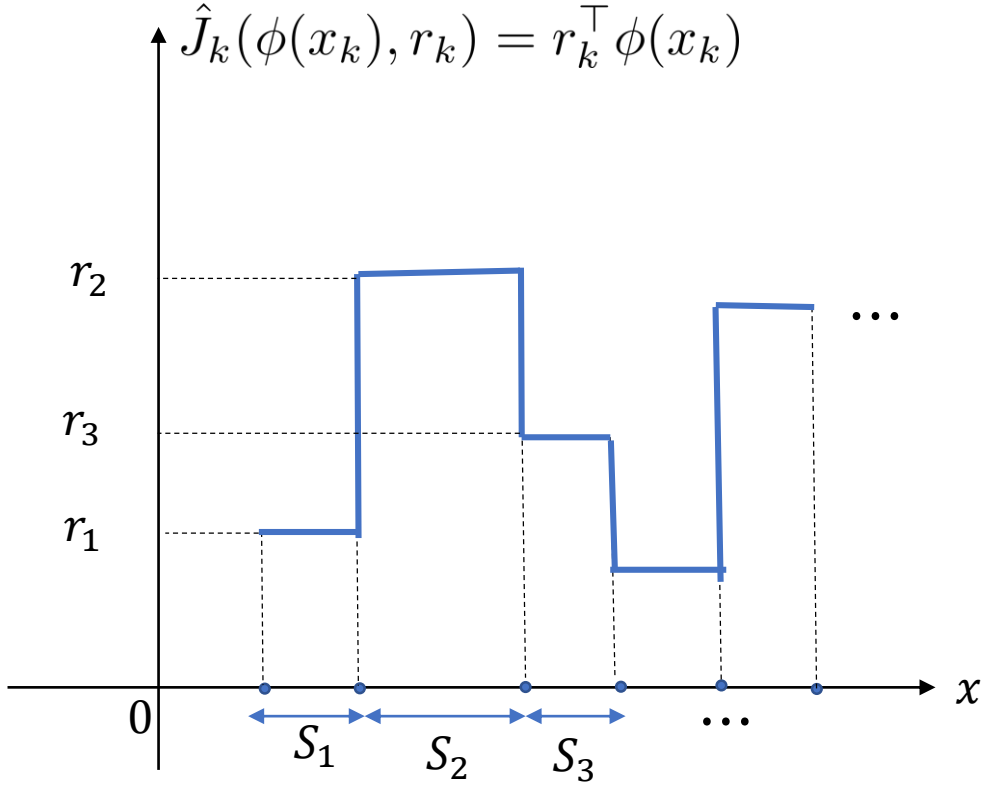
- The idea is simple: We would like to perform two tasks:
  - Extract features that "represent" the system states $x_k$
  - Use these features to provide approximate values for $\tilde{J}_m(x_m)$

- The simplest form of architecture is a **linear architecture:**

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



# Example: scalar impulses




Features are like "impulses":

$$\phi_l(x) = \begin{cases} 1 \text{ if } x \in S_l \\ 0 \text{ if } x \notin S_l \end{cases}$$

- The features are also called the *basis functions*, since their linear combination can span the values of the approximate functions $\tilde{J}_k(x_k, r_k)$

# Example: AlphaGo

- The AlphaGo AI software (Silver et al, 2016) uses handcrafted features, based on experience.

- The state $x_k$ is the board position at the $k'th$ turn, which is extremely complex to represent. Therefore, the following features $\phi(x_k)$ are used to represent the state:

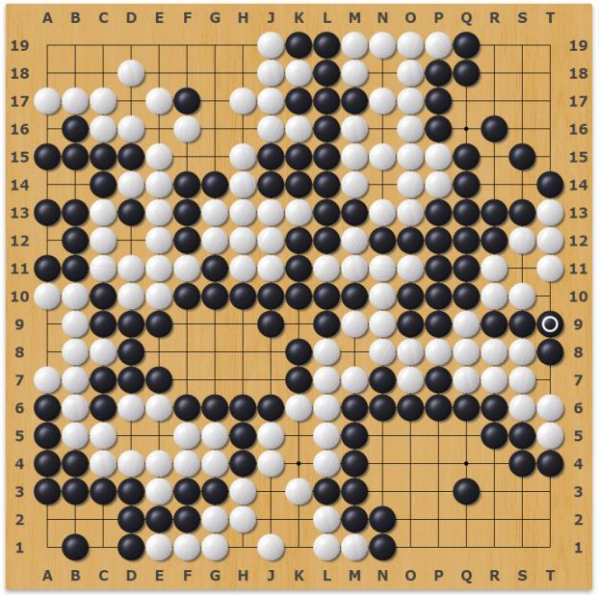
**Extended Data Table 2 | Input features for neural networks**

| Feature | # of planes | Description |
|---|---|---|
| Stone colour | 3 | Player stone / opponent stone / empty |
| Ones | 1 | A constant plane filled with 1 |
| Turns since | 8 | How many turns since a move was played |
| Liberties | 8 | Number of liberties (empty adjacent points) |
| Capture size | 8 | How many opponent stones would be captured |
| Self-atari size | 8 | How many of own stones would be captured |
| Liberties after move | 8 | Number of liberties after this move is played |
| Ladder capture | 1 | Whether a move at this point is a successful ladder capture |
| Ladder escape | 1 | Whether a move at this point is a successful ladder escape |
| Sensibleness | 1 | Whether a move is legal and does not fill its own eyes |
| Zeros | 1 | A constant plane filled with 0 |
| Player color | 1 | Whether current player is black |

Feature planes used by the policy network (all but last feature) and value network (all features).

Features are like binary ("one-hot") encodings of the board picture (a 19x19 image)

# Example: AlphaGo



19x19 image of the board

**Feature Encoding**

Feature: “Position Color”:

1. Black
2. White
3. Empty

19x19 binary matrices
(“feature planes”)

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**Black**

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**White**

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

**Empty**

# Least-Squares Regression

- Suppose we have at hand our features:

$$\phi_l(x_k),\ \forall l \in \{1, ..., m\}$$

- Now we need to *train* our linear model, that is we need to find the best configuration of the parameter vector $r = (r_1, \ldots, r_m)$ that best approximates the cost-to-go $J_k(x_k)$ .

- Suppose we collect, via simulation, pairs of states and "future costs":

$$(x^s, J^s),\ \forall s \in \{1, ..., S\}$$

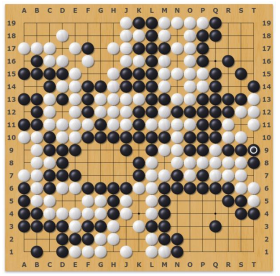
- Then we can solve the following least-squares problem:

$$\min_r \left\{ \sum_{s=1}^{S} (r^\top \phi(x^s) - J^s)^2 \right\}$$

# Example: AlphaGo

- Like we saw, in the timid-play/bold-play example, in AlphaGo, the "cost-to-go" is defined as the probability of winning the Go game given that the board position is $x$.

- For example we can collect pairs of board positions and the eventual outcome of the game associated with that position:



$x^1$

Feature encoding: → $\phi(x^1)$ Game outcome: → "cost-to-go" $J^1 = 1$ (a win)

$x^2$

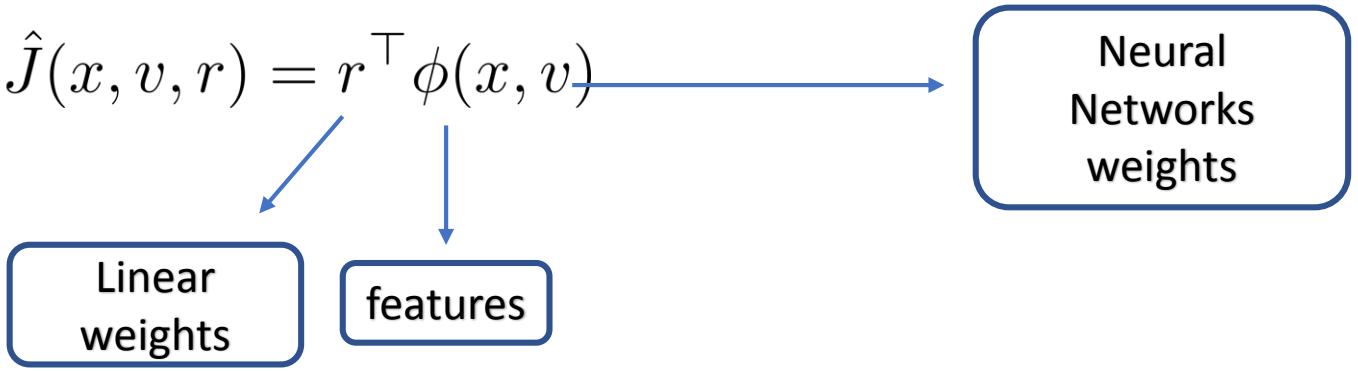
Feature encoding: → $\phi(x^2)$ Game outcome: → "cost-to-go" $J^2 = 0$ (a loss)

# Example: AlphaGo

- Now, suppose that we would like to capture more "complex" behaviors:
  - We want the features $\phi$ to capture relationships and important aspects of the states (for example in Go: strong positions, strong attack openings, etc.

- Then we can augmented the approximation architecture as follows (we dropped the stage index $k$ for simplicity):

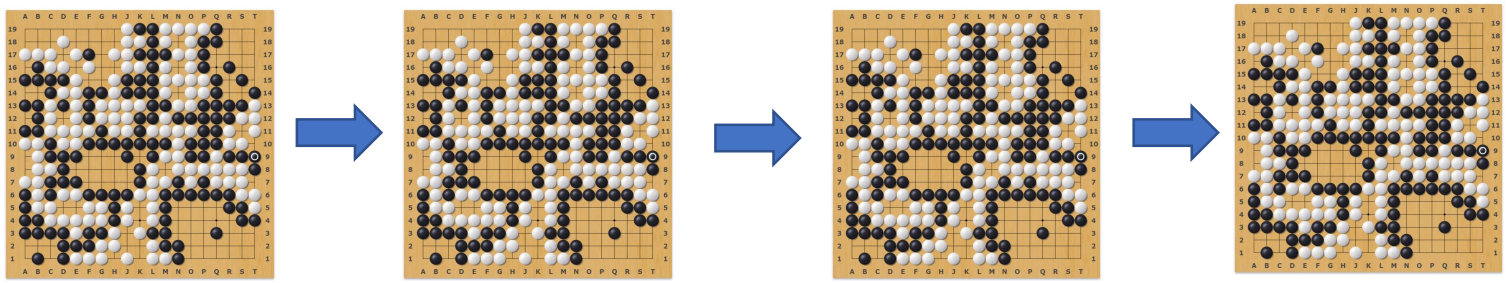
$$\tilde{J}(x) = \hat{J}(x, v, r) = r^\top \phi(x, v)$$




- The training problem is still: $\min_{r,v} \Big\{ \sum_{s=1}^{S} (r^\top \phi(x^s, v) - J^s)^2 \Big\}$

# Example: AlphaGo

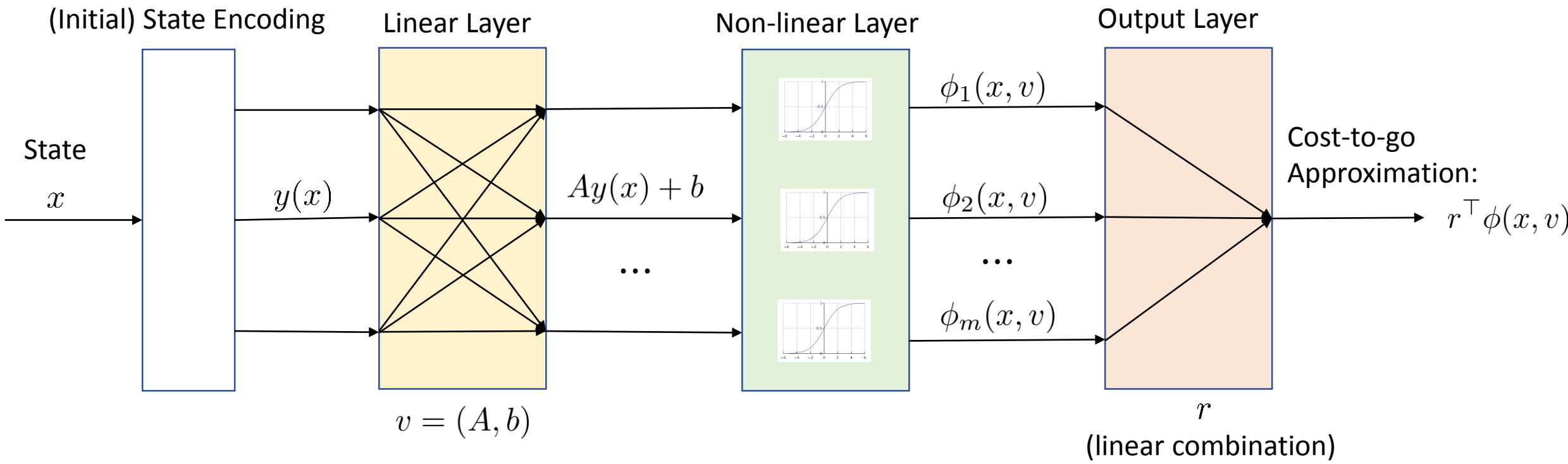
- For example we would like to “string” together sequences of good board position to obtain features relative to a combination of moves (tactics):




- So the feature evaluated at $x_k$ captures the relation between it and the past 3 board positions:
  - “heads-up” for convolutions!

# Single-Layer Perceptron

- The simplest Neural Network is the Single-Layer Perceptron:



# Training Neural Networks

- For a non-linear function $\sigma$ we can write the training problem for the 1-layer Neural Network:

$$\min_{A,b,r} \left\{ \sum_{s=1}^{S} \left( \sum_{l=1}^{m} r_l \sigma((Ay(x^s) + b)_l) - J^s \right)^2 \right\}$$

- Where we can add additional *regularization* as needed.

- However, the optimization now becomes more challenging, since the least-squares problems involve a non-linear function.

- Hence we need to resort to iterative methods to solve the above problem:
  - Stochastic Gradient Descent
  - Variable-Scaling Methods(ex: AdaGrad)

# Stochastic Gradient Descent

- Let $\theta = (A, b, r)$ and define:

$$f(x^s, J^s, \theta) = \left( \sum_{l=1}^{m} r_l \sigma((Ay(x^s) + b)_l) - J^s \right)^2$$

- Then the problem becomes:

$$\min_{\theta} \left\{ \sum_{s=1}^{S} f(x^s, J^s, \theta) \right\} = \min_{\theta} F(\theta)$$

- The basic idea is to treat the above problem as a “sample-average” approximation of the expectation over the distribution of the samples pairs:

$$(x^s, J^s), \forall s \in \{1, ..., S\}$$

# Stochastic Gradient Descent

- Then we can approximate the gradient of the objective function by the gradient of a single sample instead:

$$\nabla_\theta F(\theta) \approx \nabla_\theta f(x^s, J^s, \theta)$$

- And given some initial configuration $\theta^0$, we can define the iterative scheme:

$$\theta^{t+1} = \theta^t - \alpha_t \nabla_\theta J(x^s, J^s, \theta^s)$$

- Where we select which sample to use uniformly at random.

- The basic Stochastic Gradient Descent can be enhanced in a myriad of ways, but for our purposes it will suffice to stop here.

# Deep Neural Networks (DNN)

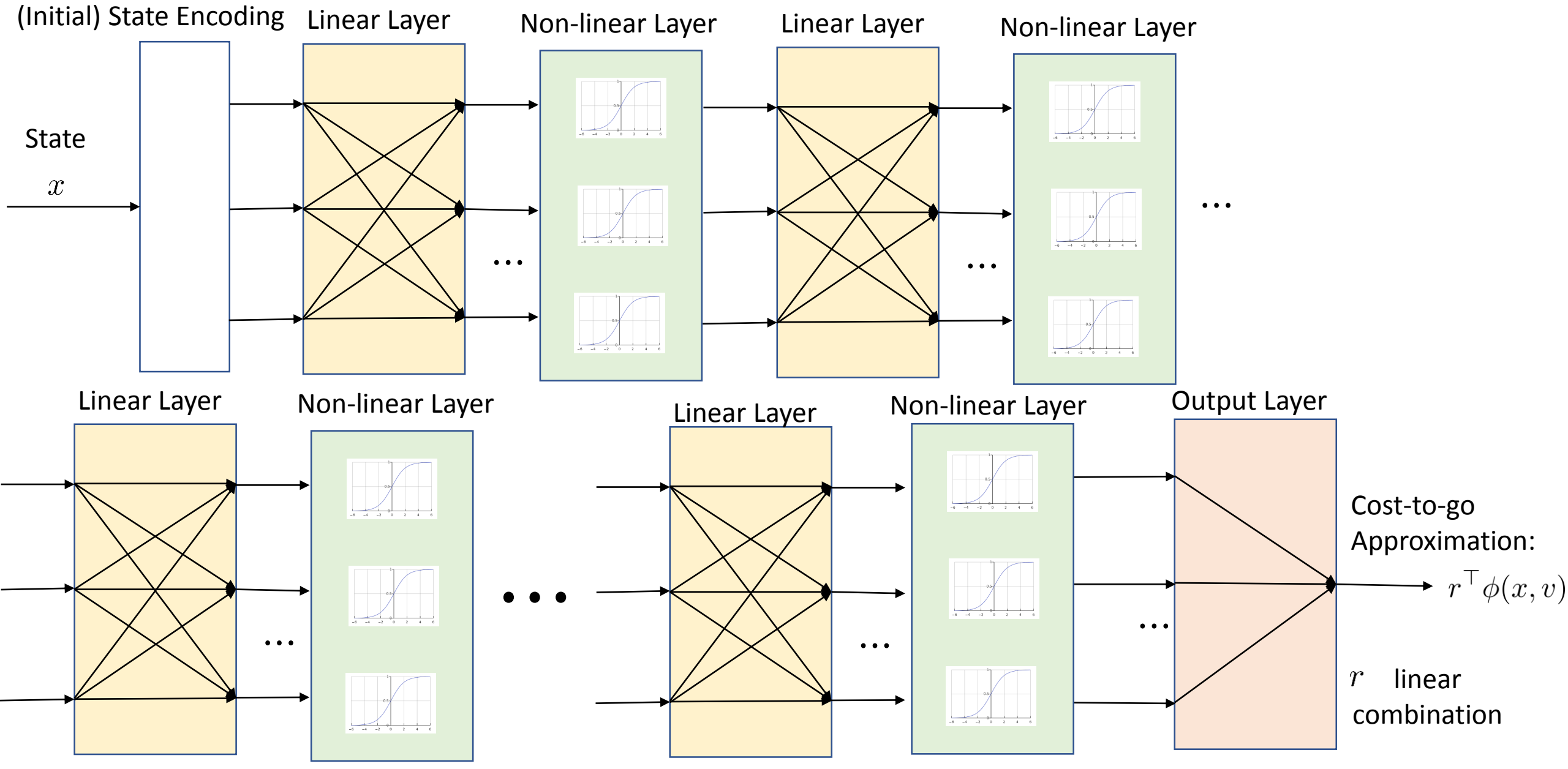
- In order to capture complex behavior, using a single extremely large layer is not good in practice:
  - We may not be able to capture temporal dependencies well
  - Difficulties to create a “hierarchy” of features
  - Gradients will contain many components and not possible to “break” it down in small pieces to facilitate computation

- So the alternative is to use many (but smaller) layers, leading to a Deep (long) sequence of layers.

- This gives us flexibility in designing each layer and tailor the architecture to the needs of the application.

- Example: AlphaGo uses networks with more than 17 layers.

# Deep Neural Networks (DNN)



# Training DNN's: Backpropagation

- The training of DNN's , in essence, is the same as the single-layer perceptron: We use Stochastic Gradients or it's variants/extensions.

- However, we can exploit the sequential layering to break the computation of the gradients:
  - Applying the chain-rule in a smart way.

- We can see that a DNN is a sequence of mappings, which maps the state $x$ to it's value function $\tilde{J}(x)$.
  - The sequence of mappings is done via an alternation of linear mappings and non-linear mappings

$$\begin{cases} x \to L_i x \text{ for every linear layer i} \\ x \to \Sigma_i x \text{ for every non-linear layer i} \end{cases}$$

# Training DNN's: Backpropagation

- We can then represent the entire approximation architecture with m layers as:

$$L_{m+1}\Sigma_m L_m \cdots \Sigma_1 L_1 y(x)$$

- Where $\theta$ is the collection of all parameters across all layers.

- Then the non-linear least-squares becomes:

$$\min_{\theta}\left\{\sum_{s=1}^{S}\left(L_{m+1}\Sigma_m L_m \cdots \Sigma_1 L_1 y(x^s) - J^s\right)^2\right\} = \min_{\theta}\left\{\sum_{s=1}^{S} E_s(L_1, \Sigma_1, \ldots, L_{m+1})\right\}$$

- Where $E_s(L_1, \ldots, L_{m+1})$ is the error function of sample s:

# Training DNN's: Backpropagation

- For each sample s, applying the chain rule we compute the partial derivative of $L_k(i,j)$, the ij'th component of the matrix $L_k$:

$$\frac{\partial E(L_1,...,L_{m+1})}{\partial L_k(i,j)} = -e^\top L_{m+1}\bar{\Sigma}_{m+1}L_m \cdots L_{k+1}\bar{\Sigma}_k I_{ij}\Sigma_{k-1}L_{k-1}\cdots\Sigma_1 L_1 y(x^s)$$

- Where:

$$e = J^s - F(L_1,...,L_{m+1})y(x^s) \qquad \bar{\Sigma}_n(z_n) = \begin{bmatrix} \frac{\partial\sigma_n}{\partial z}(z_n^1) & 0 & \cdots & 0 \\ 0 & \frac{\partial\sigma_n}{\partial z}(z_n^2) & \cdots 0 & \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \frac{\partial\sigma_n}{\partial z}(z_n^p) \end{bmatrix}$$

$I_{ij}$ = matrix with all components equal to 0, except the ij'th component, which is equal to 1

# Training DNN's: Backpropagation

- All derivatives can be obtained by an efficient forward-backward pass as follows:

- (1) Proceed forward by sequentially computing the output of each linear layer:

$$L_1 x,\ L_2 \Sigma_1 L_1 y(x^s),\ \ldots\ ,\ L_{m+1}\Sigma_m L_m \cdots \Sigma_1 L_1 y(x^s)$$

Evaluate the error vector $e = J^s - F(L_1, \ldots, L_{m+1}, x^s)$.
Evaluate the partial derivatives matrices $\bar{\Sigma}_i$ evaluated at the above points.

- (2) Proceed backward sequentially calculating the terms:

$$2e^\top L_{m+1}\bar{\Sigma}_m L_m \cdots L_{k+1}\bar{\Sigma}_k \text{ for } k \in \{m, \ldots, 1\}$$

And substituting in:

$$\frac{\partial E(L_1, \ldots, L_{m+1})}{\partial L_k(i,j)} = -e^\top L_{m+1}\bar{\Sigma}_{m+1} L_m \cdots L_{k+1}\bar{\Sigma}_k I_{ij}\Sigma_{k-1}L_{k-1}\cdots\Sigma_1 L_1 y(x^s)$$

# Using DNN's in DP Approximation

- Now let's return to our approximate DP framework:

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, ..., N-1\}$$

- Where $\tilde{J}_{k+1}(x_{k+1})$ is the approximate value function.

- We will use the DNN formulation to provide this approximation.

- Note that if used this way, this is essentially a 1-step lookahead approximation, where the lookahead problem is given directly by the DNN.

# Fitted Value Iteration: finite horizon case

- We will present here the finite horizon of the Fitted Value Iteration algorithm.
- In practice, it is not used much, as most practical problems are treated as Infinite-Horizon problems. However it will provide a basis of understanding as we move to the infinite horizon case (next lecture).
- In the finite horizon case, for each time period $k$, we will have one approximation architecture and it's associated parameter $\theta_k = (r_k, v_k)$.
- Or alternatively, you can think you have a single architecture, where the stage $k$ enters as an input.

# Fitted Value Iteration: finite horizon case

- In the finite horizon case, the algorithm proceeds (like always) backwards in time.

- First we start at stage $N-1$, and solve the following non-linear least-squares minimization:

$$\theta^*_{N-1} \in \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left( \tilde{J}_{N-1}(x^s_{N-1}, \theta_{N-1}) - \min_{u_{N-1} \in U_{N-1}(x^s_{N-1})} \left\{ \mathbb{E}_{w_{N-1}} \left[ g_{N-1}(x^s_{N-1}, u_{N-1}, w_{N-1}) + g_N(f_{N-1}(x^s_{N-1}, u_{N-1}, w_{N-1})) \right] \right\} \right)^2 \right\}$$

- Note that this long expression is ***still*** a (non-linear) regression problem. Note we can define:

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

# Fitted Value Iteration: finite horizon case

- And we have equivalently:

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

- Now, we proceed backwards in time:

$$\theta^*_k \in \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left( \tilde{J}_k(x^s_k, \theta_k) - \min_{u_k \in U_k(x^s_k)} \left\{ \mathbb{E}_{w_k} \left[ g_k(x^s_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x^s_k, u_k, w_k), \theta^*_{k+1}) \right] \right\} \right)^2 \right\}$$

- Where use the optimal architecture of the future stage to provide the cost-to-go values.

# Fitted Value Iteration: finite horizon case

- Likewise, if we let:

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

- We write the k'th step of the algorithm as the regression:

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

- The algorithm keeps going backwards until we compute $\theta_0^*$ at time period 0.

# Fitted Value Iteration: finite horizon case

- A few notes are warranted.

- (1) We can use Simulation (e.g.: generating scenarios) for the disturbances $w_k's$ at each step of the algorithm.
  - Thus, using sample average approximation (SAA) to compute the expecations.

- (2) At each period $k$ The generated samples of states $x_k^1, \dots, x_k^S$ need to "represent" the entire state-space.
  - We will study this notion of obtain a "rich" array of samples in the infinite horizon context.

- (3) The computation on N is large can be excessive, since we are training N different architectures.
  - By moving to the infinite horizon case, we will argue about using stationarity to provide a single DNN architecture that will be used for all time periods.