Feature-Based Aggregation and Deep Reinforcement Learning

Dimitri P. Bertsekas

Laboratory for Information and Decision Systems
Massachusetts Institute of Technology

Arizona State University

April 2018
AlphaZero

Plays much better than all chess programs
Plays different!
Learned from scratch ... with 4 hours of training!
Same algorithm learned multiple games (Go, Shogi)
AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a “probabilistic ranking” of the possible moves
- A player’s games are used to train an “improved” player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS
AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a "probabilistic ranking" of the possible moves
- A player’s games are used to train an "improved" player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS
AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a "probabilistic ranking" of the possible moves
- A player’s games are used to train an “improved” player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS
AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a "probabilistic ranking" of the possible moves
- A player’s games are used to train an “improved” player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS
AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a "probabilistic ranking" of the possible moves
- A player’s games are used to train an "improved" player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS
AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a "probabilistic ranking" of the possible moves
- A player’s games are used to train an “improved" player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS
AlphaZero Training Structure

AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a “probabilistic ranking" of the possible moves
- A player’s games are used to train an “improved" player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro,1992), but is more complicated because of the MCTS
AlphaZero implements a form of policy iteration/approximate DP method

- Generates a sequence of players/policies, each implemented by a neural net
- The neural net of a player/policy provides at any position: the "value" of the position, and a "probabilistic ranking" of the possible moves
- A player’s games are used to train an "improved" player (self-learning)
- The games of a player are generated by Monte-Carlo Tree Search (MCTS, a form of randomized multistep lookahead)
- Training uses a form of regression
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $\leftrightarrow$ Stochastic/two player games
- Combinatorial optimization $\leftrightarrow$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker \( \leftrightarrow \) Stochastic/two player games
- Combinatorial optimization \( \leftrightarrow \) Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $\longleftrightarrow$ Stochastic/two player games
- Combinatorial optimization $\longleftrightarrow$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
 Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $\leftrightarrow$ Stochastic/two player games
- Combinatorial optimization $\leftrightarrow$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

 Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $\leftarrow\rightarrow$ Stochastic/two player games
- Combinatorial optimization $\leftarrow\rightarrow$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ... 
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $\leftrightarrow$ Stochastic/two player games
- Combinatorial optimization $\leftrightarrow$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $\leftarrow\rightarrow$ Stochastic/two player games
- Combinatorial optimization $\leftarrow\rightarrow$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
  - ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $\longleftrightarrow$ Stochastic/two player games
- Combinatorial optimization $\longleftrightarrow$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - Approximation (to reduce dimension)
  - Simulation (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
Exact DP applies to a very broad range of optimization problems

- Deterministic/one decision maker $$\leftarrow$$ Stochastic/two player games
- Combinatorial optimization $$\leftarrow$$ Optimal control w/ infinite state/control spaces
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/Reinforcement Learning

- Overcomes the difficulties of exact DP by using:
  - **Approximation** (to reduce dimension)
  - **Simulation** (in place of a math model)
- Can be applied to very challenging/large scale problems
- Has proved itself in many fields ...
- ... BUT implementation is a challenge/art and success is not guaranteed
- Still there is theory that guides the art
A Summary

Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton [TD(\(\lambda\)), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- Math framework is DP (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- No dominant method. Some ideas are solid and some are heuristic
- Success depends on finding the right mix of implementation ideas, and using massive computational power
- The AlphaZero program combines in a skillful way ideas that have been known since around 2005
A Summary

Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early90s**: Approximation and simulation-based methods: Barto/Sutton [TD($\lambda$), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- Math framework is DP (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- No dominant method. Some ideas are solid and some are heuristic
- Success depends on finding the right mix of implementation ideas, and using massive computational power
- The AlphaZero program combines in a skillful way ideas that have been known since around 2005
Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton [TD($\lambda$), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
  - Late 90s-Present: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- Math framework is DP (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- No dominant method. Some ideas are solid and some are heuristic
- Success depends on finding the right mix of implementation ideas, and using massive computational power
- The AlphaZero program combines in a skillful way ideas that have been known since around 2005
### A Summary

#### Some History
- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early90s**: Approximation and simulation-based methods: Barto/Sutton [TD(\(\lambda\)), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

#### Methodology
- Math framework is DP (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- No dominant method. Some ideas are solid and some are heuristic
- Success depends on finding the right mix of implementation ideas, and using massive computational power
- The AlphaZero program combines in a skillful way ideas that have been known since around 2005
A Summary

Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton [TD($\lambda$), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- **Math framework is DP** (plus function approximation, training by simulation)
  - Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
  - Supervised learning vs unsupervised learning (or self-learning)
  - No dominant method. Some ideas are solid and some are heuristic
  - Success depends on finding the right mix of implementation ideas, and using massive computational power
  - The AlphaZero program combines in a skillful way ideas that have been known since around 2005
Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton [TD($\lambda$), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- **Math framework is DP** (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- No dominant method. Some ideas are solid and some are heuristic
- Success depends on finding the right mix of implementation ideas, and using massive computational power
- The AlphaZero program combines in a skillful way ideas that have been known since around 2005
A Summary

Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton \([\text{TD}(\lambda)]\), AI-DP connection, Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- **Math framework is DP** (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
  - No dominant method. Some ideas are solid and some are heuristic
  - Success depends on finding the right mix of implementation ideas, and using massive computational power
  - The AlphaZero program combines in a skillful way ideas that have been known since around 2005
A Summary

Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton [TD(\(\lambda\)), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- **Math framework is DP** (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- **No dominant method**. Some ideas are solid and some are heuristic
  - Success depends on finding the right mix of implementation ideas, and using massive computational power
  - The AlphaZero program combines in a skillful way ideas that have been known since around 2005
A Summary

Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton [TD(\(\lambda\)), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- **Math framework is DP** (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- No dominant method. Some ideas are solid and some are heuristic
- Success depends on finding the right mix of implementation ideas, and using massive computational power
- The AlphaZero program combines in a skillful way ideas that have been known since around 2005
A Summary

Some History

- **1950s-60s**: Bellman (DP), Shannon (chess), Samuel (checkers)
- **80s-early 90s**: Approximation and simulation-based methods: Barto/Sutton [TD(λ), AI-DP connection], Watkins (Q-learning), Tesauro (backgammon, self-learning)
- **1990s**: Rigorous analysis, mathematical understanding, first books
- **Late 90s-Present**: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Model Predictive Control

Methodology

- **Math framework is DP** (plus function approximation, training by simulation)
- Approximations in value space and in policy space (compact/low-dimensional, feature-based approximation architectures)
- Supervised learning vs unsupervised learning (or self-learning)
- **No dominant method**. Some ideas are solid and some are heuristic
- Success depends on **finding the right mix of implementation ideas**, and using massive computational power
- The AlphaZero program combines in a skillful way ideas that have been known since around 2005
Purpose of this Talk

Selectively survey the state of the art with focus on:

- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:

- Provide an approximation architecture for the cost function of a policy
- Automatically construct the features of the architecture using self-generated data
- Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Purpose of this Talk

Selectively survey the state of the art with focus on:
- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:
- Provide an approximation architecture for the cost function of a policy
- Automatically construct the features of the architecture using self-generated data
- Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Purpose of this Talk

Selectively survey the state of the art with focus on:

- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:

- Provide an approximation architecture for the cost function of a policy
- Automatically construct the features of the architecture using self-generated data
- Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Purpose of this Talk

Selectively survey the state of the art with focus on:
- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:
- Provide an approximation architecture for the cost function of a policy
- Automatically construct the features of the architecture using self-generated data
- Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Purpose of this Talk

Selectively survey the state of the art with focus on:
- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:
- Provide an approximation architecture for the cost function of a policy
  - Automatically construct the features of the architecture using self-generated data
  - Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Selectively survey the state of the art with focus on:

- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:

- Provide an approximation architecture for the cost function of a policy
- Automatically construct the features of the architecture using self-generated data
- Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Purpose of this Talk

Selectively survey the state of the art with focus on:

- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:

- Provide an approximation architecture for the cost function of a policy
- Automatically construct the features of the architecture using self-generated data
- Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Selectively survey the state of the art with focus on:
- Approximate policy iteration
- Neural network implementations
- Aggregation

Describe the relevant contributions of neural networks:
- Provide an approximation architecture for the cost function of a policy
- Automatically construct the features of the architecture using self-generated data
- Use in neural network-based policy iteration

Describe the feature-based aggregation methodology, and how it can be used in combination with neural nets
Survey paper


DP/RL Book references

- Bertsekas and Tsitsiklis, Neuro-Dynamic Programming, 1996
- Sutton and Barto, Reinforcement Learning, 1998 (2nd ed. on-line, 2018)

My latest theoretical monograph on DP

Bertsekas, Abstract Dynamic Programming: 2nd edition, 2018
Survey paper


DP/RL Book references

- Bertsekas and Tsitsiklis, Neuro-Dynamic Programming, 1996
- Sutton and Barto, Reinforcement Learning, 1998 (2nd ed. on-line, 2018)

My latest theoretical monograph on DP

Bertsekas, Abstract Dynamic Programming: 2nd edition, 2018
Survey paper


DP/RL Book references

- Bertsekas and Tsitsiklis, Neuro-Dynamic Programming, 1996
- Sutton and Barto, Reinforcement Learning, 1998 (2nd ed. on-line, 2018)

My latest theoretical monograph on DP

Bertsekas, Abstract Dynamic Programming: 2nd edition, 2018
Survey paper


DP/RL Book references

- Bertsekas and Tsitsiklis, Neuro-Dynamic Programming, 1996
- Sutton and Barto, Reinforcement Learning, 1998 (2nd ed. on-line, 2018)

My latest theoretical monograph on DP

Bertsekas, Abstract Dynamic Programming: 2nd edition, 2018
References

Survey paper

DP/RL Book references
- Bertsekas and Tsitsiklis, Neuro-Dynamic Programming, 1996
- Sutton and Barto, Reinforcement Learning, 1998 (2nd ed. on-line, 2018)

My latest theoretical monograph on DP
Bertsekas, Abstract Dynamic Programming: 2nd edition, 2018
RL uses Max/Value, DP uses Min/Cost

- Reward of a stage = (Opposite of) Cost of a stage.
- State value = (Opposite of) State cost.
- Value (or state-value) function = (Opposite of) Cost function.

Markov chain terminology

- Agent = Controller or decision maker.
- Action = Control.
- Environment = System.

Methods terminology

- Learning = Solving a DP problem using simulation.
- Self-learning (or self-play in the context of games) = Solving a DP problem using simulation-based policy iteration.
- Planning vs Learning distinction = Solving a DP problem with math model-based vs model-free simulation.
- Prediction = Policy evaluation.
RL uses Max/Value, DP uses Min/Cost

- **Reward of a stage** = (Opposite of) Cost of a stage.
- **State value** = (Opposite of) State cost.
- **Value (or state-value) function** = (Opposite of) Cost function.

Markov chain terminology

- **Agent** = Controller or decision maker.
- **Action** = Control.
- **Environment** = System.

Methods terminology

- **Learning** = Solving a DP problem using simulation.
- **Self-learning (or self-play in the context of games)** = Solving a DP problem using simulation-based policy iteration.
- **Planning vs Learning distinction** = Solving a DP problem with math model-based vs model-free simulation.
- **Prediction** = Policy evaluation.
## Relations and Terminology in RL/AI and DP/Control

### RL uses Max/Value, DP uses Min/Cost
- **Reward of a stage** = (Opposite of) Cost of a stage.
- **State value** = (Opposite of) State cost.
- **Value (or state-value) function** = (Opposite of) Cost function.

### Markov chain terminology
- **Agent** = Controller or decision maker.
- **Action** = Control.
- **Environment** = System.

### Methods terminology
- **Learning** = Solving a DP problem using simulation.
- **Self-learning (or self-play in the context of games)** = Solving a DP problem using simulation-based policy iteration.
- **Planning vs Learning distinction** = Solving a DP problem with math model-based vs model-free simulation.
- **Prediction** = Policy evaluation.
### Relations and Terminology in RL/AI and DP/Control

<table>
<thead>
<tr>
<th><strong>RL uses Max/Value, DP uses Min/Cost</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <strong>Reward of a stage</strong> = (Opposite of) Cost of a stage.</td>
</tr>
<tr>
<td>- <strong>State value</strong> = (Opposite of) State cost.</td>
</tr>
<tr>
<td>- <strong>Value (or state-value) function</strong> = (Opposite of) Cost function.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Markov chain terminology</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <strong>Agent</strong> = Controller or decision maker.</td>
</tr>
<tr>
<td>- <strong>Action</strong> = Control.</td>
</tr>
<tr>
<td>- <strong>Environment</strong> = System.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Methods terminology</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- <strong>Learning</strong> = Solving a DP problem using simulation.</td>
</tr>
<tr>
<td>- <strong>Self-learning (or self-play in the context of games)</strong> = Solving a DP problem using simulation-based policy iteration.</td>
</tr>
<tr>
<td>- <strong>Planning vs Learning distinction</strong> = Solving a DP problem with math model-based vs model-free simulation.</td>
</tr>
<tr>
<td>- <strong>Prediction</strong> = Policy evaluation.</td>
</tr>
</tbody>
</table>
### Relations and Terminology in RL/AI and DP/Control

<table>
<thead>
<tr>
<th>RL uses Max/Value, DP uses Min/Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Reward of a stage = (Opposite of) Cost of a stage.</td>
</tr>
<tr>
<td>- State value = (Opposite of) State cost.</td>
</tr>
<tr>
<td>- Value (or state-value) function = (Opposite of) Cost function.</td>
</tr>
</tbody>
</table>

### Markov chain terminology

| Agent = Controller or decision maker. |
| Action = Control. |
| Environment = System. |

### Methods terminology

| Learning = Solving a DP problem using simulation. |
| Self-learning (or self-play in the context of games) = Solving a DP problem using simulation-based policy iteration. |
| Planning vs Learning distinction = Solving a DP problem with math model-based vs model-free simulation. |
| Prediction = Policy evaluation. |
### Relations and Terminology in RL/AI and DP/Control

#### RL uses Max/Value, DP uses Min/Cost
- **Reward of a stage** = (Opposite of) Cost of a stage.
- **State value** = (Opposite of) State cost.
- **Value (or state-value) function** = (Opposite of) Cost function.

#### Markov chain terminology
- **Agent** = Controller or decision maker.
- **Action** = Control.
- **Environment** = System.

#### Methods terminology
- **Learning** = Solving a DP problem using simulation.
- **Self-learning (or self-play in the context of games)** = Solving a DP problem using simulation-based policy iteration.
- **Planning vs Learning distinction** = Solving a DP problem with math model-based vs model-free simulation.
- **Prediction** = Policy evaluation.
RL uses Max/Value, DP uses Min/Cost

- **Reward of a stage** = (Opposite of) Cost of a stage.
- **State value** = (Opposite of) State cost.
- **Value (or state-value) function** = (Opposite of) Cost function.

Markov chain terminology

- **Agent** = Controller or decision maker.
- **Action** = Control.
- **Environment** = System.

Methods terminology

- **Learning** = Solving a DP problem using simulation.
- **Self-learning (or self-play in the context of games)** = Solving a DP problem using simulation-based policy iteration.
- **Planning vs Learning distinction** = Solving a DP problem with math model-based vs model-free simulation.
- **Prediction** = Policy evaluation.
Relations and Terminology in RL/AI and DP/Control

RL uses Max/Value, DP uses Min/Cost

- Reward of a stage = (Opposite of) Cost of a stage.
- State value = (Opposite of) State cost.
- Value (or state-value) function = (Opposite of) Cost function.

Markov chain terminology

- Agent = Controller or decision maker.
- Action = Control.
- Environment = System.

Methods terminology

- Learning = Solving a DP problem using simulation.
- Self-learning (or self-play in the context of games) = Solving a DP problem using simulation-based policy iteration.
- Planning vs Learning distinction = Solving a DP problem with math model-based vs model-free simulation.
- Prediction = Policy evaluation.
### Relations and Terminology in RL/AI and DP/Control

<table>
<thead>
<tr>
<th>RL uses Max/Value, DP uses Min/Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reward of a stage</strong> = (Opposite of) Cost of a stage.</td>
</tr>
<tr>
<td><strong>State value</strong> = (Opposite of) State cost.</td>
</tr>
<tr>
<td><strong>Value (or state-value) function</strong> = (Opposite of) Cost function.</td>
</tr>
</tbody>
</table>

### Markov chain terminology

- **Agent** = Controller or decision maker.
- **Action** = Control.
- **Environment** = System.

### Methods terminology

- **Learning** = Solving a DP problem using simulation.
- **Self-learning (or self-play in the context of games)** = Solving a DP problem using simulation-based policy iteration.
- **Planning vs Learning distinction** = Solving a DP problem with math model-based vs model-free simulation.
- **Prediction** = Policy evaluation.
RL uses Max/Value, DP uses Min/Cost

- **Reward of a stage** = (Opposite of) Cost of a stage.
- **State value** = (Opposite of) State cost.
- **Value (or state-value) function** = (Opposite of) Cost function.

Markov chain terminology

- **Agent** = Controller or decision maker.
- **Action** = Control.
- **Environment** = System.

Methods terminology

- **Learning** = Solving a DP problem using simulation.
- **Self-learning (or self-play in the context of games)** = Solving a DP problem using simulation-based policy iteration.
- **Planning vs Learning distinction** = Solving a DP problem with math model-based vs model-free simulation.
- **Prediction** = Policy evaluation.
Outline

1. **Exact and Approximate Policy Iteration**
2. **Approximate PI with Neural Nets**
3. **Feature-Based Aggregation**
4. **Feature-Based Aggregation with Neural Networks**
1. Exact and Approximate Policy Iteration
2. Approximate PI with Neural Nets
3. Feature-Based Aggregation
4. Feature-Based Aggregation with Neural Networks
Outline

1. Exact and Approximate Policy Iteration
2. Approximate PI with Neural Nets
3. Feature-Based Aggregation
4. Feature-Based Aggregation with Neural Networks
Outline

1. Exact and Approximate Policy Iteration
2. Approximate PI with Neural Nets
3. Feature-Based Aggregation
4. Feature-Based Aggregation with Neural Networks
Outline

1. Exact and Approximate Policy Iteration
2. Approximate PI with Neural Nets
3. Feature-Based Aggregation
4. Feature-Based Aggregation with Neural Networks
Discounted Infinite Horizon Problem

Transition probabilities $p_{ij}(u)$

Cost $\alpha^k g(i, u, j)$

Controlled Markov Chain

A Markov chain with states 1, ..., $n$, and control $u$

- $p_{ij}(u)$: Transition probability from $i$ to $j$ under $u$
- $\alpha^k g(i, u, j)$: Cost of the $k$th transition; $\alpha \in (0, 1)$: discount factor

Policy (or feedback controller) $\mu$: A mapping of each state $i$ to a control $\mu(i)$

- Total cost of $\mu$ starting at $i_0$: $J_{\mu}(i_0) = E \{ \sum_{k=0}^{\infty} \alpha^k g(i_k, \mu(i_k), i_{k+1}) \}$
- Optimal cost starting at $i_0$: $J^*(i_0) = \min_{\mu} J_{\mu}(i_0)$
- Optimal policy $\mu^*$: Satisfies $J_{\mu^*}(i) = J^*(i)$ for all $i$
Discounted Infinite Horizon Problem

Transition probabilities $p_{ij}(u)$
Cost $\alpha^k g(i, u, j)$

Controlled Markov Chain

A Markov chain with states 1, \ldots, n, and control $u$

- $p_{ij}(u)$: Transition probability from $i$ to $j$ under $u$
- $\alpha^k g(i, u, j)$: Cost of the $k$th transition; $\alpha \in (0, 1)$: discount factor

Policy (or feedback controller) $\mu$: A mapping of each state $i$ to a control $\mu(i)$

- Total cost of $\mu$ starting at $i_0$: $J_\mu(i_0) = E \left\{ \sum_{k=0}^{\infty} \alpha^k g(i_k, \mu(i_k), i_{k+1}) \right\}$
- Optimal cost starting at $i_0$: $J^*(i_0) = \min_{\mu} J_\mu(i_0)$
- Optimal policy $\mu^*$: Satisfies $J_{\mu^*}(i) = J^*(i)$ for all $i$
Discounted Infinite Horizon Problem

Transition probabilities $p_{ij}(u)$
Cost $\alpha^k g(i, u, j)$

Controlled Markov Chain

A Markov chain with states $1, \ldots, n$, and control $u$

- $p_{ij}(u)$: Transition probability from $i$ to $j$ under $u$
- $\alpha^k g(i, u, j)$: Cost of the $k$th transition; $\alpha \in (0, 1)$: discount factor

Policy (or feedback controller) $\mu$: A mapping of each state $i$ to a control $\mu(i)$

- Total cost of $\mu$ starting at $i_0$: $J_\mu(i_0) = E \left\{ \sum_{k=0}^{\infty} \alpha^k g(i_k, \mu(i_k), i_{k+1}) \right\}$
- Optimal cost starting at $i_0$: $J^*(i_0) = \min_\mu J_\mu(i_0)$
- Optimal policy $\mu^*$: Satisfies $J_{\mu^*}(i) = J^*(i)$ for all $i$
Discounted Infinite Horizon Problem

Transition probabilities $p_{ij}(u)$
Cost $\alpha^k g(i, u, j)$

Controlled Markov Chain

A Markov chain with states 1, \ldots, $n$, and control $u$

- $p_{ij}(u)$: Transition probability from $i$ to $j$ under $u$
- $\alpha^k g(i, u, j)$: Cost of the $k$th transition; $\alpha \in (0, 1)$: discount factor

Policy (or feedback controller) $\mu$: A mapping of each state $i$ to a control $\mu(i)$

- Total cost of $\mu$ starting at $i_0$: $J_\mu(i_0) = E \{ \sum_{k=0}^{\infty} \alpha^k g(i_k, \mu(i_k), i_{k+1}) \}$
- Optimal cost starting at $i_0$: $J^*(i_0) = \min_{\mu} J_\mu(i_0)$
- Optimal policy $\mu^*$: Satisfies $J_{\mu^*}(i) = J^*(i)$ for all $i$
Discounted Infinite Horizon Problem

Transition probabilities $p_{ij}(u)$

Cost $\alpha^k g(i, u, j)$

Controlled Markov Chain

A Markov chain with states 1, \ldots, $n$, and control $u$

- $p_{ij}(u)$: Transition probability from $i$ to $j$ under $u$
- $\alpha^k g(i, u, j)$: Cost of the $k$th transition; $\alpha \in (0, 1)$: discount factor

Policy (or feedback controller) $\mu$: A mapping of each state $i$ to a control $\mu(i)$

- Total cost of $\mu$ starting at $i_0$: $J_\mu(i_0) = E \left\{ \sum_{k=0}^{\infty} \alpha^k g(i_k, \mu(i_k), i_{k+1}) \right\}$
- Optimal cost starting at $i_0$: $J^*(i_0) = \min_{\mu} J_\mu(i_0)$
- Optimal policy $\mu^*$: Satisfies $J_{\mu^*}(i) = J^*(i)$ for all $i$
Discounted Infinite Horizon Problem

Transition probabilities $p_{ij}(u)$

Cost $\alpha^k g(i, u, j)$

Controlled Markov Chain

A Markov chain with states $1, \ldots, n$, and control $u$

- $p_{ij}(u)$: Transition probability from $i$ to $j$ under $u$
- $\alpha^k g(i, u, j)$: Cost of the $k$th transition; $\alpha \in (0, 1)$: discount factor

Policy (or feedback controller) $\mu$: A mapping of each state $i$ to a control $\mu(i)$

- Total cost of $\mu$ starting at $i_0$: $J_\mu(i_0) = E \left\{ \sum_{k=0}^{\infty} \alpha^k g(i_k, \mu(i_k), i_{k+1}) \right\}$
- Optimal cost starting at $i_0$: $J^*(i_0) = \min_\mu J_\mu(i_0)$
- Optimal policy $\mu^*$: Satisfies $J_{\mu^*}(i) = J^*(i)$ for all $i$
Discounted Infinite Horizon Problem

Transition probabilities $p_{ij}(u)$

Cost $\alpha^k g(i, u, j)$

Controlled Markov Chain

A Markov chain with states 1, $\ldots$, $n$, and control $u$

- $p_{ij}(u)$: Transition probability from $i$ to $j$ under $u$
- $\alpha^k g(i, u, j)$: Cost of the $k$th transition; $\alpha \in (0, 1)$: discount factor

Policy (or feedback controller) $\mu$: A mapping of each state $i$ to a control $\mu(i)$

- Total cost of $\mu$ starting at $i_0$: $J_\mu(i_0) = E \left\{ \sum_{k=0}^{\infty} \alpha^k g(i_k, \mu(i_k), i_{k+1}) \right\}$
- Optimal cost starting at $i_0$: $J^*(i_0) = \min_\mu J_\mu(i_0)$
- Optimal policy $\mu^*$: Satisfies $J_{\mu^*}(i) = J^*(i)$ for all $i$
Bellman’s equation for $J^*$

$$J^*(i) = \min_u \sum_{i=1}^{n} p_{ij}(u) \{ g(i, u, j) + \alpha J^*(j) \}, \quad \text{for all } i$$

Optimal cost at $i = \min_u E\{\text{1st stage exp. cost + optimal cost of remaining stages}\}$

Policy evaluation (Bellman) equation for the cost function $J_\mu$ of a given policy $\mu$

$$J_\mu(i) = \sum_{i=1}^{n} p_{ij}(\mu(i)) \{ g(i, \mu(i), j) + \alpha J_\mu(j) \}, \quad \text{for all } i$$

Policy improvement principle

Given a policy $\mu$ and its evaluation $J_\mu$, we can obtain an improved policy $\hat{\mu}$ through

$$\hat{\mu}(i) = \arg \min_u \sum_{i=1}^{n} p_{ij}(u) \{ g(i, u, j) + \alpha J_\mu(j) \}, \quad \text{for all } i$$

We have $J_{\hat{\mu}}(i) \leq J_\mu(i)$ for all $i$
Bellman’s equation for $J^*$

$$J^*(i) = \min_u \sum_{i=1}^n p_{ij}(u) \{ g(i, u, j) + \alpha J^*(j) \}, \quad \text{for all } i$$

Optimal cost at $i = \min_u E\{1\text{st stage exp. cost} + \text{optimal cost of remaining stages}\}$

Policy evaluation (Bellman) equation for the cost function $J_\mu$ of a given policy $\mu$

$$J_\mu(i) = \sum_{i=1}^n p_{ij}(\mu(i)) \{ g(i, \mu(i), j) + \alpha J_\mu(j) \}, \quad \text{for all } i$$

Policy improvement principle

Given a policy $\mu$ and its evaluation $J_\mu$, we can obtain an improved policy $\hat{\mu}$ through

$$\hat{\mu}(i) = \arg \min_u \sum_{i=1}^n p_{ij}(u) \{ g(i, u, j) + \alpha J_\mu(j) \}, \quad \text{for all } i$$

We have $J_{\hat{\mu}}(i) \leq J_\mu(i)$ for all $i$
Basic Theory

Bellman’s equation for $J^*$

$$J^*(i) = \min_u \sum_{i=1}^{n} p_{ij}(u) \{ g(i, u, j) + \alpha J^*(j) \}, \quad \text{for all } i$$

Optimal cost at $i = \min_u E\{1\text{st stage exp. cost} + \text{optimal cost of remaining stages}\}$

Policy evaluation (Bellman) equation for the cost function $J_\mu$ of a given policy $\mu$

$$J_\mu(i) = \sum_{i=1}^{n} p_{ij}(\mu(i)) \{ g(i, \mu(i), j) + \alpha J_\mu(j) \}, \quad \text{for all } i$$

Policy improvement principle

Given a policy $\mu$ and its evaluation $J_\mu$, we can obtain an improved policy $\hat{\mu}$ through

$$\hat{\mu}(i) = \arg\min_u \sum_{i=1}^{n} p_{ij}(u) \{ g(i, u, j) + \alpha J_\mu(j) \}, \quad \text{for all } i$$

We have $J_{\hat{\mu}}(i) \leq J_\mu(i)$ for all $i$
Exact and Approximate Policy Iteration (PI)

Exact policy iteration is successive policy improvement:

\[ \mu_0 \Rightarrow \mu_1 : \text{improved policy over } \mu_0 \Rightarrow \mu_2 : \text{improved policy over } \mu_1 \Rightarrow \cdots \]

Converges to an optimal policy.

Approximate policy iteration is policy improvement w/ approximate evaluation:

\[ \mu_0 \Rightarrow \mu_1 : \text{"improved" policy over } \mu_0 \Rightarrow \mu_2 : \text{"improved" policy over } \mu_1 \Rightarrow \cdots \]

"Converges" to optimum within an error bound [of order \( O((1 - \alpha)^2) \) or \( O((1 - \alpha)) \)]. Possibility of multistep lookahead and Monte Carlo tree search (not discussed here).
Exact policy iteration is successive policy improvement:

\[ \mu_0 \Rightarrow \mu_1 : \text{improved policy over } \mu_0 \Rightarrow \mu_2 : \text{improved policy over } \mu_1 \Rightarrow \cdots \]

Converges to an optimal policy.

Approximate policy iteration is policy improvement w/ approximate evaluation:

\[ \mu_0 \Rightarrow \mu_1 : \text{"improved" policy over } \mu_0 \Rightarrow \mu_2 : \text{"improved" policy over } \mu_1 \Rightarrow \cdots \]

"Converges" to optimum within an error bound [of order \( O((1 - \alpha)^2) \) or \( O((1 - \alpha)) \)]. Possibility of multistep lookahead and Monte Carlo tree search (not discussed here).
Exact and Approximate Policy Iteration (PI)

Exact policy iteration is successive policy improvement:

\[ \mu_0 \Rightarrow \mu_1 : \text{improved policy over } \mu_0 \Rightarrow \mu_2 : \text{improved policy over } \mu_1 \Rightarrow \cdots \]

Converges to an optimal policy.

Approximate policy iteration is policy improvement w/ approximate evaluation:

\[ \mu_0 \Rightarrow \hat{\mu}_1 : \text{"improved" policy over } \mu_0 \Rightarrow \mu_2 : \text{"improved" policy over } \mu_1 \Rightarrow \cdots \]

"Converges" to optimum within an error bound [of order \( O((1 - \alpha)^2) \) or \( O((1 - \alpha)) \)]. Possibility of multistep lookahead and Monte Carlo tree search (not discussed here).
Outline

1 Exact and Approximate Policy Iteration

2 Approximate PI with Neural Nets

3 Feature-Based Aggregation

4 Feature-Based Aggregation with Neural Networks
Feature-Based Policy Evaluation

\[ \tilde{J}_\mu(F(i), r) : \text{Feature-based parametric architecture} \]
\[ F(i) = (F_1(i), \ldots, F_s(i)) : \text{Vector of Features of } i \]
\[ r = (r_1, \ldots, r_s) : \text{Vector of parameters} \]
\[ \text{If } \tilde{J}_\mu(F(i)) = \sum_{\ell=1}^{s} F_\ell(i)r_\ell \]

it is a linear feature-based architecture

Approximation in a space of basis functions

---

Features provide a lower-dimensional representation/approximation of \( J_\mu \)

- The features can be viewed as basis functions
- The weights depend on \( \mu \) (sometimes the features also)
- Critical question: How to find good features?
  - Handcrafted, tailored to the problem at hand
  - Automatically, e.g., using a neural network (this is the BIG contribution of NNs)
Feature-Based Policy Evaluation

Initial Policy

Current Policy $\mu$
Evaluate Approximate Cost $\tilde{J}_\mu(F(i), r)$ of $i$

Generate “Improved” Policy $\hat{\mu}$

$\tilde{J}_\mu(F(i), r)$: Feature-based parametric architecture

$F(i) = (F_1(i), \ldots, F_s(i))$: Vector of Features of $i$

$r = (r_1, \ldots, r_s)$: Vector of parameters

If $\tilde{J}_\mu(F(i)) = \sum_{\ell=1}^{s} F_\ell(i)r_\ell$

it is a linear feature-based architecture

Approximation in a space of basis functions

Policy Improvement

Features provide a lower-dimensional representation/approximation of $J_\mu$

- The features can be viewed as basis functions
- The weights depend on $\mu$ (sometimes the features also)
- Critical question: How to find good features?
  - Handcrafted, tailored to the problem at hand
  - Automatically, e.g., using a neural network (this is the BIG contribution of NNs)
Feature-Based Policy Evaluation

\[ \tilde{J}_\mu(F(i), r) \]: Feature-based parametric architecture

\[ F(i) = (F_1(i), \ldots, F_s(i)) \]: Vector of Features of \( i \)

\[ r = (r_1, \ldots, r_s) \]: Vector of parameters

If \( \tilde{J}_\mu(F(i)) = \sum_{\ell=1}^{s} F_\ell(i)r_\ell \)

it is a linear feature-based architecture

Approximation in a space of basis functions

Policy Improvement

Features provide a lower-dimensional representation/approximation of \( J_\mu \)

- The features can be viewed as basis functions
- The weights depend on \( \mu \) (sometimes the features also)

Critical question: How to find good features?

- Handcrafted, tailored to the problem at hand
- Automatically, e.g., using a neural network (this is the BIG contribution of NNs)
Feature-Based Policy Evaluation

\[ \tilde{J}_\mu(F(i), r) : \text{Feature-based parametric architecture} \]
\[ F(i) = (F_1(i), \ldots, F_s(i)) : \text{Vector of Features of } i \]
\[ r = (r_1, \ldots, r_s) : \text{Vector of parameters} \]
\[ \text{If } \tilde{J}_\mu(F(i)) = \sum_{\ell=1}^s F_\ell(i) r_\ell \]
\[ \text{it is a linear feature-based architecture} \]
\[ \text{Approximation in a space of basis functions} \]

Policy Improvement

Features provide a lower-dimensional representation/approximation of \( J_\mu \)

- The features can be viewed as basis functions
- The weights depend on \( \mu \) (sometimes the features also)
- Critical question: How to find good features?
  - Handcrafted, tailored to the problem at hand
  - Automatically, e.g., using a neural network (this is the BIG contribution of NNs)
Feature-Based Policy Evaluation

Initial Policy

Current Policy $\mu$
Evaluate Approximate Cost $\tilde{J}_\mu(F(i), r)$ of $i$

Generate “Improved” Policy $\hat{\mu}$

$\tilde{J}_\mu(F(i), r)$: Feature-based parametric architecture
$F(i) = (F_1(i), \ldots, F_s(i))$: Vector of Features of $i$
$r = (r_1, \ldots, r_s)$: Vector of parameters

If $\tilde{J}_\mu(F(i)) = \sum_{\ell=1}^{s} F_\ell(i)r_\ell$

it is a linear feature-based architecture

Approximation in a space of basis functions

Policy Improvement

Features provide a lower-dimensional representation/approximation of $J_\mu$

- The features can be viewed as basis functions
- The weights depend on $\mu$ (sometimes the features also)
- Critical question: How to find good features?
  - Handcrafted, tailored to the problem at hand
  - Automatically, e.g., using a neural network (this is the BIG contribution of NNs)
Features provide a lower-dimensional representation/approximation of $J_\mu$

- The features can be viewed as basis functions
- The weights depend on $\mu$ (sometimes the features also)
- **Critical question:** How to find good features?
  - Handcrafted, tailored to the problem at hand
  - Automatically, e.g., using a neural network (this is the BIG contribution of NNs)
NN-Based Policy Evaluation for a Given Policy $\mu$

Generate many state-cost samples $(i_m, \beta_m), m = 1, \ldots, M, \beta_m \approx J_\mu(i_m)$

- Use nonlinear optimization/regression: Find $(v, r)$ that minimizes
  \[ \sum_{m=1}^{M} (\tilde{J}_\mu(i_m, v, r) - \beta_m)^2 \]

- For this we may use an incremental gradient method (also called SGD)
- Making the method work is an art (regularization, hot start, stepsize etc)
- Simulation is used to generate the cost samples
- Universal approximation theorem
- Alternative regressions may be used (e.g., based on temporal differences, etc)
NN-Based Policy Evaluation for a Given Policy $\mu$

**Diagram:**
- **State Encoding**
- **Linear Layer Parameter** $v = (A, b)$
- **Nonlinear Layer**
- **Features** $F_1(i, v), F_2(i, v), \ldots, F_s(i, v)$
- **Linear Weighting Parameter** $r = (r_1, \ldots, r_s)$
- **Cost Approximation** $\sum_{\ell=1}^{s} F_\ell(i, v)r_\ell$

**Generate many state-cost samples** $(i_m, \beta_m), m = 1, \ldots, M, \beta_m \approx J_\mu(i_m)$

- Use **nonlinear optimization/regression**: Find $(v, r)$ that minimizes
  $$\sum_{m=1}^{M} (\tilde{J}_\mu(i_m, v, r) - \beta_m)^2$$
- For this we may use an **incremental gradient method** (also called SGD)
- Making the method work is an art (regularization, hot start, stepsize etc)
- Simulation is used to generate the cost samples
- **Universal approximation theorem**
- Alternative regressions may be used (e.g., based on temporal differences, etc)
NN-Based Policy Evaluation for a Given Policy $\mu$

Generate many state-cost samples $(i_m, \beta_m)$, $m = 1, \ldots, M$, $\beta_m \approx J_\mu(i_m)$

- Use nonlinear optimization/regression: Find $(v, r)$ that minimizes
  \[
  \sum_{m=1}^{M} (\tilde{J}_\mu(i_m, v, r) - \beta_m)^2
  \]

- For this we may use an incremental gradient method (also called SGD)
  - Making the method work is an art (regularization, hot start, stepsize etc)
  - Simulation is used to generate the cost samples
  - Universal approximation theorem
  - Alternative regressions may be used (e.g., based on temporal differences, etc)
NN-Based Policy Evaluation for a Given Policy $\mu$

Generate many state-cost samples $(i_m, \beta_m), m = 1, \ldots, M, \beta_m \approx J_\mu(i_m)$

- Use nonlinear optimization/regression: Find $(v, r)$ that minimizes
  \[
  \sum_{m=1}^{M} (\tilde{J}_\mu(i_m, v, r) - \beta_m)^2
  \]

- For this we may use an incremental gradient method (also called SGD)
- Making the method work is an art (regularization, hot start, stepsize etc)
- Simulation is used to generate the cost samples
- Universal approximation theorem
- Alternative regressions may be used (e.g., based on temporal differences, etc)
NN-Based Policy Evaluation for a Given Policy \( \mu \)

Generate many state-cost samples \((i_m, \beta_m), m = 1, \ldots, M, \beta_m \approx J_\mu(i_m)\)

- Use **nonlinear optimization/regression**: Find \((v, r)\) that minimizes
  \[
  \sum_{m=1}^{M} \left( \tilde{J}_\mu(i_m, v, r) - \beta_m \right)^2
  \]

- For this we may use an **incremental gradient method** (also called SGD)
- Making the method work is an art (regularization, hot start, stepsize etc)
- Simulation is used to generate the cost samples
- **Universal approximation theorem**
- **Alternative regressions** may be used (e.g., based on temporal differences, etc)
NN-Based Policy Evaluation for a Given Policy $\mu$

Generate many state-cost samples $(i_m, \beta_m), m = 1, \ldots, M, \beta_m \approx J_\mu(i_m)$

- Use **nonlinear optimization/regression**: Find $(v, r)$ that minimizes
  \[
  \sum_{m=1}^{M} (\tilde{J}_\mu(i_m, v, r) - \beta_m)^2
  \]

- For this we may use an **incremental gradient method** (also called SGD)
- Making the method work is an art (regularization, hot start, stepsize etc)
- Simulation is used to generate the cost samples
- **Universal approximation theorem**
- Alternative regressions may be used (e.g., based on temporal differences, etc)
NN-Based Policy Evaluation for a Given Policy $\mu$

Generate many state-cost samples $(i_m, \beta_m), m = 1, \ldots, M, \beta_m \approx J_\mu(i_m)$

- Use **nonlinear optimization/regression**: Find $(v, r)$ that minimizes

\[
\sum_{m=1}^{M} (\tilde{J}_\mu(i_m, v, r) - \beta_m)^2
\]

- For this we may use an **incremental gradient method** (also called SGD)
- Making the method work is an art (regularization, hot start, stepsize etc)
- Simulation is used to generate the cost samples
- Universal approximation theorem
- Alternative regressions may be used (e.g., based on temporal differences, etc)

![Diagram](image-url)
A deep NN just has many layers

- Simulation and training is the same as in single layer nets (the incremental gradient method is called **backpropagation**)
- Can be viewed as providing a “hierarchy of features”
- More “sophisticated” features with each stage, fewer weights needed (?)
- The last (and most sophisticated) set of features is the one used in the cost approximation
- **Is deeper better?** Tesauro’s and subsequent backgammon implementations used one hidden layer!
- For our purposes, deeper is better. There are fewer final features in deep NNs
A deep NN just has many layers

- Simulation and training is the same as in single layer nets (the incremental gradient method is called backpropagation)
- Can be viewed as providing a “hierarchy of features"
- More “sophisticated” features with each stage, fewer weights needed (?)
- The last (and most sophisticated) set of features is the one used in the cost approximation
- Is deeper better? Tesauro’s and subsequent backgammon implementations used one hidden layer!
- For our purposes, deeper is better. There are fewer final features in deep NNs
A deep NN just has many layers

- Simulation and training is the same as in single layer nets (the incremental gradient method is called backpropagation)
- Can be viewed as providing a “hierarchy of features"
  - More “sophisticated” features with each stage, fewer weights needed (?)
  - The last (and most sophisticated) set of features is the one used in the cost approximation
- Is deeper better? Tesauro’s and subsequent backgammon implementations used one hidden layer!
- For our purposes, deeper is better. There are fewer final features in deep NNs
A deep NN just has many layers

- Simulation and training is the same as in single layer nets (the incremental gradient method is called backpropagation)
- Can be viewed as providing a “hierarchy of features"
- More “sophisticated” features with each stage, fewer weights needed (?)
- The last (and most sophisticated) set of features is the one used in the cost approximation
- Is deeper better? Tesauro’s and subsequent backgammon implementations used one hidden layer!
- For our purposes, deeper is better. There are fewer final features in deep NNs
**Use of Deep NNs**

A deep NN just has many layers

- Simulation and training is the same as in single layer nets (the incremental gradient method is called **backpropagation**)
- Can be viewed as providing a “hierarchy of features"
- More “sophisticated” features with each stage, fewer weights needed (?)
- The last (and most sophisticated) set of features is the one used in the cost approximation

- Is deeper better? Tesauro’s and subsequent backgammon implementations used one hidden layer!
- For our purposes, deeper is better. There are fewer final features in deep NNs
A deep NN just has many layers

- Simulation and training is the same as in single layer nets (the incremental gradient method is called backpropagation)
- Can be viewed as providing a “hierarchy of features"
- More “sophisticated” features with each stage, fewer weights needed (?)
- The last (and most sophisticated) set of features is the one used in the cost approximation
- **Is deeper better?** Tesauro’s and subsequent backgammon implementations used one hidden layer!
- For our purposes, deeper is better. There are fewer final features in deep NNs
A deep NN just has many layers

- Simulation and training is the same as in single layer nets (the incremental gradient method is called backpropagation)
- Can be viewed as providing a “hierarchy of features"
- More “sophisticated” features with each stage, fewer weights needed (?)
- The last (and most sophisticated) set of features is the one used in the cost approximation
- Is deeper better? Tesauro’s and subsequent backgammon implementations used one hidden layer!
- For our purposes, deeper is better. There are fewer final features in deep NNs
An old idea: Problem approximation (rather than algorithm approximation)

- Group “similar” states together and represent them as a single state
- Approximate the original DP problem with a lower-dimensional DP problem, called aggregate problem
- Solve the aggregate problem and “extend” its cost function to the original
- The aggregate problem can be solved by exact DP and simulation-based methods

A simple example: Approximate a fine grid with a coarse grid

Another example (hard aggregation): Partition the state space into disjoint subsets, each viewed as a single “aggregate state”
Basic Principles of Aggregation

An old idea: Problem approximation (rather than algorithm approximation)

- Group “similar” states together and represent them as a single state
- Approximate the original DP problem with a lower-dimensional DP problem, called aggregate problem
- Solve the aggregate problem and “extend” its cost function to the original
- The aggregate problem can be solved by exact DP and simulation-based methods

A simple example: Approximate a fine grid with a coarse grid

Another example (hard aggregation): Partition the state space into disjoint subsets, each viewed as a single “aggregate state”
**Basic Principles of Aggregation**

**An old idea: Problem approximation (rather than algorithm approximation)**

- Group “similar” states together and represent them as a single state
- Approximate the original DP problem with a lower-dimensional DP problem, called **aggregate problem**
- Solve the aggregate problem and “extend” its cost function to the original
- The aggregate problem can be solved by **exact** DP and simulation-based methods

**A simple example: Approximate a fine grid with a coarse grid**

---

**Another example (hard aggregation):** Partition the state space into disjoint subsets, each viewed as a single “aggregate state”
An old idea: Problem approximation (rather than algorithm approximation)

- Group “similar” states together and represent them as a single state
- Approximate the original DP problem with a lower-dimensional DP problem, called *aggregate problem*
- Solve the aggregate problem and “extend” its cost function to the original
- The aggregate problem can be solved by *exact* DP and simulation-based methods

A simple example: Approximate a fine grid with a coarse grid

Another example (hard aggregation): Partition the state space into disjoint subsets, each viewed as a single “aggregate state"
Basic Principles of Aggregation

An old idea: Problem approximation (rather than algorithm approximation)

- Group “similar” states together and represent them as a single state
- Approximate the original DP problem with a lower-dimensional DP problem, called aggregate problem
- Solve the aggregate problem and “extend” its cost function to the original
- The aggregate problem can be solved by exact DP and simulation-based methods

A simple example: Approximate a fine grid with a coarse grid

Another example (hard aggregation): Partition the state space into disjoint subsets, each viewed as a single “aggregate state"
Basic Principles of Aggregation

An old idea: Problem approximation (rather than algorithm approximation)

- Group “similar” states together and represent them as a single state
- Approximate the original DP problem with a lower-dimensional DP problem, called aggregate problem
- Solve the aggregate problem and “extend” its cost function to the original
- The aggregate problem can be solved by exact DP and simulation-based methods

A simple example: Approximate a fine grid with a coarse grid

Another example (hard aggregation): Partition the state space into disjoint subsets, each viewed as a single “aggregate state"
Idea: Group together states with “similar” features (i.e., small variation of $F$)

Aggregate states: Disjoint subsets $S_1, \ldots, S_q$ of state-feature pairs $(i, F(i))$

- System states $j$ relate to the aggregate states according to “membership/interpolation weights” $\phi_{1\ell}, \ldots, \phi_{n\ell}$ (called aggregation probabilities)
- Each aggregate state $S_\ell$ relates to its subset of states $I_\ell = \{ i \mid (i, F(i)) \in S_\ell \}$ according to “importance weights” $d_{\ell 1}, \ldots, d_{\ell n}$ (called disaggregation probabilities)
- Constraints:
  - If $j \in S_\ell$ then $\phi_{j\ell} = 1$ (membership weight 1 for states in an aggregate state)
  - If $i \notin I_\ell$ then $d_{\ell i} = 0$ (importance weight 0 for states outside an aggregate state)
Idea: Group together states with “similar” features (i.e., small variation of $F$)

Aggregate states: Disjoint subsets $S_1, \ldots, S_q$ of state-feature pairs $(i, F(i))$

- System states $j$ relate to the aggregate states according to “membership/interpolation weights” $\phi_{1\ell}, \ldots, \phi_{n\ell}$ (called aggregation probabilities)
- Each aggregate state $S_\ell$ relates to its subset of states $I_\ell = \{ i \mid (i, F(i)) \in S_\ell \}$ according to “importance weights” $d_{1\ell}, \ldots, d_{n\ell}$ (called disaggregation probabilities)
- Constraints:
  - If $j \in S_\ell$ then $\phi_{j\ell} = 1$ (membership weight 1 for states in an aggregate state)
  - If $i \notin I_\ell$ then $d_{i\ell} = 0$ (importance weight 0 for states outside an aggregate state)
Idea: Group together states with “similar” features (i.e., small variation of $F$)

Aggregate states: Disjoint subsets $S_1, \ldots, S_q$ of state-feature pairs $(i, F(i))$

- System states $j$ relate to the aggregate states according to “membership/interpolation weights” $\phi_{1\ell}, \ldots, \phi_{n\ell}$ (called aggregation probabilities)
- Each aggregate state $S_\ell$ relates to its subset of states $I_\ell = \{i \mid (i, F(i)) \in S_\ell\}$ according to “importance weights” $d_{\ell 1}, \ldots, d_{\ell n}$ (called disaggregation probabilities)

Constraints:
- If $j \in S_\ell$ then $\phi_{j\ell} = 1$ (membership weight 1 for states in an aggregate state)
- If $i \not\in I_\ell$ then $d_{\ell i} = 0$ (importance weight 0 for states outside an aggregate state)
Idea: Group together states with “similar" features (i.e., small variation of $F$)

Aggregate states: Disjoint subsets $S_1, \ldots, S_q$ of state-feature pairs $(i, F(i))$

- System states $j$ relate to the aggregate states according to “membership/interpolation weights" $\phi_1, \ldots, \phi_n$ (called aggregation probabilities)
- Each aggregate state $S_\ell$ relates to its subset of states $l_\ell = \{i \mid (i, F(i)) \in S_\ell\}$ according to “importance weights" $d_{\ell 1}, \ldots, d_{\ell n}$ (called disaggregation probabilities)
- Constraints:
  - If $j \in S_\ell$ then $\phi_j = 1$ (membership weight 1 for states in an aggregate state)
  - If $i \notin l_\ell$ then $d_{\ell i} = 0$ (importance weight 0 for states outside an aggregate state)
Idea: Group together states with "similar" features (i.e., small variation of $F$)

Aggregate states: Disjoint subsets $S_1, \ldots, S_q$ of state-feature pairs $(i, F(i))$

- System states $j$ relate to the aggregate states according to "membership/interpolation weights" $\phi_1, \ldots, \phi_n$ (called aggregation probabilities)
- Each aggregate state $S_\ell$ relates to its subset of states $I_\ell = \{i \mid (i, F(i)) \in S_\ell\}$ according to "importance weights" $d_{\ell 1}, \ldots, d_{\ell n}$ (called disaggregation probabilities)
- Constraints:
  - If $j \in S_\ell$ then $\phi_{j \ell} = 1$ (membership weight 1 for states in an aggregate state)
  - If $i \notin I_\ell$ then $d_{\ell i} = 0$ (importance weight 0 for states outside an aggregate state)
States: Aggregate states plus two copies of the original system states

Costs and transition probabilities: As shown

Optimal costs: $r^*_\ell$ for aggregate state $S_\ell$, $\tilde{J}_0(i)$ for left state $i$, $\tilde{J}_1(j)$ for right state $j$

By Bellman's equation for the aggregate problem we have

$$\tilde{J}_1(j) = \sum_{\ell=1}^{q} \phi_{j\ell} r^*_\ell, \quad j = 1, \ldots, n \quad \text{(piecewise linear)}$$

Once we compute $r^*_\ell$, we can obtain an "improved" policy

$$\hat{\mu}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{\ell=1}^{q} \phi_{j\ell} r^*_\ell \right), \quad i = 1, \ldots, n$$
Aggregate DP Problem: Approximation through Features

- **States**: Aggregate states plus two copies of the original system states
- **Costs and transition probabilities**: As shown
  - Optimal costs: \( r^*_\ell \) for aggregate state \( S_\ell \), \( \tilde{J}_0(i) \) for left state \( i \), \( \tilde{J}_1(j) \) for right state \( j \)
  - By Bellman's equation for the aggregate problem we have
    \[
    \tilde{J}_1(j) = \sum_{\ell=1}^{q} \phi_{j\ell} r^*_\ell, \quad j = 1, \ldots, n \quad \text{(piecewise linear)}
    \]
  - Once we compute \( r^*_\ell \), we can obtain an “improved” policy
    \[
    \hat{\mu}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{\ell=1}^{q} \phi_{j\ell} r^*_\ell \right), \quad i = 1, \ldots, n
    \]
States: Aggregate states plus two copies of the original system states

Costs and transition probabilities: As shown

Optimal costs: $r^*_\ell$ for aggregate state $S_\ell$, $\tilde{J}_0(i)$ for left state $i$, $\tilde{J}_1(j)$ for right state $j$

By Bellman's equation for the aggregate problem we have

$$\tilde{J}_1(j) = \sum_{\ell=1}^{q} \phi_{j,\ell} r^*_\ell, \quad j = 1, \ldots, n \quad \text{(piecewise linear)}$$

Once we compute $r^*_\ell$, we can obtain an "improved" policy

$$\hat{\mu}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{\ell=1}^{q} \phi_{j,\ell} r^*_\ell \right), \quad i = 1, \ldots, n$$
Aggregate DP Problem: Approximation through Features

- **States**: Aggregate states plus two copies of the original system states
- **Costs and transition probabilities**: As shown
- **Optimal costs**: $r_\ell^*$ for aggregate state $S_\ell$, $\tilde{J}_0(i)$ for left state $i$, $\tilde{J}_1(j)$ for right state $j$
- By Bellman’s equation for the aggregate problem we have
  \[
  \tilde{J}_1(j) = \sum_{\ell=1}^{q} \phi_{j\ell} r_\ell^*, \quad j = 1, \ldots, n \quad \text{(piecewise linear)}
  \]
- Once we compute $r_\ell^*$, we can obtain an “improved” policy
  \[
  \hat{\mu}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{\ell=1}^{q} \phi_{j\ell} r_\ell^* \right), \quad i = 1, \ldots, n
  \]
Aggregate DP Problem: Approximation through Features

- **States**: Aggregate states plus two copies of the original system states
- **Costs and transition probabilities**: As shown
- **Optimal costs**: $r_\ell^*$ for aggregate state $S_\ell$, $\tilde{J}_0(i)$ for left state $i$, $\tilde{J}_1(j)$ for right state $j$
- By Bellman’s equation for the aggregate problem we have
  \[
  \tilde{J}_1(j) = \sum_{\ell=1}^{q} \phi_{j\ell} r_\ell^*, \quad j = 1, \ldots, n \quad \text{(piecewise linear)}
  \]

Once we compute $r_\ell^*$, we can obtain an “improved” policy
\[
\hat{\mu}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u) \left( g(i, u, j) + \alpha \sum_{\ell=1}^{q} \phi_{j\ell} r_\ell^* \right), \quad i = 1, \ldots, n
\]
Aggregation-Based Approximate Policy Iteration

1. Initial Policy
2. Generate Features $F(i)$ of Current Policy $\mu$
3. Formulate Aggregate Problem
4. Generate “Improved” Policy $\hat{\mu}$ by “Solving” the Aggregate Problem

Use a Neural Network or Other Scheme
Possibly Include “Handcrafted” Features

Form the Aggregate States
Choose the Aggregation and Disaggregation Probabilities
Aggregation-Based Approximate Policy Iteration

1. Initial Policy

2. Generate Features $F(i)$ of Current Policy $\mu$

3. Formulate Aggregate Problem

4. Generate “Improved” Policy $\hat{\mu}$ by “Solving” the Aggregate Problem

5. Use a Neural Network or Other Scheme
   Possibly Include “Handcrafted” Features

6. Form the Aggregate States
   Choose the Aggregation and Disaggregation Probabilities

Bertsekas (M.I.T.)
Aggregation and Reinforcement Learning
Properties of the Aggregate Problem

Aggregate problem lends itself to simulation if the original problem does
- $r^*_\ell$ is computable with exact/tabular methods, e.g., TD(\(\lambda\)), LSTD, LSPE, Q-learning
- $r^*_\ell$ "roughly approximates" $J^*(i)$ for states $i$ in aggregate state $S\ell$

Intuition and analysis/error bounds suggest the following general strategy:
Find features that conform to $J^*$, i.e.,

$$F(i) \approx F(i') \implies J^*(i) \approx J^*(i')$$

Form aggregate states where $F$ varies little
Properties of the Aggregate Problem

Aggregate problem lends itself to simulation if the original problem does

- \( r^*_\ell \) is computable with exact/tabular methods, e.g., TD(\( \lambda \)), LSTD, LSPE, Q-learning
- \( r^*_\ell \) “roughly approximates” \( J^*(i) \) for states \( i \) in aggregate state \( S_\ell \)

Intuition and analysis/error bounds suggest the following general strategy:

Find features that conform to \( J^* \), i.e.,

\[
F(i) \approx F(i') \implies J^*(i) \approx J^*(i')
\]

Form aggregate states where \( F \) varies little
Properties of the Aggregate Problem

- Aggregate problem lends itself to simulation if the original problem does
- $r^*_\ell$ is computable with exact/tabular methods, e.g., TD($\lambda$), LSTD, LSPE, Q-learning
- $r^*_\ell$ “roughly approximates” $J^*(i)$ for states $i$ in aggregate state $S_\ell$

Intuition and analysis/error bounds suggest the following general strategy:

Find features that conform to $J^*$, i.e.,

$$F(i) \approx F(i') \implies J^*(i) \approx J^*(i')$$

Form aggregate states where $F$ varies little
Properties of the Aggregate Problem

Aggregate problem lends itself to simulation if the original problem does
- \( r^*_\ell \) is computable with exact/tabular methods, e.g., TD(\( \lambda \)), LSTD, LSPE, Q-learning
- \( r^*_\ell \) “roughly approximates” \( J^*(i) \) for states \( i \) in aggregate state \( S_\ell \)

Intuition and analysis/error bounds suggest the following general strategy:
Find features that conform to \( J^* \), i.e.,

\[ F(i) \approx F(i') \implies J^*(i) \approx J^*(i') \]

Form aggregate states where \( F \) varies little
Properties of the Aggregate Problem

Aggregate problem lends itself to simulation if the original problem does
- $r^*_\ell$ is computable with exact/tabular methods, e.g., TD($\lambda$), LSTD, LSPE, Q-learning
- $r^*_\ell$ “roughly approximates" $J^*(i)$ for states $i$ in aggregate state $S_\ell$

Intuition and analysis/error bounds suggest the following general strategy:
Find features that conform to $J^*$, i.e.,
\[ F(i) \approx F(i') \implies J^*(i) \approx J^*(i') \]
Form aggregate states where $F$ varies little
Suppose we have a function $V$ with “similar form” to $J^*$ (up to a constant shift).

- We can use $V$ as a feature mapping and group states with similar values of $V$.
- Each interval may contain one or multiple states.
- Many intervals lead to more accurate but more time-consuming solution.

Extend this idea to a vector of scoring functions $V(i) = (V_1(i), \ldots, V_s(i))$. 
Suppose we have a function $V$ with “similar form" to $J^*$ (up to a constant shift)

- We can use $V$ as a feature mapping and group states with similar values of $V$
- Each interval may contain one or multiple states
- Many intervals lead to more accurate but more time-consuming solution

Extend this idea to a vector of scoring functions $V(i) = (V_1(i), \ldots, V_s(i))$
Using “Scoring” Functions

Suppose we have a function $V$ with “similar form” to $J^*$ (up to a constant shift)

- We can use $V$ as a feature mapping and group states with similar values of $V$
- Each interval may contain one or multiple states
- Many intervals lead to more accurate but more time-consuming solution

Extend this idea to a vector of scoring functions $V(i) = (V_1(i), \ldots, V_s(i))$
Suppose we have a function $V$ with "similar form" to $J^*$ (up to a constant shift)

- We can use $V$ as a feature mapping and group states with similar values of $V$
- Each interval may contain one or multiple states
- Many intervals lead to more accurate but more time-consuming solution

Extend this idea to a vector of scoring functions $V(i) = (V_1(i), \ldots, V_s(i))$
Suppose we have a function $V$ with “similar form" to $J^*$ (up to a constant shift)

- We can use $V$ as a feature mapping and group states with similar values of $V$
- Each interval may contain one or multiple states
- Many intervals lead to more accurate but more time-consuming solution

Extend this idea to a vector of scoring functions $V(i) = (V_1(i), \ldots, V_s(i))$
1. Exact and Approximate Policy Iteration
2. Approximate PI with Neural Nets
3. Feature-Based Aggregation
4. Feature-Based Aggregation with Neural Networks
Approximate PI with Aggregation and Neural Nets

“Standard” NN-based PI

NN-based PI with aggregation

- Start with a training set of state-cost pairs generated using the current policy $\mu$
- Evaluate $\mu$ using the NN; obtain a feature map $F$, and a sample of $(i, F(i))$ pairs
- Construct aggregate states and a feature-based aggregate problem (essentially use $F$ as a vector scoring function, possibly with some handcrafted features)
- Use as “improved” policy $\hat{\mu}$ the optimal policy of the aggregate problem
- More work for policy improvement, but may yield better “improved” policy
Approximate PI with Aggregation and Neural Nets

"Standard" NN-based PI

NN-based PI with aggregation

- Start with a training set of state-cost pairs generated using the current policy $\mu$.
- Evaluate $\mu$ using the NN; obtain a feature map $F$, and a sample of $(i, F(i))$ pairs.
- Construct aggregate states and a feature-based aggregate problem (essentially use $F$ as a vector scoring function, possibly with some handcrafted features).
- Use as "improved" policy $\hat{\mu}$ the optimal policy of the aggregate problem.
- More work for policy improvement, but may yield better "improved" policy.
Start with a training set of state-cost pairs generated using the current policy $\mu$

- Evaluate $\mu$ using the NN; obtain a feature map $F$, and a sample of $(i, F(i))$ pairs
- Construct aggregate states and a feature-based aggregate problem (essentially use $F$ as a vector scoring function, possibly with some handcrafted features)
- Use as "improved" policy $\hat{\mu}$ the optimal policy of the aggregate problem
- More work for policy improvement, but may yield better "improved" policy
Approximate PI with Aggregation and Neural Nets

“Standard” NN-based PI

![Diagram of NN-based PI with aggregation]

- Start with a training set of state-cost pairs generated using the current policy \( \mu \)
- Evaluate \( \mu \) using the NN; obtain a feature map \( F \), and a sample of \((i, F(i))\) pairs
- Construct aggregate states and a feature-based aggregate problem (essentially use \( F \) as a vector scoring function, possibly with some handcrafted features)
- Use as “improved” policy \( \hat{\mu} \) the optimal policy of the aggregate problem
- More work for policy improvement, but may yield better “improved” policy
Approximate PI with Aggregation and Neural Nets

"Standard" NN-based PI

1. **Current Policy** $\mu$
2. **Neural Network**
3. **Feature Vector** $F(i)$
4. **Approximate Cost** $\hat{J}_\mu(F(i))$
5. **Policy Improvement**
6. **Improved Policy** $\hat{\mu}$

NN-based PI with aggregation

1. **Current Policy** $\mu$
2. **Neural Network**
3. **Feature Vector** $F(i)$
4. **Sampling**
5. **Aggregate States**
6. **Aggregate Problem Optimization**
7. **Improved Policy** $\hat{\mu}$

- Start with a training set of state-cost pairs generated using the current policy $\mu$
- Evaluate $\mu$ using the NN; obtain a feature map $F$, and a sample of $(i, F(i))$ pairs
- Construct aggregate states and a feature-based aggregate problem (essentially use $F$ as a vector scoring function, possibly with some handcrafted features)
- Use as "improved" policy $\hat{\mu}$ the optimal policy of the aggregate problem
- More work for policy improvement, but may yield better "improved" policy

**Bertsekas (M.I.T.)**

Aggregation and Reinforcement Learning

26 / 28
Approximate PI with Aggregation and Neural Nets

“Standard” NN-based PI

NN-based PI with aggregation

- Start with a training set of state-cost pairs generated using the current policy \( \mu \)
- Evaluate \( \mu \) using the NN; obtain a feature map \( F \), and a sample of \((i, F(i))\) pairs
- Construct aggregate states and a feature-based aggregate problem (essentially use \( F \) as a vector scoring function, possibly with some handcrafted features)
- Use as “improved” policy \( \hat{\mu} \) the optimal policy of the aggregate problem
- More work for policy improvement, but may yield better “improved” policy
Approximate PI with Aggregation and Neural Nets

“Standard” NN-based PI

<table>
<thead>
<tr>
<th>Current Policy $\mu$</th>
<th>Neural Network</th>
<th>Feature Vector $F(i)$</th>
<th>Approximate Cost $\hat{J}_\mu(F(i))$</th>
<th>Policy Improvement</th>
<th>Approximately Improved Policy $\hat{\mu}$</th>
</tr>
</thead>
</table>

NN-based PI with aggregation

<table>
<thead>
<tr>
<th>Current Policy $\mu$</th>
<th>Neural Network</th>
<th>Feature Vector $F(i)$</th>
<th>Sampling</th>
<th>Aggregate States</th>
<th>Aggregate Problem Optimization</th>
<th>Approximately Improved Policy $\hat{\mu}$</th>
</tr>
</thead>
</table>

- Start with a training set of state-cost pairs generated using the current policy $\mu$
- Evaluate $\mu$ using the NN; obtain a feature map $F$, and a sample of $(i, F(i))$ pairs
- Construct aggregate states and a feature-based aggregate problem (essentially use $F$ as a vector scoring function, possibly with some handcrafted features)
- Use as “improved” policy $\hat{\mu}$ the optimal policy of the aggregate problem
- More work for policy improvement, but may yield better “improved” policy
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features** of the cost function of a policy
  - Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation
  - Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!

- The RL game successes are spectacular, but they have benefited from **perfectly known and stable models** and relatively small number of controls (per state)
- On the positive side, massive computational power together with distributed computation are a source of hope
- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features of the cost function of a policy**
- **Good features, once extracted can be used for other purposes, including aggregation.** Deep NNs provide fewer final features, which favors aggregation
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!
- The RL game successes are spectacular, but they have benefited from **perfectly known and stable models and relatively small number of controls** (per state)
- On the positive side, massive computational power together with distributed computation are a source of hope
- There is an exciting journey ahead ...
Concluding Remarks

- **NNs resolve a major difficulty of approximate PI:** *Automatically extract features of the cost function of a policy*

- **Good features, once extracted can be used for other purposes, including aggregation.** Deep NNs provide fewer final features, which favors aggregation

- **Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors**

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!

- The RL game successes are spectacular, but they have benefited from perfectly known and stable models and relatively small number of controls (per state)

- On the positive side, massive computational power together with distributed computation are a source of hope

- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features of the cost function of a policy**
- Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!
- The RL game successes are spectacular, but they have benefited from perfectly known and stable models and relatively small number of controls (per state)
- On the positive side, massive computational power together with distributed computation are a source of hope
- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features of the cost function of a policy**
- Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!
- The RL game successes are spectacular, but they have benefited from perfectly known and stable models and relatively small number of controls (per state)
- On the positive side, massive computational power together with distributed computation are a source of hope
- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features of the cost function of a policy**
- **Good features, once extracted can be used for other purposes, including aggregation.** Deep NNs provide fewer final features, which favors aggregation
- **Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors**

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!
- The RL game successes are spectacular, but they have benefited from **perfectly known and stable models and relatively small number of controls (per state)**
- On the positive side, massive computational power together with distributed computation are a source of hope
- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features** of the cost function of a policy
- Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!
- The RL game successes are spectacular, but they have benefited from perfectly known and stable models and relatively small number of controls (per state)
- On the positive side, massive computational power together with distributed computation are a source of hope
- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features of the cost function of a policy**
- Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!

- The RL game successes are spectacular, but they have benefited from perfectly known and stable models and relatively small number of controls (per state)
- On the positive side, massive computational power together with distributed computation are a source of hope
- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features of the cost function of a policy**
- Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation.
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors.

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!

- The RL game successes are spectacular, but they have benefited from **perfectly known and stable models** and relatively **small number of controls** (per state)
  - On the positive side, massive computational power together with distributed computation are a source of hope
  - There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: **Automatically extract features of the cost function of a policy**
- Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation.
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors.

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!
- The RL game successes are spectacular, but they have benefited from **perfectly known and stable models** and relatively **small number of controls** (per state)
- On the positive side, massive computational power together with distributed computation are a source of hope.
- There is an exciting journey ahead ...
Concluding Remarks

- NNs resolve a major difficulty of approximate PI: Automatically extract features of the cost function of a policy.
- Good features, once extracted can be used for other purposes, including aggregation. Deep NNs provide fewer final features, which favors aggregation.
- Aggregation benefits from the solidity of exact DP algorithms, but must deal with quantization errors.

Some words of caution

- There are challenging issues for self-learning/approximate PI implementation.
  - Approximation architecture design using features
  - Exploration/sample design
  - Training algorithms
  - Oscillations
  - Recognizing success or failure!
- The RL game successes are spectacular, but they have benefited from perfectly known and stable models and relatively small number of controls (per state).
- On the positive side, massive computational power together with distributed computation are a source of hope.
- There is an exciting journey ahead ...
Thank you!