Deep reinforcement learning

Hado van Hasselt
Big picture
History

Big picture

- Industrial revolution (1750 - 1850) and Machine Age (1870 - 1940)
  - Implement repetitive manual solutions with machines
History

Big picture

- Industrial revolution (1750 - 1850) and Machine Age (1870 - 1940)
  - Implement *repetitive manual solutions* with machines
- Digital revolution (1960 - now) and Information Age
  - Implement *repetitive mental solutions* with machines
History
Big picture

- Industrial revolution (1750 - 1850) and Machine Age (1870 - 1940)
  - Implement repetitive manual solutions with machines
- Digital revolution (1960 - now) and Information Age
  - Implement repetitive mental solutions with machines

In both cases: have to come up with solution first
History

Big picture

- Industrial revolution (1750 - 1850) and Machine Age (1870 - 1940)
  - Implement repetitive manual solutions with machines
- Digital revolution (1960 - now) and Information Age
  - Implement repetitive mental solutions with machines

In both cases: have to come up with solution first

- AI revolution
  - We only specify the goal, solutions are found autonomously
Artificial intelligence

Big picture

- Symbolic GOFAI
  - Conclusions are derived, but rules are programmed and static
  - Hand-picked knowledge formalism & level of abstraction
  - Hard to deal with messy data and uncertainty
Artificial intelligence

Big picture

- **Symbolic GOFAI**
  - Conclusions are derived, but rules are programmed and static
  - Hand-picked knowledge formalism & level of abstraction
  - Hard to deal with messy data and uncertainty

- **Classic statistics**
  - Analyse data
  - We make decisions based on analysis
Artificial intelligence

Big picture

- Symbolic GOFAI
  - Conclusions are derived, but rules are programmed and static
  - Hand-picked knowledge formalism & level of abstraction
  - Hard to deal with messy data and uncertainty

- Classic statistics
  - Analyse data
  - We make decisions based on analysis

- True AI should learn to make decisions autonomously
Reinforcement learning
Reinforcement learning

A framework for making decisions

- RL provides a general-purpose framework for making decisions

Image credits - AIGA Collection, Martin Vanco
Reinforcement learning
A framework for making decisions

- RL provides a general-purpose framework for making decisions
  - RL is about learning to act
  - Each action can alter the state of the world, and can result in reward
  - Goal: optimize future rewards (which may be internal to the agent)
Reinforcement learning

Examples

- Examples of reinforcement learning domains:
  - Video games (including Atari)
  - Board games (including the game of Go)
  - Robotics
  - Recommender systems
  - ...

Deep reinforcement learning — Hado van Hasselt
Reinforcement learning

Examples

- Examples of reinforcement learning domains:
  - Video games (including Atari)
  - Board games (including the game of Go)
  - Robotics
  - Recommender systems
  - …

- Essentially, problems that involves making decisions and/or making predictions about the future
The goal is to learn a policy of behaviour. (At least) three possibilities:

- Learn policy directly
- Learn values of each action - infer policy by inspection
- Learn a model - infer policy by planning
Approaches to reinforcement learning

- The goal is to learn a policy of behaviour
- (At least) three possibilities:
  - Learn policy directly
  - Learn values of each action - infer policy by inspection
  - Learn a model - infer policy by planning
- Agents therefore typically have at least one of these components:
  - **Policy** - maps current state to action
  - **Value function** - prediction of value for each state and action
  - **Model** - agent’s representation of the environment.
Reinforcement learning

Components

- Policy: \( \pi(s) = a \)
- Value: \( Q(s, a) \approx \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \ldots | S_t = s, A_t = a] \)
- Model: \( m(s, a) \approx \mathbb{E}[S_{t+1} | S_t = s, A_t = a] \)
Reinforcement learning

Components

- **Policy**: \( \pi(s) = a \)
- **Value**: \( Q(s, a) \approx \mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \ldots | S_t = s, A_t = a] \)
- **Model**: \( m(s, a) \approx \mathbb{E}[S_{t+1} | S_t = s, A_t = a] \)

- All components are functions
- We need to represent and learn these functions
Deep reinforcement learning

Use **deep learning** to learn
**policies**, **values**, and/or **models**
to use in a reinforcement learning domain
Deep reinforcement learning

- Reinforcement learning provides: a framework for making decisions
- Deep learning provides: tools to learn components
Deep reinforcement learning

- Reinforcement learning provides: a framework for making decisions
- Deep learning provides: tools to learn components

\[ \text{AI} = \text{RL} + \text{DL} \, ? \]

- Concretely, we implement RL components with deep neural networks
Deep Q Networks
Q-learning
An algorithm to learn values

- The optimal value function fulfills:

\[ Q^*(s, a) = \mathbb{E} \left[ R_{t+1} + \max_b Q^*(S_{t+1}, b) \mid S_t = s, A_t = a \right] \] (Bellman, 1957)
Q-learning
An algorithm to learn values

- The optimal value function fulfills:

\[ Q^*(s, a) = \mathbb{E} \left[ R_{t+1} + \max_b Q^*(S_{t+1}, b) \mid S_t = s, A_t = a \right] \]

(Bellman, 1957)

- We can turn this into a TD algorithm:

\[ Q_{t+1}(S_t, A_t) = Q_t(S_t, A_t) + \alpha \left( R_{t+1} + \gamma \max_a Q_t(S_{t+1}, a) - Q_t(S_t, A_t) \right) \]

(Watkins 1989)
Q-learning
An algorithm to learn values

- By learning off-policy about the policy that is currently greedy, Q-learning can approximate the optimal value function $Q^*$.
- With $Q^*$, we have an optimal policy:

$$\pi^*(s) = \text{argmax } Q^*(s, \cdot)$$
DQN

(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

- Learns to play video games simply by playing
- Can learn Q function by Q-learning

\[ \Delta \mathbf{w} = \alpha \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \mathbf{w}) - Q(S_t, A_t; \mathbf{w}) \right) \nabla_{\mathbf{w}} Q(S_t, A_t; \mathbf{w}) \]
DQN

- Aside: we can phrase the update as a **loss**

$$\text{minimize } \frac{1}{2} \| y - q(s, a; \theta) \|_2$$  where, e.g.,  
$$y = R_{t+1} + \gamma \max_a q(S_{t+1}, a; \theta)$$

- Typically, we consider the target \( y \) as constant, and ignore the dependence on the parameters
  - E.g., in TensorFlow you might use placeholders, or a `stop_gradient`
  - Interpretation: \( y \) is an estimate for (off-policy) expected return \( \mathbb{E}[G_t | \pi, \sigma] \)
  - Then just update towards this estimate
Deep reinforcement learning — Hado van Hasselt

- Learns to play video games simply by playing
- Can learn Q function by Q-learning

\[ \Delta w = \alpha \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta) - Q(S_t, A_t; \theta) \right) \nabla \theta Q(S_t, A_t; \theta) \]

- Core components of DQN include:
  - Target networks (Mnih et al. 2015)
    \[ \Delta w = \alpha \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta^-) - Q(S_t, A_t; \theta) \right) \nabla \theta Q(S_t, A_t; \theta) \]
  - Experience replay (Lin 1992): replay previous tuples (s, a, r, s')
Target Network Intuition

(Slide credit: Vlad Mnih)

- Changing the value of one action will change the value of other actions and similar states.
- The network can end up chasing its own tail because of bootstrapping.
- Somewhat surprising fact - bigger networks are less prone to this because they alias less.

\[ L_i(\theta_i) = \mathbb{E}_{s,a,s',r \sim D} \left( r + \gamma \max_{a'} Q(s', a' ; \theta^-) - Q(s, a; \theta_i) \right)^2 \]
Experience replay

- Idea: store experiences, learn from them more than once
  - In Nature DQN, sample uniformly, see each sample 4 times on average
- Benefits:
  - More data efficient
  - Learning resembles supervised learning more (deep learning likes this)
DQN
(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

- Many later improvements to DQN
  - Double Q-learning (van Hasselt 2010, van Hasselt et al. 2015)
  - Prioritized replay (Schaul et al. 2016)
  - Dueling networks (Wang et al. 2016)
  - Asynchronous learning (Mnih et al. 2016)
  - Adaptive normalization of values (van Hasselt et al. 2016)
  - ... many more ...

DQN (Mnih, Kavukcuoglu, Silver, et al., Nature 2015)
Experience replay

- We can view the replay as an empirical (non-parametric) model.
- Can we query this model more cleverly?
- Yes:
  - Sample non-uniformly: prioritized replay really helps! (Schaul et al. 2016)
Prioritized Experience Replay

(Slide credit: Vlad Mnih)

- Replaying all transitions with equal probability is highly suboptimal.
- Replay transitions in proportion to absolute Bellman error:
  \[ r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \]
- Leads to much faster learning.

<table>
<thead>
<tr>
<th>Median</th>
<th>DQN baseline</th>
<th>DQN rank-based</th>
<th>Double DQN (tuned) baseline</th>
<th>Double DQN (tuned) rank-based</th>
<th>Double DQN (tuned) proportional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>48%</td>
<td>106%</td>
<td>111%</td>
<td>113%</td>
<td>128%</td>
</tr>
<tr>
<td></td>
<td>122%</td>
<td>355%</td>
<td>418%</td>
<td>454%</td>
<td>551%</td>
</tr>
<tr>
<td>&gt; baseline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt; human</td>
<td>15</td>
<td>25</td>
<td>30</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td># games</td>
<td>49</td>
<td>49</td>
<td>57</td>
<td>57</td>
<td>57</td>
</tr>
</tbody>
</table>

"Prioritized Experience Replay", Schaul et al. (2016)
Double DQN
(van Hasselt, Guez, Silver, AAAI 2015)

DQN:

\[ \Delta w = \alpha \left( r + \max_{a'} Q(s', a'; w^-) - Q(s, a; w) \right) \nabla_w Q(s, a; w) \]
Double DQN
(van Hasselt, Guez, Silver, AAAI 2015)

DQN:

$$\Delta w = \alpha \left( r + \max_{a'} Q(s', a'; w^-) - Q(s, a; w) \right) \nabla_w Q(s, a; w) = \Delta w = \alpha \left( r + Q(s', \arg\max_{a'} Q(s', a'; w^-); w^-) - Q(s, a; w) \right) \nabla_w Q(s, a; w)$$
Double DQN
(van Hasselt, Guez, Silver, AAAI 2015)

DQN:
\[ \Delta w = \alpha \left( r + \max_{a'} Q(s', a'; w^-) - Q(s, a; w) \right) \nabla_w Q(s, a; w) \]

\[
= \Delta w = \alpha \left( r + Q(s', \arg\max_{a'} Q(s', a'; w^-); w^-) - Q(s, a; w) \right) \nabla_w Q(s, a; w) \]

Double DQN:
\[ \Delta w = \alpha (r + Q(s', \arg\max_{a'} Q(s', a'; w); w^-) - Q(s, a)) \nabla_w Q(s, a; w) \]

Idea: decorrelate selection and evaluation to mitigate overestimation
Double DQN
(van Hasselt, Guez, Silver, AAAI 2015)
Double DQN
(van Hasselt, Guez, Silver, AAAI 2015)
Double DQN
(van Hasselt, Guez, Silver, AAAI 2015)
Double DQN
(van Hasselt, Guez, Silver, AAAI 2015)
The take-home message is:
- Be aware of the properties of your learning algorithms
- Track and analyse statistics
- If you understand what the problem is, a solution is sometimes very simple
Insights

- The take-home message is:
  - Be aware of the properties of your learning algorithms
  - Track and analyse statistics
  - If you understand what the problem is, a solution is sometimes very simple

- **RL-aware DL and DL-aware RL**
  - Target networks, experience replay: DL-aware RL
  - Next up, dueling networks: RL-aware DL
Dueling DQN

(Slide credit: Vlad Mnih)

- Value-Advantage decomposition of $Q$:
  \[ Q^\pi(s, a) = V^\pi(s) + A^\pi(s, a) \]

- Dueling DQN (Wang et al., 2015):
  \[ Q(s, a) = V(s) + A(s, a) - \frac{1}{|A|} \sum_{a=1}^{|A|} A(s, a) \]

### Atari Results

<table>
<thead>
<tr>
<th></th>
<th>30 no-ops</th>
<th>Human Starts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>Prior. Duel Clip</td>
<td>591.9%</td>
<td>172.1%</td>
</tr>
<tr>
<td>Prior. Single</td>
<td>434.6%</td>
<td>123.7%</td>
</tr>
<tr>
<td>Duel Clip</td>
<td>373.1%</td>
<td>151.5%</td>
</tr>
<tr>
<td>Single Clip</td>
<td>341.2%</td>
<td>132.6%</td>
</tr>
<tr>
<td>Single</td>
<td>307.3%</td>
<td>117.8%</td>
</tr>
<tr>
<td>Nature DQN</td>
<td>227.9%</td>
<td>79.1%</td>
</tr>
</tbody>
</table>

"Dueling Network Architectures for Deep Reinforcement Learning", Wang et al. (2016)
A task is defined by its rewards

- Atari: change in score
- Go: win (+1) or lose (-1)
Rewards
Defining optimality

- A task is defined by its rewards
  - Atari: change in score
  - Go: win (+1) or lose (-1)
- In DQN, all rewards were clipped to [-1, 1]
  - This helps learning
  - But it also changes the objective
Adaptive normalization
(van Hasselt et al. NIPS 2016)

- Optimization algorithms like normalized updates
- Clipping rewards is one solution, but we can do better
- We tried **adaptive target normalization** (algorithm is called Pop-Art)
Adaptive normalization
(van Hasselt et al. NIPS 2016)
Unclipping rewards

Videos at: hadovanhasselt.com/2016/08/17/atari-videos/
Policy gradients and actor-critic methods

Several slides adapted from Vlad Mnih
Policy Gradient

- We can often do better if the policy is differentiable.
  - Optimize the performance with gradient descent.
- The goal is to compute the gradient of the objective:
  \[ \nabla_\theta J(\theta) = \nabla_\theta \mathbb{E} [r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots] \]
- How can we compute this when rewards aren't differentiable?
- It turns out that there is a simple unbiased estimate of this gradient.
Contextual Bandit Policy Gradient

- Consider the simple one-step MDP (contextual bandit) setting.
- Start states are distributed according to $d$ and episodes are one step long.

\[
\nabla_\theta \mathbb{E}[R(S, A)] = \nabla_\theta \sum_s d(s) \sum_a \pi_\theta(a|s) R(s, a) \\
= \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(a|s) R(s, a) \\
= \sum_s d(s) \sum_a \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} R(s, a) \\
= \sum_s d(s) \sum_a \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) R(s, a) \\
= \mathbb{E}[\nabla_\theta \log \pi_\theta(A|S) R(S, A)]
\]
The gradient of the expected reward is given by:

$$\nabla_\theta \mathbb{E}[R(S, A)] = \mathbb{E}[\nabla_\theta \log \pi_\theta(A|S)R(S, A)]$$

We can approximate this with samples and update the policy using SGD:

$$\theta_{t+1} = \theta_t + \alpha R_{t+1} \nabla_\theta \log \pi_{\theta_t}(A_t|S_t)$$
Policy Gradient Theorem

● A more general result applies to full multi-step MDPs.

● For all differentiable policies:
\[ \nabla_\theta J(\theta) = \mathbb{E} \left[ \nabla_\theta \log \pi_\theta(a|s)Q^\pi(s, a) \right] \]
where expectation is over states and actions.


● There is an easy sample-based approximation (REINFORCE):
\[ \nabla_\theta \log \pi_\theta(a_t|s_t)G_t \]
where
\[ G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \]

“Simple statistical gradient-following algorithms for connectionist reinforcement learning”, Williams (1992)
Variance Reduction

- The REINFORCE gradient suffers from high variance.
- Subtracting a **baseline** keeps the gradient unbiased and reduces the variance:
  \[ \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left( G_t - b(s_t) \right) \]
- The state value function \( V(s) \) is a good choice for a baseline.
- Leads to a very intuitive form of update:
  \[ \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left( G_t - v(s_t) \right) \]
- → Increase probability when action was better than expected

“Simple statistical gradient-following algorithms for connectionist reinforcement learning”, Williams (1992)
How can policy-based methods be implemented efficiently with neural networks?

DQN uses replay, but standard PG methods are on-policy:
  - Require samples from the current policy.
  - Good off-policy PG methods have since been developed:
    - See ACER (Wang et al., 2016) and PGQL (O’Donoghue et al., 2016).
  - Idea: sample from replay, but adapt the updates so that expected gradient looks as if we use the current policy.
Asynchronous training of RL agents:

- Parallel actor-learners implemented using **CPU threads** and shared parameters.
- Online **Hogwild!**-style asynchronous updates (Recht et al., 2011, Lian et al., 2015).
- No replay? Parallel actor-learners have a similar stabilizing effect.
- Choice of RL algorithm: on-policy or off-policy, value-based or policy-based.

Asynchronous 1-step Q-Learning

- Parallel actor-learners compute online 1-step update

\[ y \leftarrow r + \gamma \max_{a'} Q(s', a'; \theta^{-}) \]

\[ \Delta \theta \leftarrow \Delta \theta + \frac{\partial (y - Q(s, a; \theta))^2}{\partial \theta} \]

- Gradients accumulated over minibatch before update
Asynchronous N-step Q-Learning

- Q-learning with a uniform mixture of backups of length 1 through N.

$$y \leftarrow \sum_{k=0}^{N-1} \gamma^k r_{t+k} + \gamma^N \max_{a'} Q(s_{t+N}, a'; \theta^-)$$

$$\Delta \theta \leftarrow \Delta \theta + \frac{\partial (y - Q(s_t, a_t; \theta))^2}{\partial \theta}$$

- Variation of “Incremental multi-step Q-learning” (Peng & Williams, 1995).
Async Advantage Actor-Critic (A3C)

- The agent learns a **policy** and a state **value function**
- Uses bootstrapped n-step returns to reduce variance
- The policy gradient multiplied by an estimate of the advantage.
  - Similar to Generalized Advantage Estimation (Schulman et al, 2015).

\[
\nabla_\theta \log \pi(a_t | s_t, \theta) \left( \sum_{k=0}^{N} \gamma^k r_{t+k} + \gamma^{N+1} V(s_{t+N+1}) - V(s_t) \right)
\]

- Train value with n-step TD learning
- You can think of this as minimizing:

\[
\left( \sum_{k=0}^{N} \gamma^k r_{t+k} + \gamma^{N+1} V(s_{s_{t+N+1}}; \theta^-) - V(s_t; \theta) \right)^2
\]
AsyncRL - Learning Speed

- Asynchronous methods trained on 16 CPU cores compared to DQN (blue) trained on a K40 GPU.
- n-step methods can be much faster than single step methods.
- Async advantage actor-critic tends to dominate the value-based methods.

AsyncRL - Scalability

- Average speedup from using K threads to reach a reference score averaged over 7 Atari games.
- **Super-linear** speed-up for 1-step methods.
Data Efficiency of 1-Step Q-learning

- Better **data efficiency** from more threads + speedup from parallel training
  - 1 thread (blue) 16 threads (yellow)
Data Efficiency of A3C

- No data-efficiency gains. Sub-linear speedup from parallel training.
  - 1 thread (blue) 16 threads (yellow)
# A3C - ATARI Results

<table>
<thead>
<tr>
<th>Method</th>
<th>Training Time</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>8 days on GPU</td>
<td>121.9%</td>
<td>47.5%</td>
</tr>
<tr>
<td>Gorilla</td>
<td>4 days, 100 machines</td>
<td>215.2%</td>
<td>71.3%</td>
</tr>
<tr>
<td>D-DQN</td>
<td>8 days on GPU</td>
<td>332.9%</td>
<td>110.9%</td>
</tr>
<tr>
<td>Dueling D-DQN</td>
<td>8 days on GPU</td>
<td>343.8%</td>
<td>117.1%</td>
</tr>
<tr>
<td>Prioritized DQN</td>
<td>8 days on GPU</td>
<td>463.6%</td>
<td>127.6%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>1 day on CPU</td>
<td>344.1%</td>
<td>68.2%</td>
</tr>
<tr>
<td>A3C, FF</td>
<td>4 days on CPU</td>
<td>496.8%</td>
<td>116.6%</td>
</tr>
<tr>
<td>A3C, LSTM</td>
<td>4 days on CPU</td>
<td>623.0%</td>
<td>112.6%</td>
</tr>
</tbody>
</table>

A3C - Procedural Maze Navigation in 3D

A3C - Continuous Control

The best deep RL methods are still very data hungry. Especially with **sparse rewards**.

Obvious solution - Learn about the environment.

Augment an RL agent with **auxiliary prediction and control tasks** to improve data efficiency.

The UNREAL agent - UNsupervised REinforcement and Auxiliary Learning.

- "Reinforcement Learning with Unsupervised Auxiliary Tasks", (Jaderberg et al. 2017)
The UNREAL Architecture

● UNREAL augments an LSTM A3C agent with 3 auxiliary tasks.

● Can be used on top of DQN, DDPG, TRPO or other agents.
The UNREAL Architecture

- Base A3C LSTM agent learns from the environment’s scalar reward signal.
- UNREAL acts using the base A3C agent’s policy.
Unsupervised RL

- Augment A3C with many **auxiliary control tasks**.
- Pixel control - learn to maximally change parts of the screen.
- Feature control (not used by UNREAL) - learn to control the internal representations.
The UNREAL Architecture

Focusing on rewards:

- Rebalanced reward prediction.

- Shape the agent’s CNN by classifying whether a sequence of frames will lead to reward.

- No need to worry about off-policy learning.
The UNREAL Architecture

Focusing on rewards:

- Value function replay.
- Faster learning of the value function.
Average human-normalized performance on 13 3D environments from DeepMind Lab.

Tasks include random maze navigation and laser tag.

Roughly a 10x improvement in data efficiency over A3C.

60% improvement in final performance.
Baduk in numbers

- 3,000 Years Old
- 40M Players
- $10^{170}$ Positions
Why is Baduk hard for computers to play?

Game tree complexity $= b^d$

Brute force search intractable:

1. Search space is huge
2. “Impossible” for computers to evaluate who is winning
Exhaustive search
Reducing depth with value network
Reducing depth with value network
Value network

\[ v_\theta(s) \]

\( s \)

\( \theta \)
Convolutional neural network
Reducing breadth with policy network
Policy network

Move probabilities

Position

\[ p(\sigma | s) \]
Monte-Carlo rollouts
Neural network training pipeline

Human expert positions → Supervised Learning policy network → Reinforcement Learning policy network → Self-play data → Value network

- Classification
- Self Play
- Regression
Internal Testing

AlphaGo (May 2017) → Wins 3/3 Matches → Ke Jie (9p)
World number 1

AlphaGo (Mar 2016) → Wins 4/5 Matches → Lee Sedol (9p)
Top player of past decade

AlphaGo (Oct 2015) → Wins 5/5 Matches → Fan Hui (2p)
3-times reigning Euro Champion

Calibration

External Testing
Planning with learned models
Learning models

Motivation

- We discussed learning policies and values
- What about models?
Learning models

Motivation

- We discussed learning policies and values
- What about models?
- Models would allow us to plan
  - Planning is useful in combinatorial and compositional domains
  - Trade off local compute to trying to store everything
  - Would allow us to use great planning algorithms
Example
Random Mazes

not connected  connected
Example

Pool
Learning models
Complexities

- Learning models from raw inputs is hard
  - What should our model capture - pixels?
  - Objectives do not match: potentially focus on irrelevant details
Learning models

Complexities

● Learning models from raw inputs is hard
  ○ What should our model capture - pixels?
  ○ Objectives do not match: potentially focus on irrelevant details

● What to do with an imprecise model?
  ○ Many planning algorithms assume model is perfect
The Predictron
(Silver, van Hasselt, Hessel, Schaul, Guez, et al., 2016)

- Main idea: learn an **abstract model**
- The model should be **good for planning**
- But it does not have to match the real dynamics
  - See also “Value iteration networks” (Tamar et al., 2016)
The Predictron

(Silver, van Hasselt, Hessel, Schaul, Guez, et al., 2016)
The Predictron
Learning abstract models

- Idea: compute looks like planning, but we do not have a separate model-learning objective
- Instead, the goal is to optimize the outcome of planning with the learnt model
- Then, learn all components end-to-end
- A model is learnt, because by construction a model exists
- But model-semantics (e.g., what does each state mean?) is not prefixed
The Predictron
Learning abstract models
The Predictron

Trajectory prediction with the abstract model

- **Left:**
  Random maze + start position

- **Right:**
  Trajectory for some policy:
  this is the target

- **Middle:**
  Internal partial plans appear in the predictron representation

- Partial trajectories were **not** in the data

- Internal plans compose sequentially into full trajectories
THANK YOU