Neural Networks

Introduction to Reinforcement Learning

CSCI 4850/5850

Kinds of Learning

Supervised

Much of what we have studied so far!

Requires a comprehensive set of labeled training data to provide feedback to the network

- Features -> Value (Regression)
- Images -> Classes (Classification)
- Source Language -> Target Language (Sequence Learning)

Semi-supervised

Often used when most of the data is unlabeled

Inductive bias is tuned (similar to unsupervised methods below) to make useful assumptions about the unlabeled examples so they can be used in the training process

More like human/animal learning where we have a few examples and then can extrapolate and learn from additional unlabeled examples

Reinforcement

Tasks where feedback is minimal and/or sparse (in time/space)

- Sparse in space: Classifying digits, but only told "right/wrong"
- Sparse in time: Translating a sentence, but only being told "right/wrong"

Can be sparse in both!

Often involves exploration (the supervisor can't provide feedback on every step)

Often used for sequential task learning (problems with states and actions)

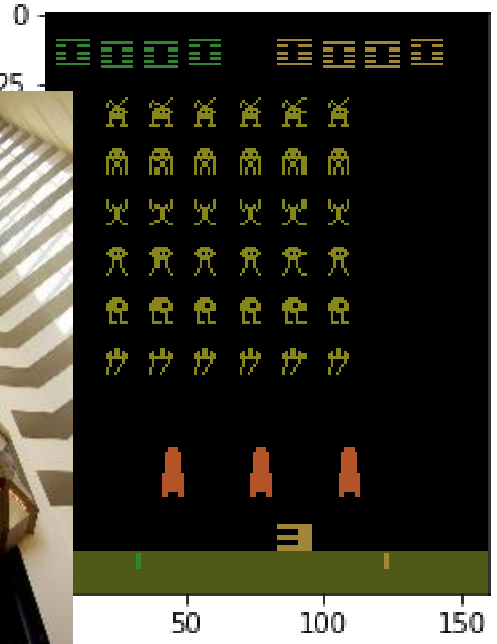
Unsupervised

No feedback – inductive bias tuned to organize or restructure the data in some manner

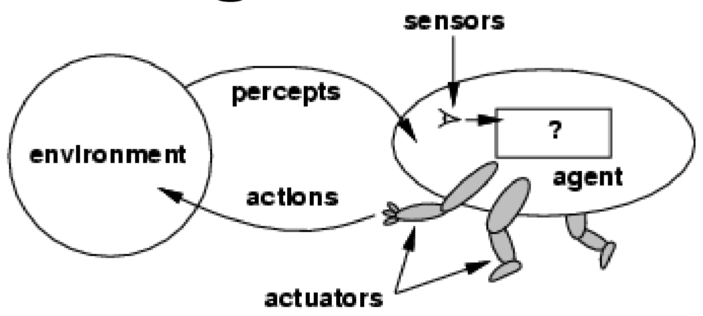
Typically used for pre- or post-processing of data for the above approaches (or for humans to use directly)

Reinforcement Learning

- Some tasks have naturally sparse feedback in time/space…
- Generally, we want to know how to act on-line (in the moment) and learn how to improve in a similar manner (although we may break the rules a little on this one with good reason…)



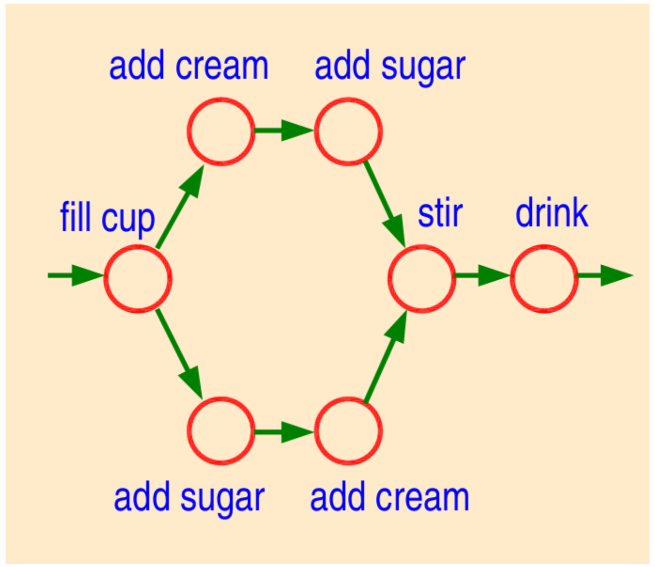
Foundational Framework: Artificial Intelligence Agents



- Two Parts
  - Agent
  - Environment
- Most problems in reinforcement learning focus on how the **agent** engages with an **environment**
- The agent's **percepts** over time divide the environment into a series of events (states)
- The agent can change the environment, and therefore affect state changes via **actuators** (actions)
  - Not acting (no-op) is an action as well…

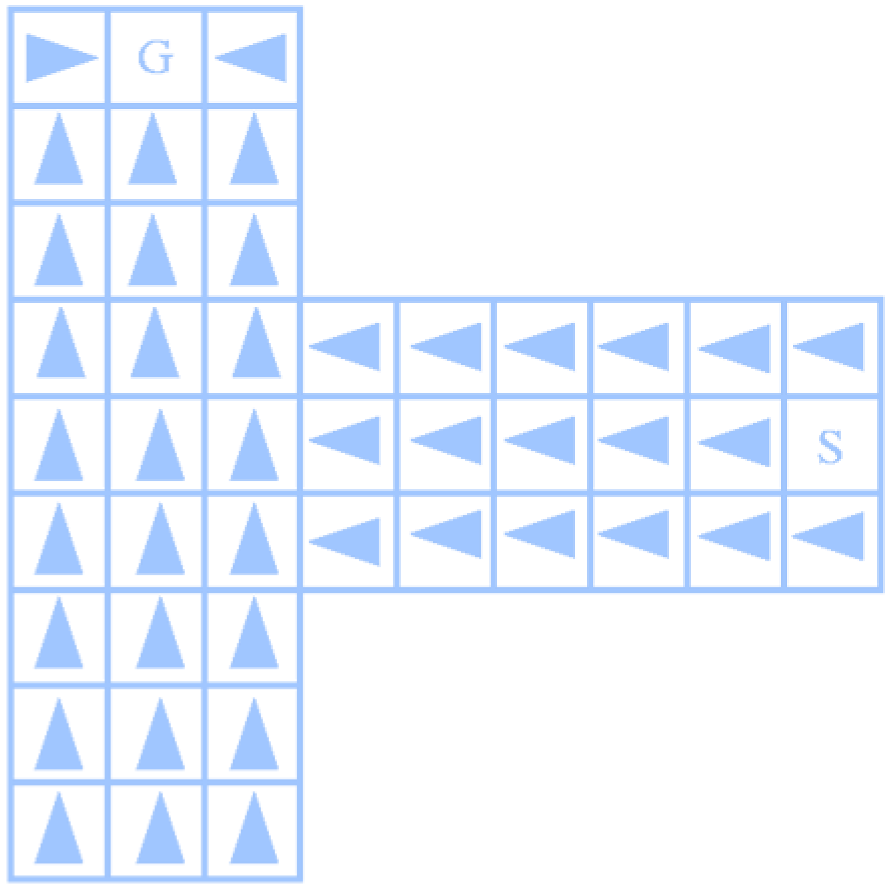
Foundational Framework: Dynamic Programming

- Reinforcement learning typically involves sparse (in time), general (in instruction) feedback
  - Usually, just a scalar signal (r) which is positive for "good work, reward", negative for "bad work, punishment", or zero for "nothing, neither reward nor punishment"
  - Typically, informative signals (positive or negative) are ***delayed*** in time (need to determine how to use these signals to update past strategies)
  - While r is provided to the agent, it is not necessarily viewed as a percept, similar to how targets are not inputs in a feedforward network
- Key idea: we want the agent to "program itself" to solve the task, instead of "telling it what to do"
  - Reduce the amount of manually-labeled training data the experimenter needs to provide…
- Rather than providing detailed instructions, can the agent learn the sequence of steps on it's own using exploration (trial-and-error)
- Most learning involves balancing:
  - exploration (finding new knowledge) and
  - exploitation (using currently learned knowledge)



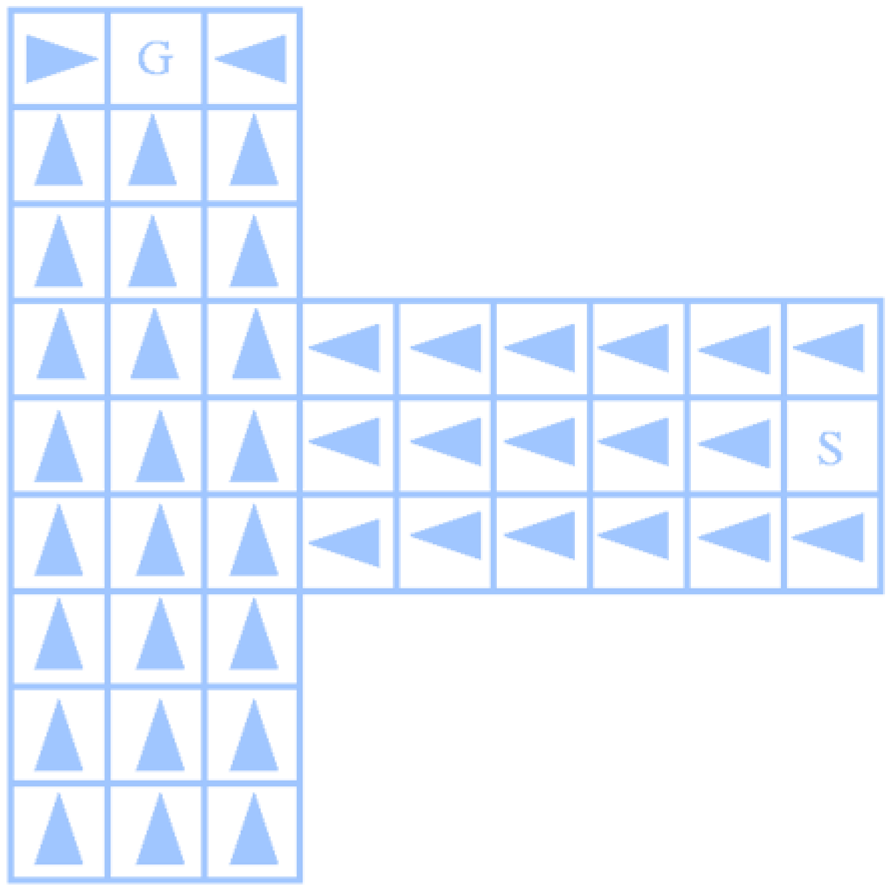
Some Notation and Definitions

- State: s
- Action: a
- Reward: r(s)
- Value: V(s)
- Policy: π(s) = a
- Q-Value: Q(s, a)



Policy Iteration

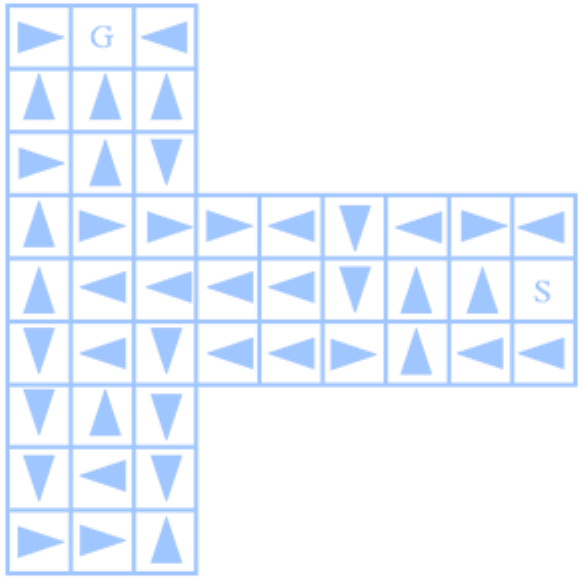
- Initialize π randomly
- Evaluate the value (V) of each state given the current policy
- A form of *dynamic programming*

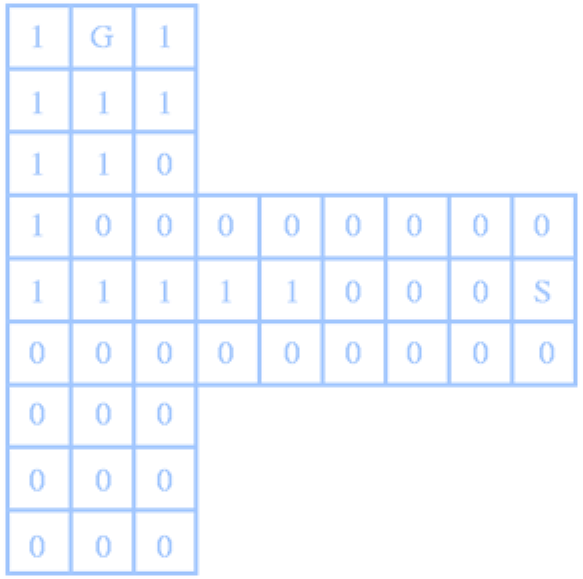


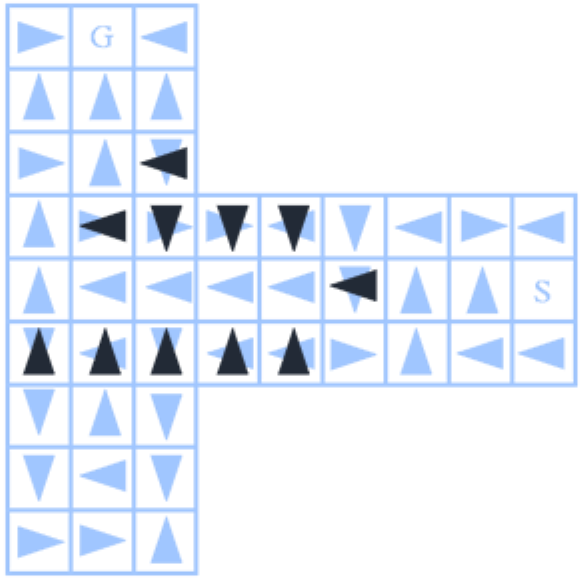
$$V(s) = r(s) + r(s+1) + \cdots + r(s_G)$$

Policy Iteration

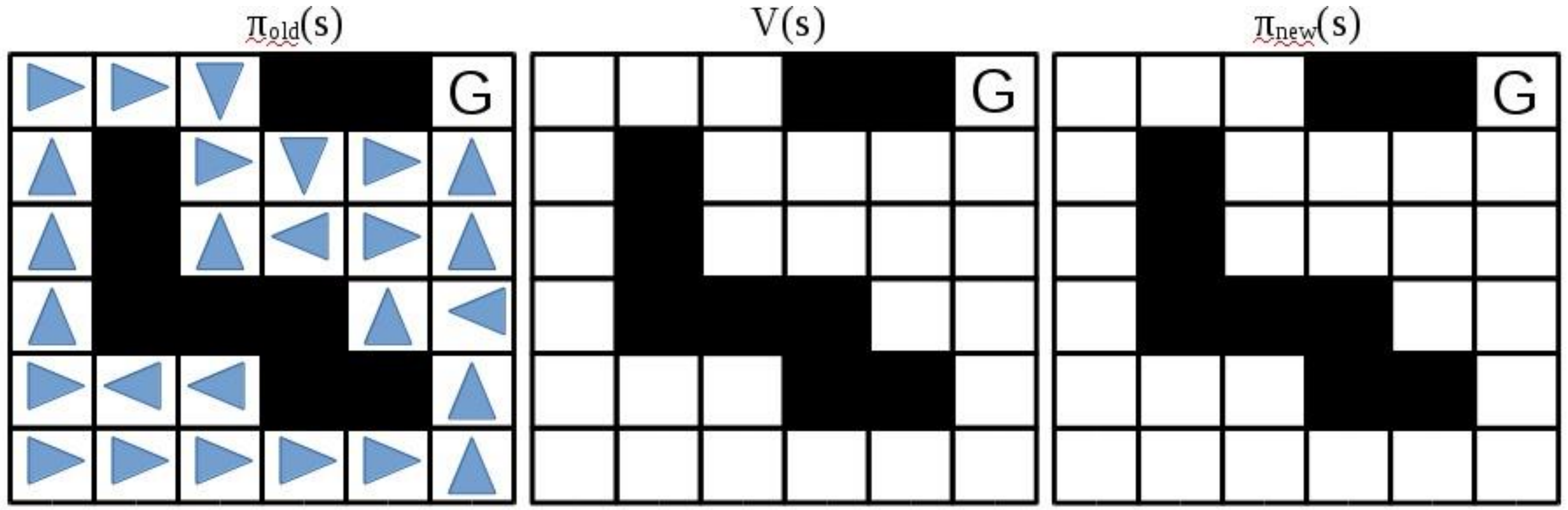
- Let the goal state (G) have a reward of 1
  - r(G) = 1, then for all S where S != G: r(S) = 0
- Now we update π to select actions that lead to states with the highest values.



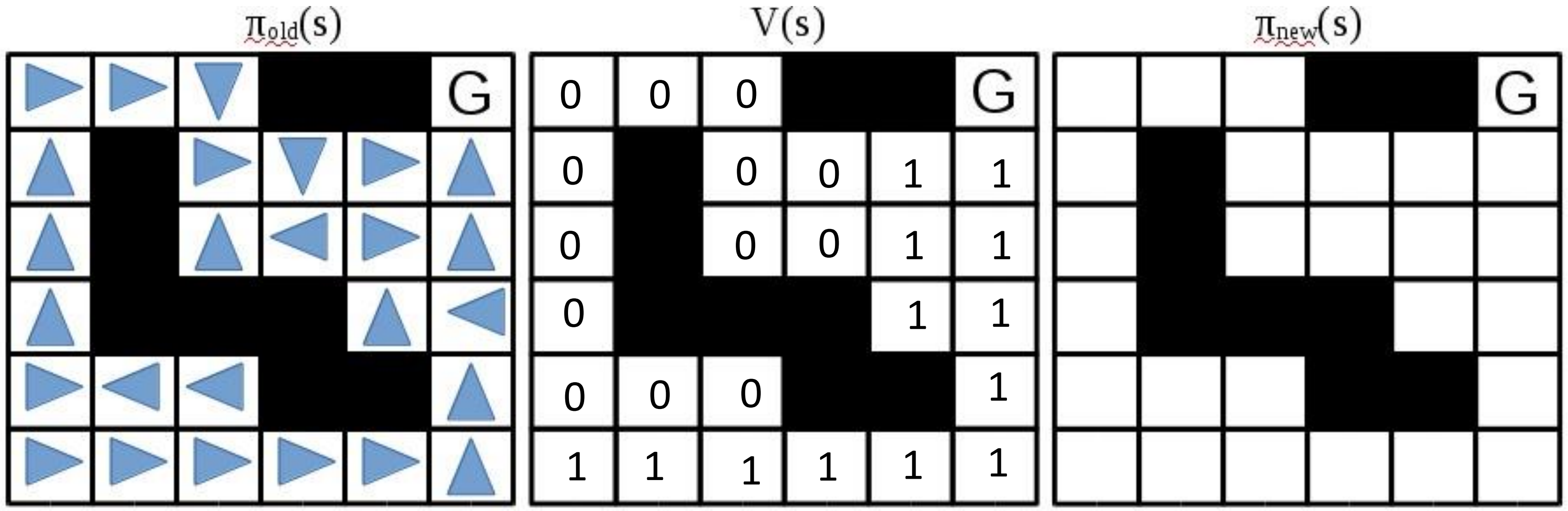




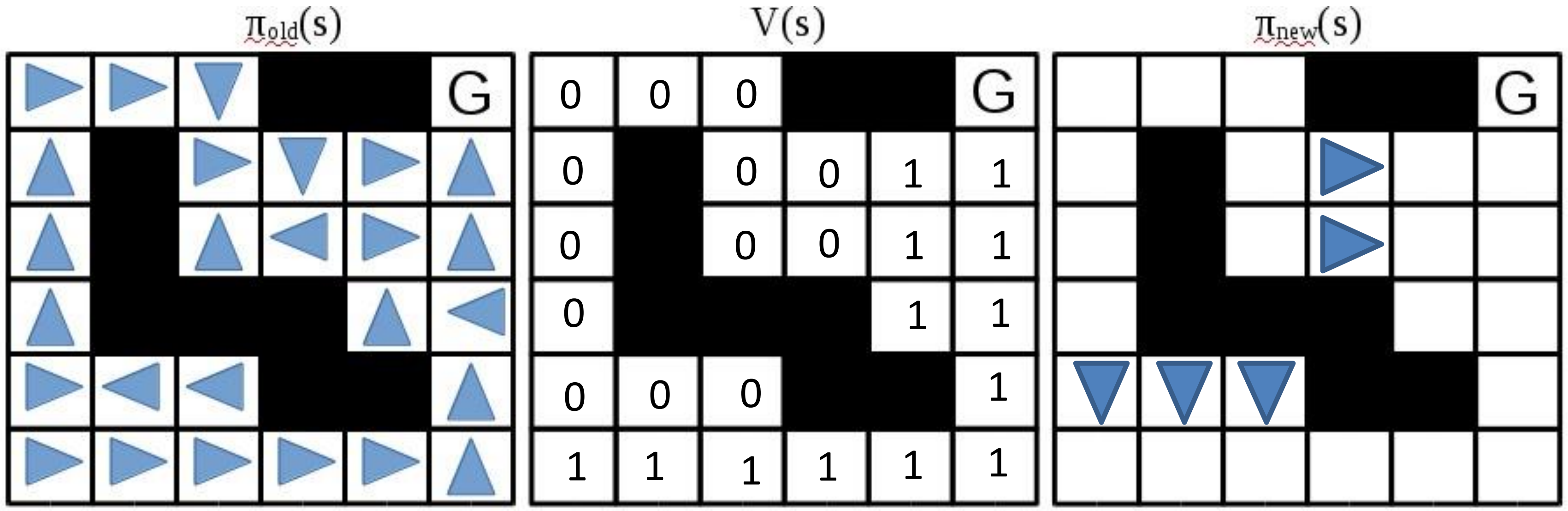
Example



Example



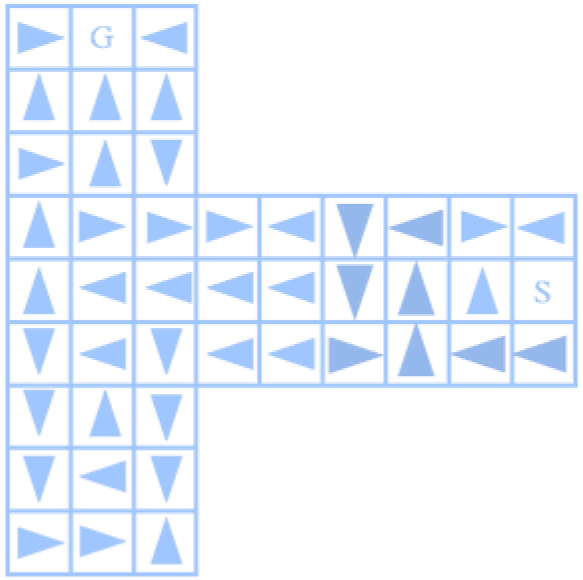
Example

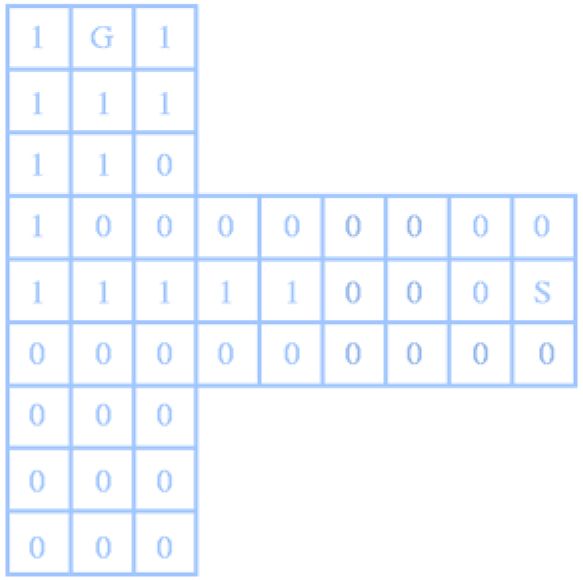


Only ***updated*** states shown…

Generalized Policy Iteration

- Rather than iterating over all states, just perform many training *episodes*.
- Episodes can be used to iteratively update both the policy (π) and the value function (V).





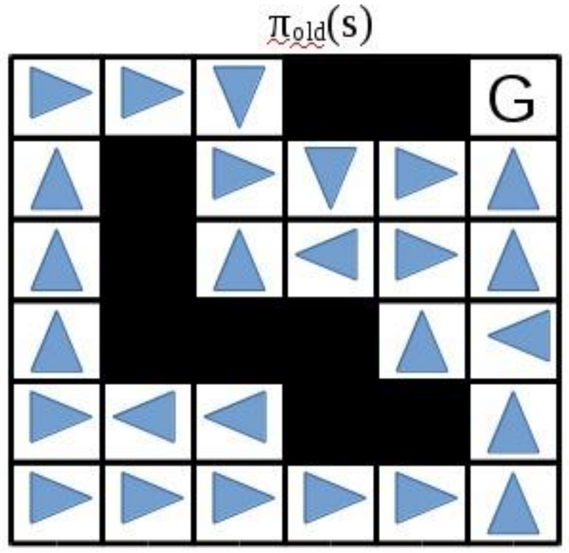
Drawbacks to Policy Iteration

- Need to *store/maintain* all state and reward information for an entire episode.
- We may not know the probability with which actions lead to some state – P(s+1 | s, a)
  - Reward information may vary on each episode.
  - Greedily updating π may not be the best strategy.
- Introducing γ: the ***discount factor***

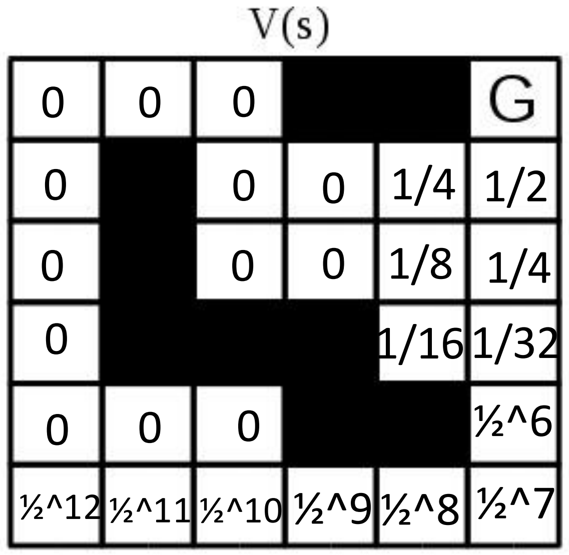
$$V(s) = \gamma^0 r(s) + \gamma^1 r(s+1) + \cdots + \gamma^n r(s+n)$$

Example

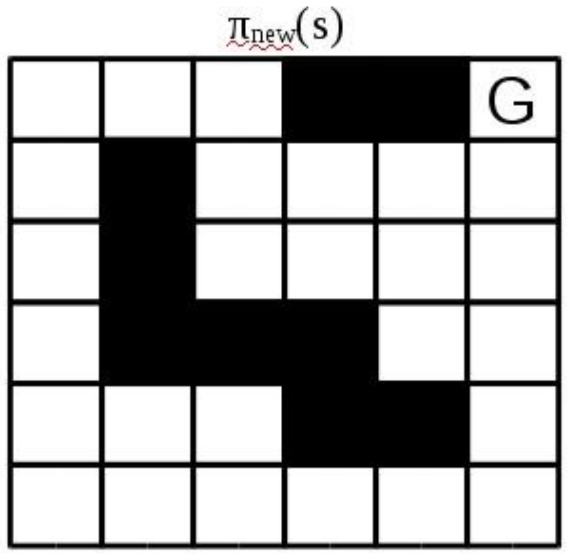
Assume: γ = 1/2



▶	▶	▼	■	■	G
▲	■	▶	▼	▶	▲
▲	■	▲	◀	▶	▲
▲	■	■	■	▲	◀
▶	◀	◀	■	■	▲
▶	▶	▶	▶	▶	▲



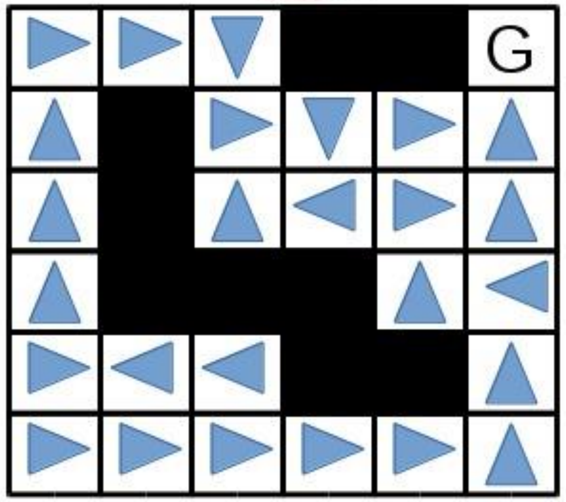
0	0	0	■	■	G
0	■	0	0	1/4	1/2
0	■	0	0	1/8	1/4
0	■	■	■	1/16	1/32
0	0	0	■	■	½^6
½^12	½^11	½^10	½^9	½^8	½^7



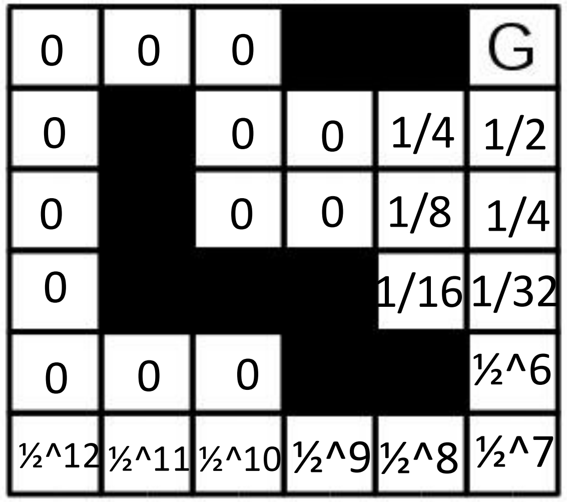
$$V(s) = \gamma^0 r(s) + \gamma^1 r(s+1) + \cdots + \gamma^n r(s+n)$$

Example

$\pi_{old}(s)$

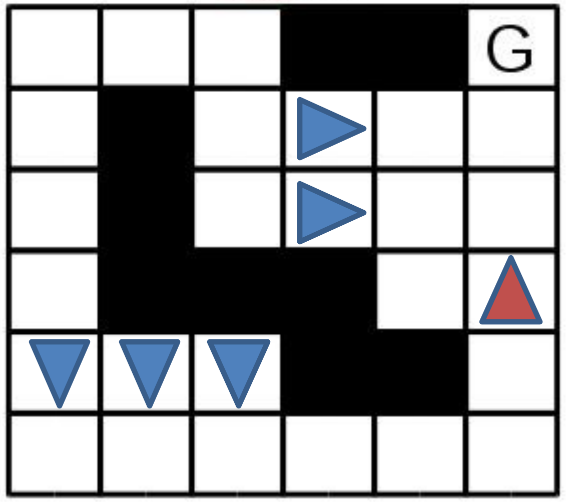


$V(s)$



0	0	0			G
0		0	0	1/4	1/2
0		0	0	1/8	1/4
0				1/16	1/32
0	0	0			½^6
½^12	½^11	½^10	½^9	½^8	½^7

$\pi_{new}(s)$



Only ***updated*** states shown…

Can determine paths
with fewest number of
actions (or time-steps)…

Temporal Difference Learning

- The value of a state can be computed incrementally, even updated incrementally.
- γ: Discount Factor

$$V(s) = \gamma^0 r(s) + \gamma^1 r(s+1) + \cdots + \gamma^n r(s+n)$$

$$V(s+1) = \gamma^0 r(s+1) + \gamma^1 r(s+2) + \cdots + \gamma^{n-1} r(s+n)$$

$$V(s) = r(s) + \gamma V(s+1)$$

$$\delta(s) = \big(r(s) + \gamma V(s+1)\big) - V(s)$$

TD Learning (On-line)

- Get reward value for current state
- Evaluate the value of taking all actions
- Let the next state be the one with highest value
- δ(s) = r(s) + γV(s+1) - V(s)
- Update V(s) = V(s) + α δ(s)
  - α is a small *learning rate* parameter (1 >= α >= 0)
- Move to next state – for all actions, a in A:

$$\pi(s) = argmax_a\left(V_a(s+1)\right)$$

Value Function Limitations

- The policy implementation for TD learning is not very practical:
  - Need to know which states will come next
  - Could be impossible due to:
    - nondeterminism (“expected” randomness)
    - noise (“unexpected” randomness)
- Key insight: the **value** can instead be tied to being in state, s, ***and*** taking a specific action, a.
- The agent has no need to understand how to predict which states will come next, it just needs to know which actions it can perform in the current state, s.

Q-Learning

- $Q(s,a) = r(s+1) + \gamma r(s+2) + \gamma^2 r(s+3) + \ldots$
- $Q(s,a) = r(s+1) + \gamma Q(s+1,a+1)$
- Observe: $r(s)$
- Choose action: max $Q(s,a^*)$
- $Q(s-1,a-1)$ += $\alpha(r(s) + \gamma \max Q(s,a^*))$
- Take action a^*
- Alternative actions may be chosen for *off-policy learning*

The update equation is often written with a forward-looking view (add one time step) but makes a little more intuitive sense when adjusted as shown above (reflective update)

Practical Considerations

- Learning the Q and/or V functions for a problem are the main point of reinforcement learning algorithms…
- Storing *unique entries* for each state, V(s), or state-action pair, Q(s,a), in a *look-up table* as we have done above isn't generally feasible...
- Neural networks can potentially learn any function, so we need to replace our look-up tables with NNs to work on general problems…

Action Selection

- Balance Exploration with Exploitation
- ε-greedy - With probability ε, take a random action, else max(A), max(V) or max(Q)
- ε-soft - use softmax : $a_i = \exp(A_i) / \Sigma_j \exp(A_j)$
- Convergence for ε-greedy/soft policies under certain conditions has been proven.
  - (Perkins and Precup, 2004)

Comparing Q-Learning to TD Learning

- The difference equations for Q and V are very similar, but Q has some practical advantages:
  - Using V(s) requires the ability to *predict* exactly (or with what probability) *which states* an action leads to
  - In the long term V(s) effectively becomes the *value of selecting the best action* (under the optimal policy), but says **nothing** about the other actions…
  - Q(s,a) calculates the *value* (under the optimal policy) for **each action**, and therefore can be used to select among them *without needing to predict* the outcome of action choices (in which state the agent will arrive)
  - Also, Q is stable under off-line learning conditions while TD technically is not (although it can easily be adapted to make it so, therefore this is not a *clear* advantage)

Learning Equation Comparisons

TD

$$V(s) = \gamma^0 r(s) + \gamma^1 r(s+1) + \ldots + \gamma^n r(s+n)$$

$$V(s+1) = \gamma^0 r(s+1) + \gamma^1 r(s+2) + \ldots + \gamma^{n-1} r(s+n)$$

$$V(s) = r(s) + \gamma V(s+1)$$

$$\delta(s) = (r(s) + \gamma V(s+1)) - V(s)$$

$$Q(s,a) = \gamma^0 r(s+1) + \gamma^1 r(s+2) + \ldots + \gamma^{n-1} r(s+n)$$

$$Q(s+1,\mathrm{a}+1) = \gamma^0 r(s+2) + \gamma^1 r(s+3) + \ldots + \gamma^{n-2} r(s+n)$$

$$Q(s,a) = r(s+1) + \gamma Q(s+1,\mathrm{a}+1)$$

$$\delta(s,a) = (r(s+1) + \gamma Q(s+1,\mathrm{a}+1)) - Q(s,a)$$

Q/Sarsa

Q-learning: always use amax$_a$(Q(s+1,a+1)) – off-policy learning
Sarsa: always use selected action Q(s+1,a+1) – on-policy learning

Sarsa is better suited to *on-line* learning in *stochastic* environments

Comparison of Tabular Results

Goal!

Periodic!

r(s)	0	0	0	0	0	0	0	0	1	0	0	0

Assume γ=1/2

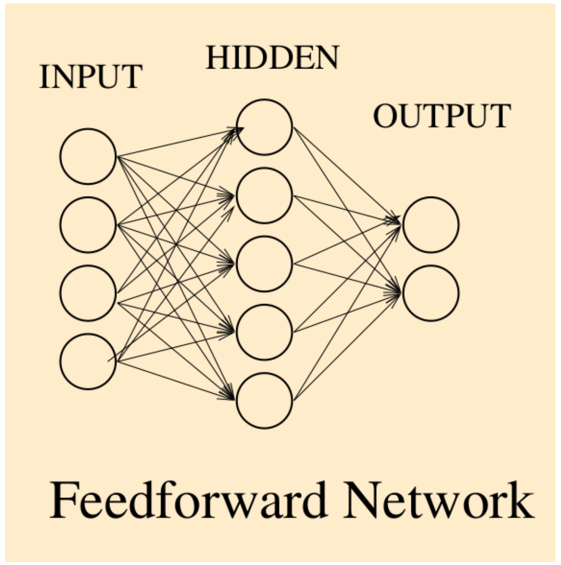
V(s)	½^4	½^5	½^6	½^5	½^4	½^3	¼	½	1	½	¼	½^3

$$V(s) = r(s) + \gamma V(s+1)$$

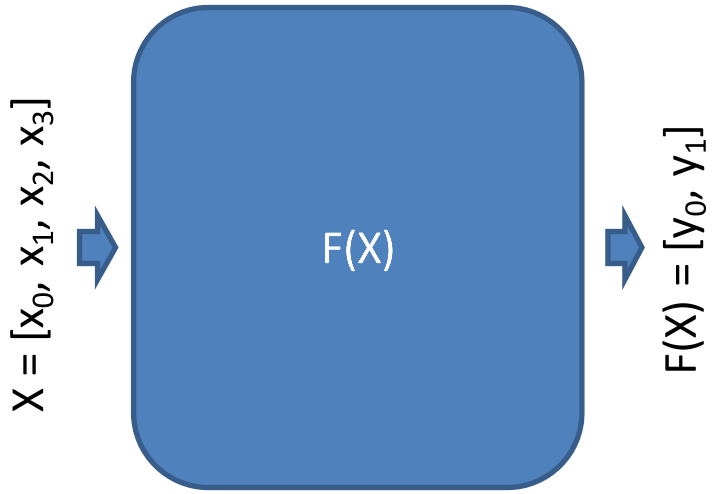
Q(s,a)	½^3	½^4	½^5	½^6	½^5	½^4	½^3	¼	X	1	½	¼	a_L
	½^5	½^6	½^5	½^4	½^3	¼	½	1	X	¼	½^3	½^4	a_R

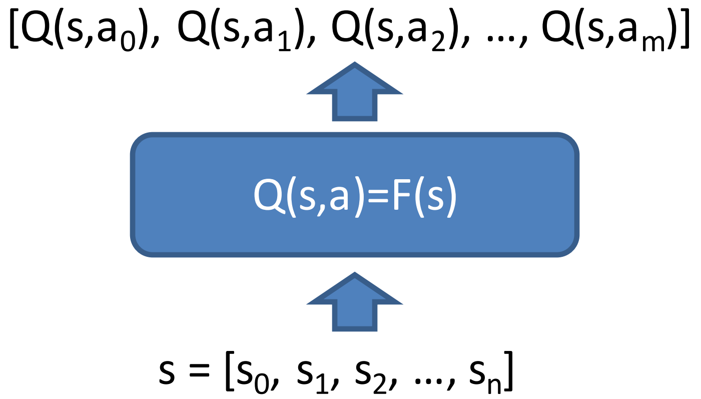
$$Q(s,a) = r(s+1) + \gamma Q(s+1, \mathrm{a}+1)$$

The Role of the Neural Net



- For problems with a *large number of states* (or continuous state spaces), ***using a look-up table is just not practical***
  - Many state spaces grow exponentially in size with increasing problem complexity, exacerbating this issue…
- Key Idea: Replace our *table* (which is really just a function) with a *function approximator* (neural network)




$[Q(s,a_0), Q(s,a_1), Q(s,a_2), \ldots, Q(s,a_m)]$
Q(s,a)=F(s)
$s = [s_0, s_1, s_2, \ldots, s_n]$

(Could be a one-hot encoding but usually we take features from the environment using sensors)

Q-function Regression

- Practical Considerations
  - Since Q-values are real numbers (not class labels) we need:
    - Activation function: linear (arbitrary real outputs)
    - Loss function: mse or logcosh (sometimes better)
      - Logcosh represents a more adaptive loss function (doesn't favor large errors)
    - Regression problem!
  - Architecture
    - Normally feed-forward neural networks are employed
    - Recurrent networks are an active area of research
- Markov Property
  - Features provided to the network need to provide all necessary information for making a good decision
  - Assume that the Q-function (alone) is enough to make the optimal decision (it's only given a feature vector!)
- Not a perfect regression problem…
  - The Q-function is used to update itself!
  - Moving targets, so it's already an ***implicitly*** recurrent architecture

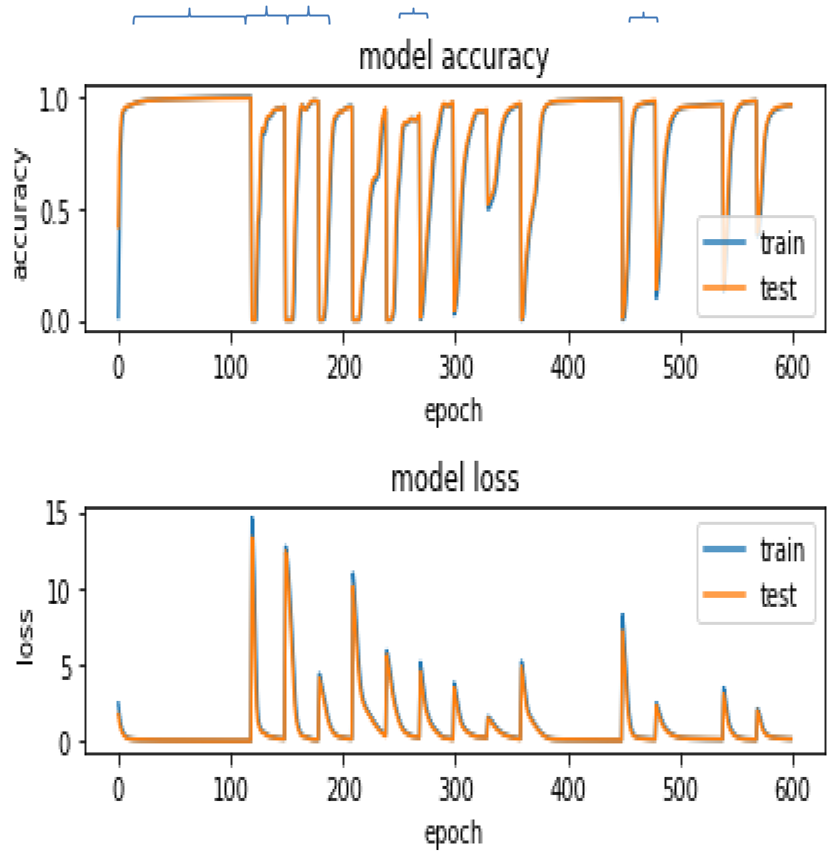
$$Q(s,a) \quad = \quad r(s+1) \quad + \quad \gamma Q(s+1, \mathrm{a}+1)$$

Challenges

- On-line learning
- Off-line learning
- Moving Targets
- Exploration/Exploitation
- Reward Scheduling
- Integration of Rewards

Catastrophic Interference
(this problem naturally occurs)

[0,1] [2,3]... [0,1] [0,1]



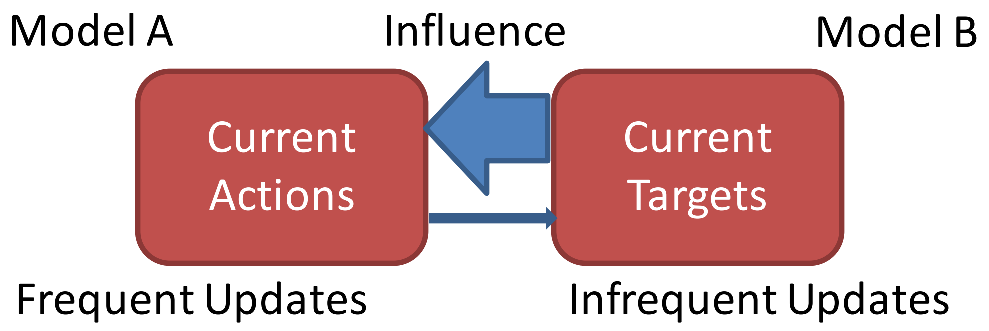
On-line Learning

- Learning on-line results in training the network using state-target patterns which have strong spatial and temporal correlations
  - Taking an action will change the state, but the next state still probably **looks** *a lot* like the last one (due to high state discretization)
  - Taking an action from a specific state still **usually leads** to the *same* next state even if the features of the next state are significantly different (due to high temporal discretization)
  - Both can still be exhibited by RL problems with low discretization as well (due to natural hierarchical structures)
- Training on temporally or spatially correlated data makes the network biased to learn the most recent information and forget less recent information…
- Why?
  - Tables allow for independent updates of Q-values (updating one Q-value has no direct effect on any others in the table)
  - Neural networks used *shared weights* for all Q-values (updating one Q-value can potentially have a direct effect on all Q-values!)

Off-line Learning

- Instead of learning on-line, we can instead retain a so-called **replay memory** of many state->action->next_state->reward transitions
- Train over **randomly sampled batches** integrating *both recent and older experiences* will enable the network to preserve prior information
- Cognitive theory suggests *sleep* may be when humans/animals perform such integration so it's not unfounded…

Moving Targets



- Regardless of the replay memory itself, once an update is made for a pattern, *others are impacted*
  - That is, after just a *single learning step*, the target used for *last step* will have ***changed***!
- This phenomenon is similar to how passing information **repeatedly through the same weights** in a recurrent neural network may *cause the gradient to explode or shrink away*.
- Therefore, we usually prepare an additional, matching network called the **target network**
- We use the target network to generate the target Q values during our use of the replay memory experiences and then update the weights in the target model after several learning cycles (this keeps the targets stable during the learning process)
  - Updating the target network (with the new learned weights) too frequently will provide no benefit
  - Updating the target network too infrequently will slow learning convergence

Exploration/Exploitation

- Early in learning, we need to encourage exploration early in a task to allow the network to find useful solutions
- Later in learning, too much exploration might degrade performance if a good policy has been discovered
- Typically, we use an ε-soft approach
  - Make ε high early in learning so that random, exploratory actions are common
  - Anneal to low ε later in learning so that random, exploratory actions are rare
  - Usually, this is then held at small value for the rest of the training regime…

Reward Scheduling

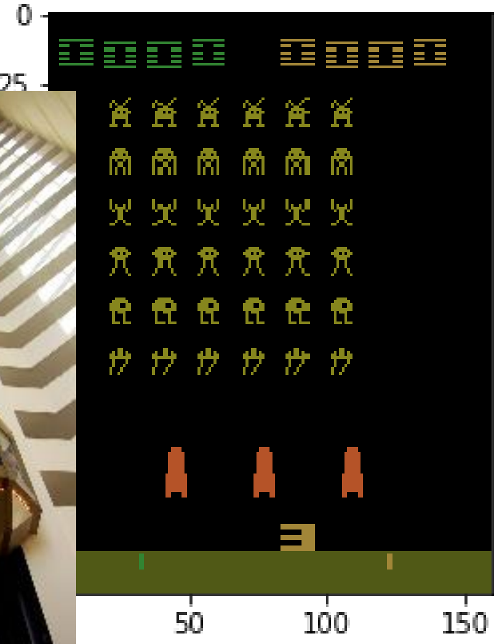
- Differences in reward schedule can drastically impact performance
  - Maze tasks:
    - 1 for goal, 0 for others: exponential time (#states)
    - 0 for goal, -1 for others: polynomial time (#states)
  - Why?
    - The network is typically initialized with small random weights, so initial Q-values are usually close to *zero*
    - Networks will spend a lot of time not updating Q values during the exploration phase because targets and outputs will then *both* be **zero**
    - Using a negative (punishing) reward means there is more direct feedback during the exploration phase that can be remembered
    - Even a small decrease in Q for a previously explored state-action pair may cause an alternative to be chosen on the next episode…
  - Performance tasks:
    - 1 for 100 time-steps performance, 0 for others: exponential time (#states)
    - 1 for each time-step during performance, -100 for failure: polynomial time (# states)
  - Why?
    - Again, it's about building up memory of past performance in the Q-values
    - Most tasks can be converted into a reward schedule that encourages memory-informed exploration over uninformed exploration (just epsilon)

Integration of Rewards

- Choosing a value of γ for your updates is not arbitrary, but also not too challenging…
  - γ is usually set between 0 and 1
  - $\gamma = 1$ means that rewards matter no matter when you get them…
    - for many tasks this is the wrong thing to do
    - Encourages any strategies that accumulate lots of reward regardless of how many steps/actions it takes
    - Very long-term thinking oriented (but doesn't consider the cost of long-term execution…)
  - $\gamma = 0$ means that rewards **only matter now**
    - Greedy approach will take a candy bar now even if promised 100 tomorrow (just wait a day)
    - Again, not usually what is desired, but can also lead to fast "solutions" to many problems
  - Typically, $\gamma = 0.95$ is a good rule of thumb
    - might need to be a little higher for your task if it is very long (0.99)
    - Encourages long-term accumulation of rewards, but still favors shorter solutions to long ones…

Reinforcement Learning

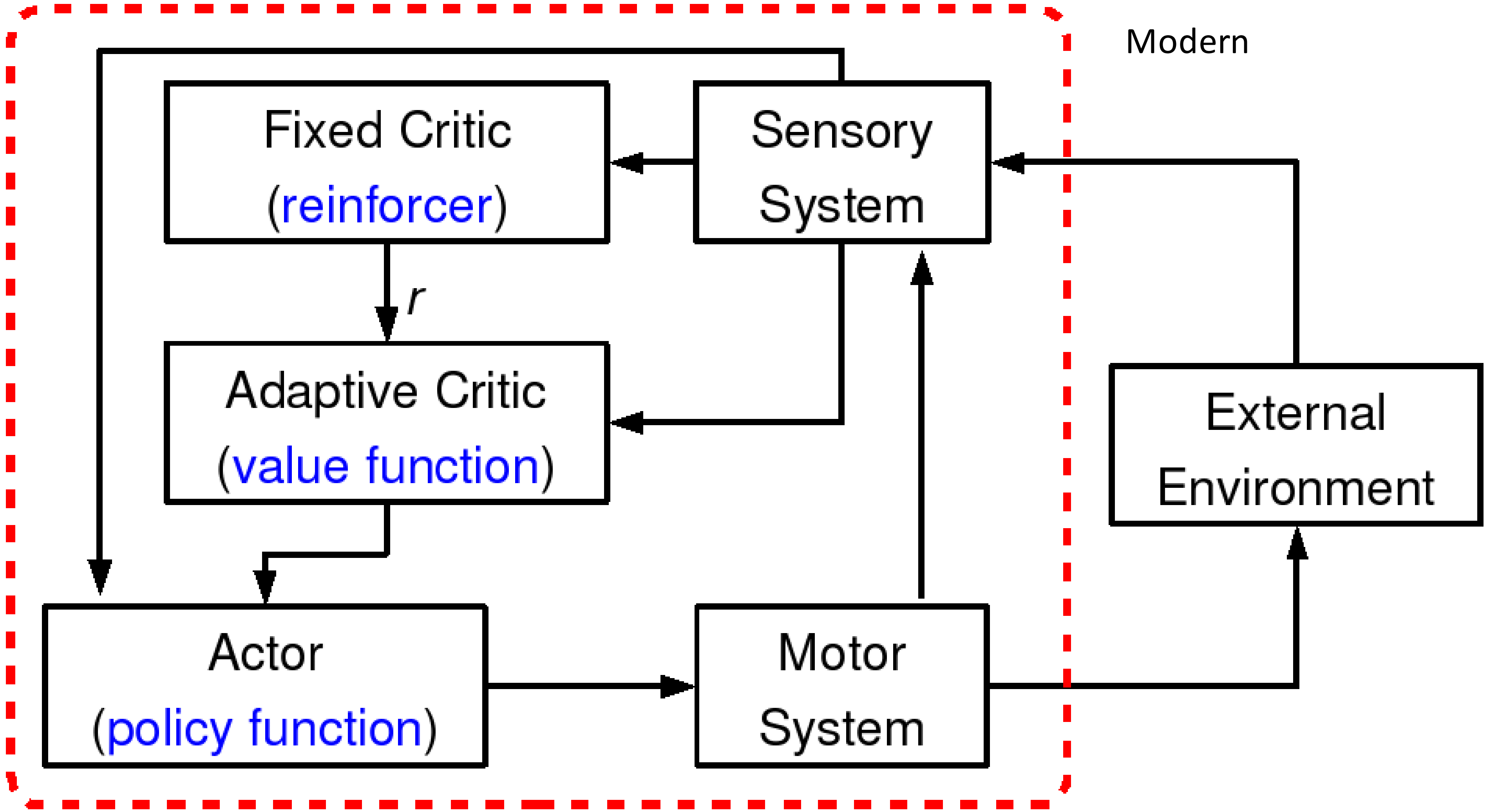
- Some tasks have naturally sparse feedback in time/space…
- Once trained, they can result in human (or even super-human) performance on many tasks



Value Function Methods

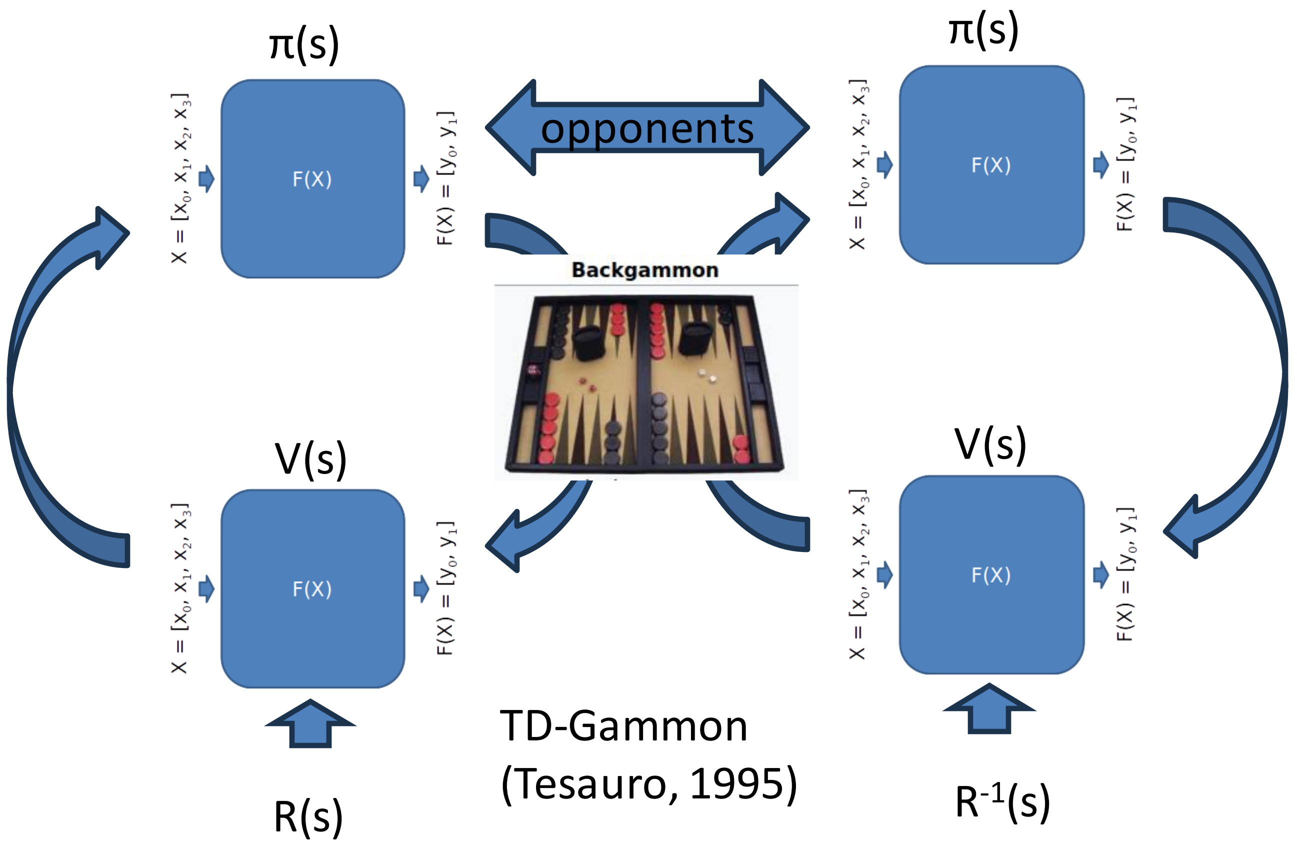
- Using neural networks to learn Q and V functions has unique challenges
- Neural networks provide a way to encode these functions in a way that is both practical and efficient
- On-line learning is still an active area of research
- Replay memories and target models have allowed for great strides in recent performance advances
  - Foundation: Tesauro, 1995; Boyan and Moore, 1995
  - AlphaGo (Google)
  - Atari (Google)
  - Robotics (Boston Dynamics)

Actor-Critic Framework
(Whitten, 1977)
(Barto, Sutton, and Anderson, 1983)



Early Success: TD-Gammon (Tesauro, 1995)

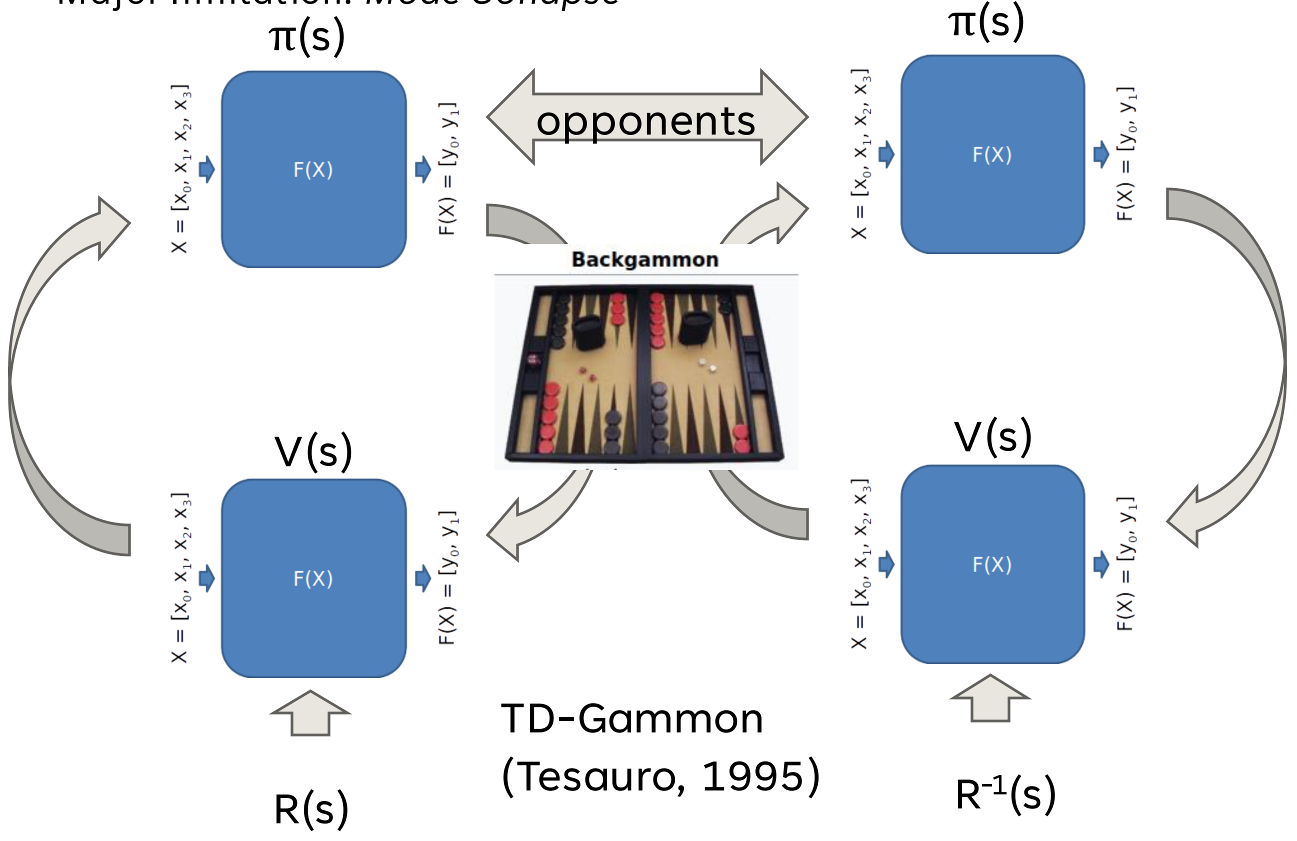
- Used a multilayer backpropagation neural network to learn the V function.
- Learned by playing against another agent.
- Reward signal was "a four component vector corresponding to the four possible outcomes of either White or Black winning either a normal win or a gammon."[1]



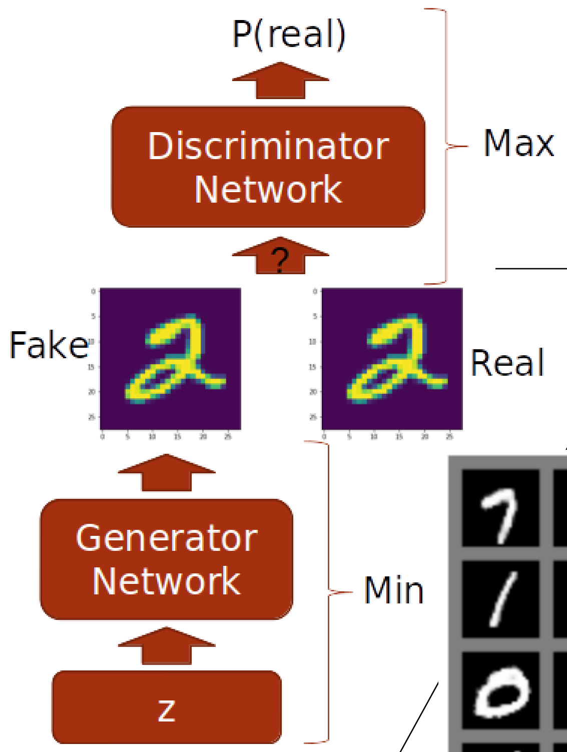
1 - http://www.research.ibm.com/massive/tdl.html

TRADITIONAL ADVERSARIAL LEARNING PROBLEMS

- Traditional *adversarial learning* involves (at minimum) two AI components which are learning to solve tasks with competing objectives.
- This approach was utilized from 1995 to 2020 to learn complex actor functions (eg. reinforcement learning) and generative function (eg. image generation)
- Major limitation: *Mode Collapse*

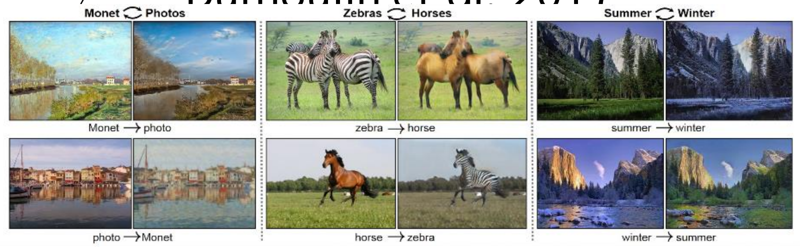


TD-Gammon (Tesauro, 1995)



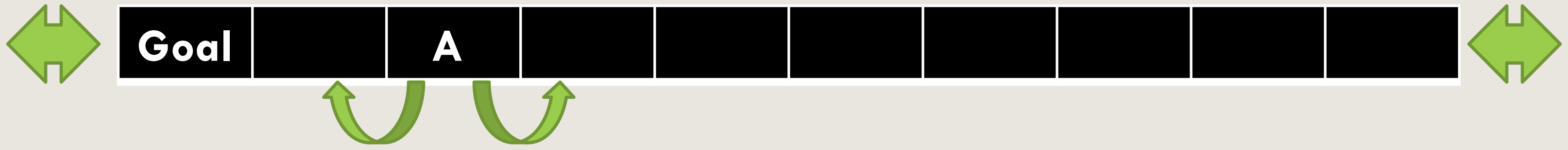
Generative Adversarial Networks (GANS)
Goodfellow et al., 2014

Dumoulin et al. 2017

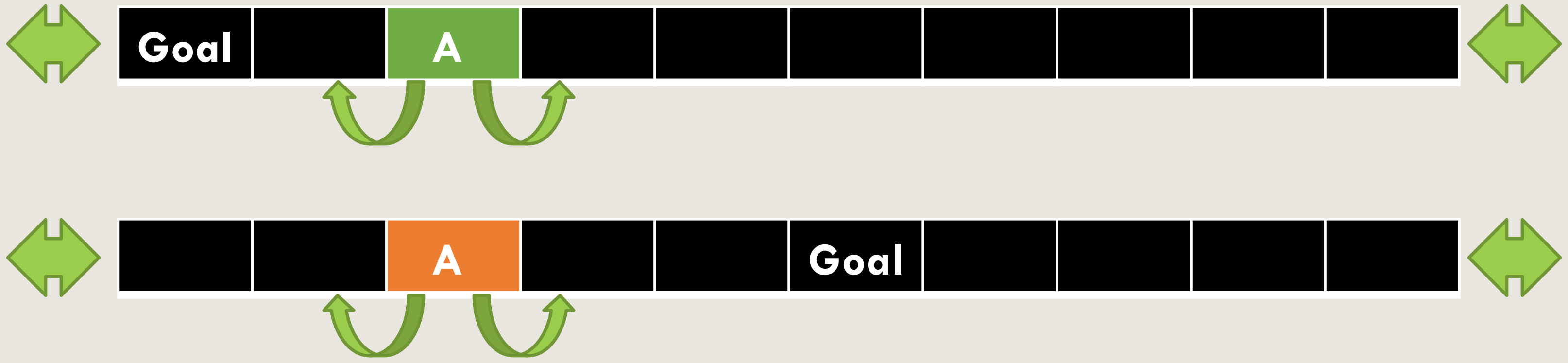


GAWWN - Zhu et al. 2017

PARTIAL OBSERVABILITY

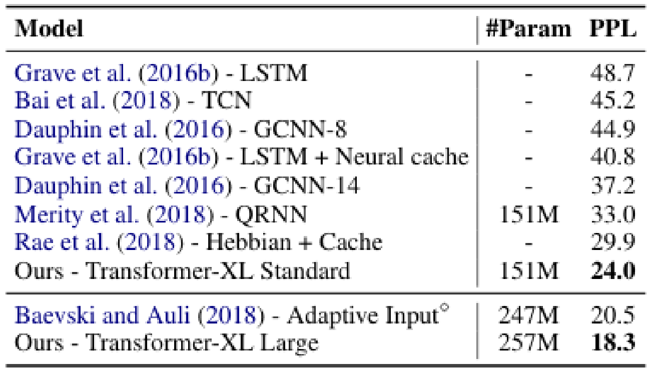


Standard 1D Maze



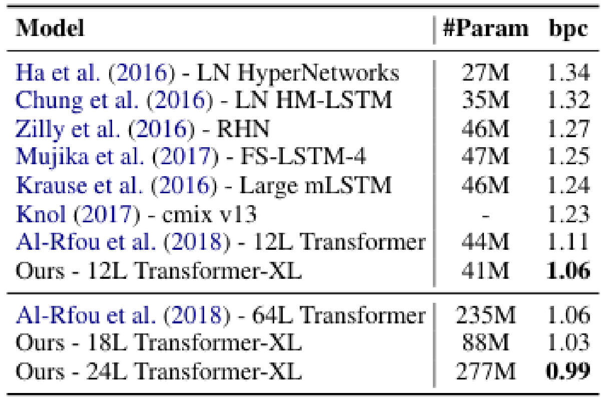
Partially Observable 1D Maze

OVERCOMING EXISTING CONTEXT LIMITS



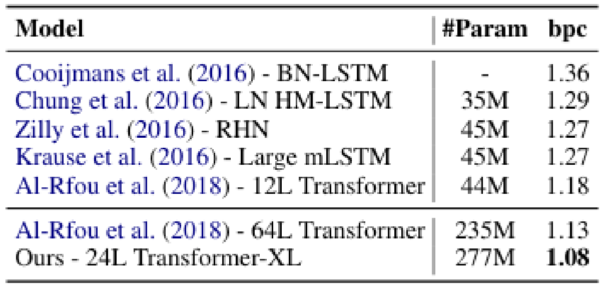
Model	#Param	PPL
Grave et al. (2016b) - LSTM	-	48.7
Bai et al. (2018) - TCN	-	45.2
Dauphin et al. (2016) - GCNN-8	-	44.9
Grave et al. (2016b) - LSTM + Neural cache	-	40.8
Dauphin et al. (2016) - GCNN-14	-	37.2
Merity et al. (2018) - QRNN	151M	33.0
Rae et al. (2018) - Hebbian + Cache	-	29.9
Ours - Transformer-XL Standard	151M	**24.0**
Baevski and Auli (2018) - Adaptive Input$^\diamond$	247M	20.5
Ours - Transformer-XL Large	257M	**18.3**

Table 1: Comparison with state-of-the-art results on WikiText-103. $^\diamond$ indicates contemporary work.



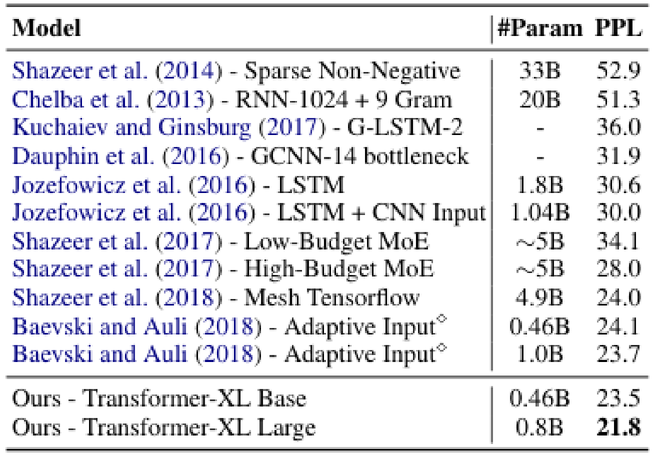
Model	#Param	bpc
Ha et al. (2016) - LN HyperNetworks	27M	1.34
Chung et al. (2016) - LN HM-LSTM	35M	1.32
Zilly et al. (2016) - RHN	46M	1.27
Mujika et al. (2017) - FS-LSTM-4	47M	1.25
Krause et al. (2016) - Large mLSTM	46M	1.24
Knol (2017) - cmix v13	-	1.23
Al-Rfou et al. (2018) - 12L Transformer	44M	1.11
Ours - 12L Transformer-XL	41M	**1.06**
Al-Rfou et al. (2018) - 64L Transformer	235M	1.06
Ours - 18L Transformer-XL	88M	1.03
Ours - 24L Transformer-XL	277M	**0.99**

Table 2: Comparison with state-of-the-art results on enwik8.



Model	#Param	bpc
Cooijmans et al. (2016) - BN-LSTM	-	1.36
Chung et al. (2016) - LN HM-LSTM	35M	1.29
Zilly et al. (2016) - RHN	45M	1.27
Krause et al. (2016) - Large mLSTM	45M	1.27
Al-Rfou et al. (2018) - 12L Transformer	44M	1.18
Al-Rfou et al. (2018) - 64L Transformer	235M	1.13
Ours - 24L Transformer-XL	277M	**1.08**

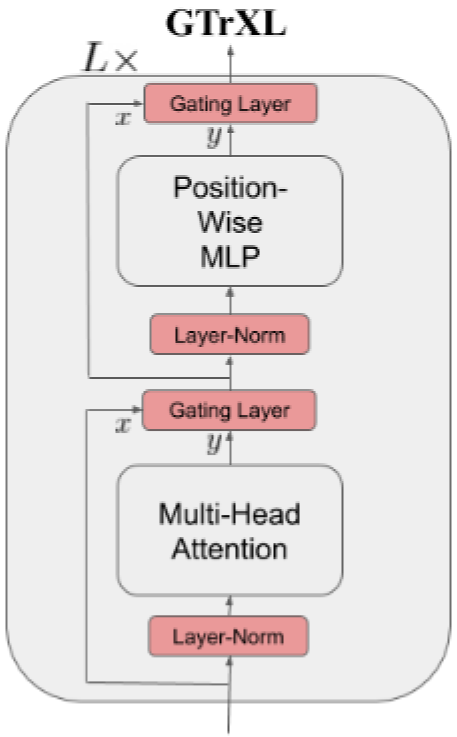
Table 3: Comparison with state-of-the-art results on text8.

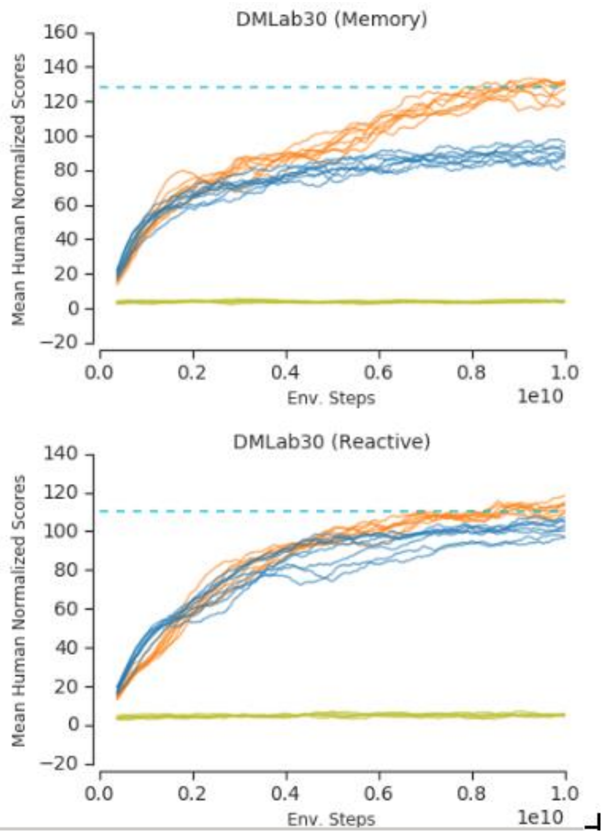


Model	#Param	PPL
Shazeer et al. (2014) - Sparse Non-Negative	33B	52.9
Chelba et al. (2013) - RNN-1024 + 9 Gram	20B	51.3
Kuchaiev and Ginsburg (2017) - G-LSTM-2	-	36.0
Dauphin et al. (2016) - GCNN-14 bottleneck	-	31.9
Jozefowicz et al. (2016) - LSTM	1.8B	30.6
Jozefowicz et al. (2016) - LSTM + CNN Input	1.04B	30.0
Shazeer et al. (2017) - Low-Budget MoE	~5B	34.1
Shazeer et al. (2017) - High-Budget MoE	~5B	28.0
Shazeer et al. (2018) - Mesh Tensorflow	4.9B	24.0
Baevski and Auli (2018) - Adaptive Input$^\diamond$	0.46B	24.1
Baevski and Auli (2018) - Adaptive Input$^\diamond$	1.0B	23.7
Ours - Transformer-XL Base	0.46B	23.5
Ours - Transformer-XL Large	0.8B	**21.8**

Table 4: Comparison with state-of-the-art results on One Billion Word. $^\diamond$ indicates contemporary work.

Dai, Zihang, et al. "Transformer-XL: Attentive language models beyond a fixed-length context." arXiv preprint arXiv:1901.02860 (2019).





Parisotto, Emilio, et al. "Stabilizing transformers for reinforcement learning." International Conference on Machine Learning. PMLR, 2020.

Policy Optimization

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$

<- Set λ=1 to match above equation…

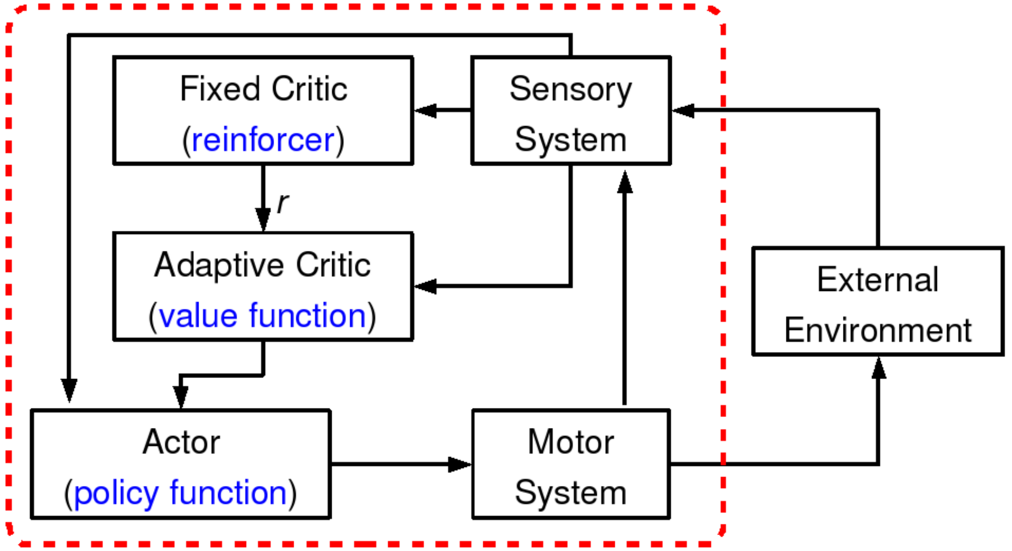
$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \operatorname{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

Algorithm 1 PPO-Clip

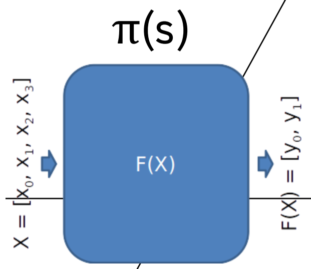
```
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, \ldots$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
           $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right),$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$
       typically via some gradient descent algorithm.
8: end for
```

Schulman et al., "Proximal Policy Optimization Algorithms." URL: http://arxiv.org/abs/1707.06347



PROXIMAL POLICY OPTIMIZATION

- Schulman et al., 2017
- Updates to the policy function are only allowed when small changes in estimated performance are predicted by the critic network
- Large policy updates prohibited



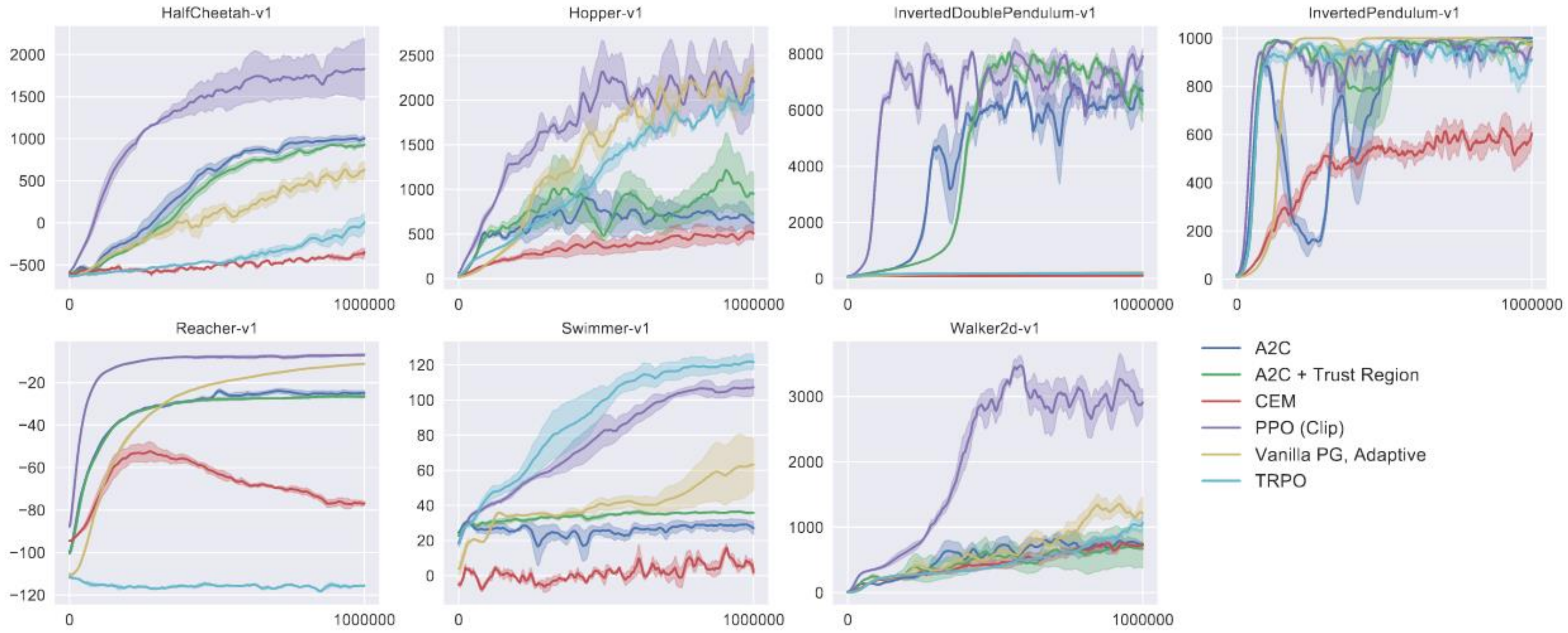


Figure 3: Comparison of several algorithms on several MuJoCo environments, training for one million timesteps.