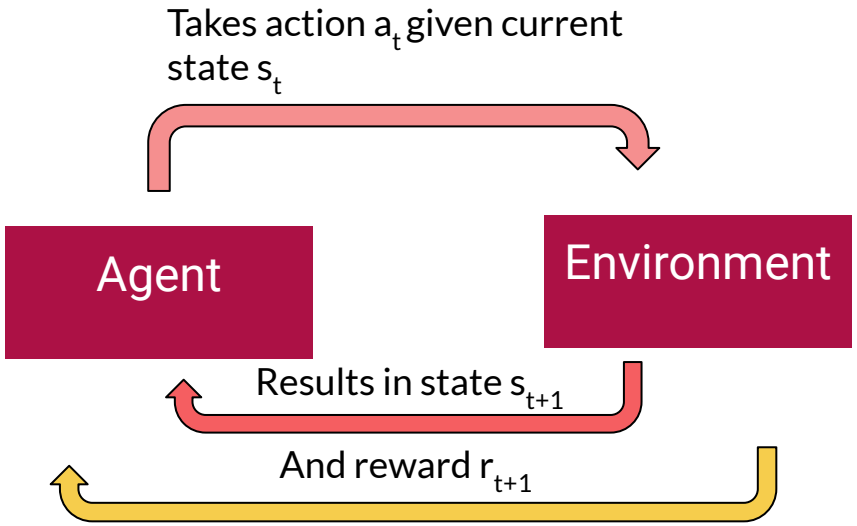
Analysis of Concurrent Reinforcement Learning Algorithms

Cathlyn Stone and Karan Sachdev
ECE 5510 - Multiprocessor Programming
December 12, 2019

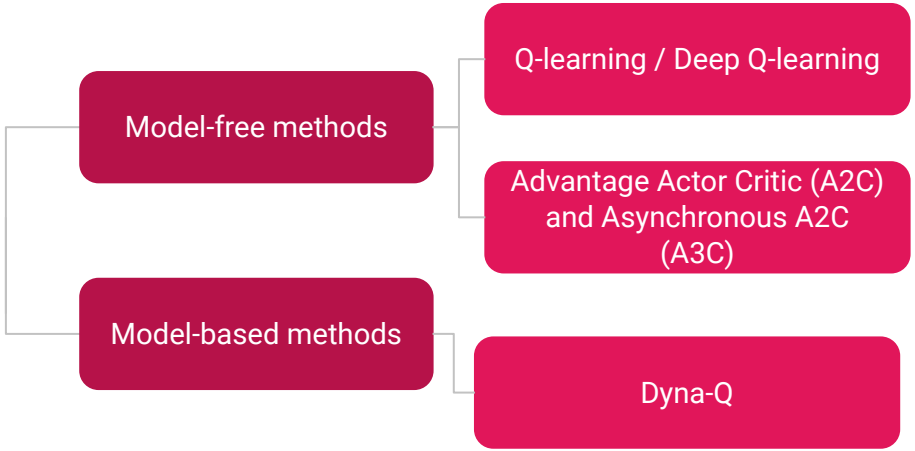
Overview of Reinforcement Learning

- Reinforcement learning is a type of machine learning, in which an agent interacts with its environment in order to maximize its expected reward



Types of Reinforcement Learning

- RL algorithms can be categorized as either **model-free** or **model-based**
- While many algorithms exist, we will specifically look at concurrent versions of:
  - **Deep Q-learning**
  - **Asynchronous Advantage Actor Critic (A3C)**
  - **Dyna-Q**



Model-free based algorithms

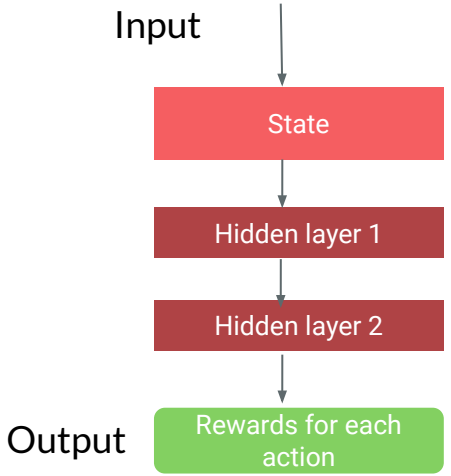
- Model-free methods try to directly learn an optimal policy or value function that will maximize expected reward
- Interact with an environment, but don't create a "model" of it

- **Q-learning** is a model-free method, in which a table Q holds expected rewards for each (state, action) pair, and is updated over time

Q(s, a)		Action (a)		
		1	2	3
State (s)	1	.012	.222	.567
	2	.561	-.45	-.11
	3	.39	.001	-.3
	4	-.41	-.63	0.00

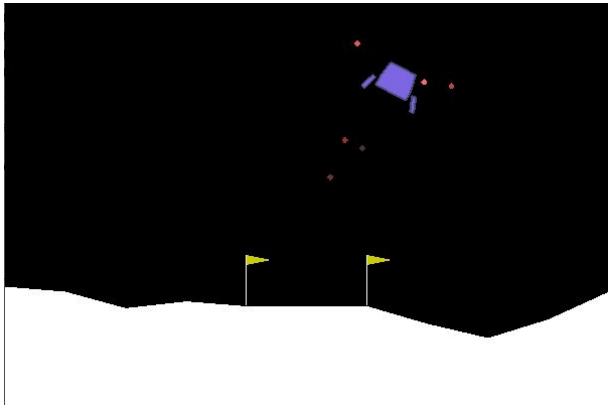
Deep Q-learning

- But Q-learning requires discrete states
- An agent playing an Atari game might have a state represented by a vector of continuous values
- Can use a neural net instead of Q-table to map an input state vector to expected rewards for each action



Playing Atari games

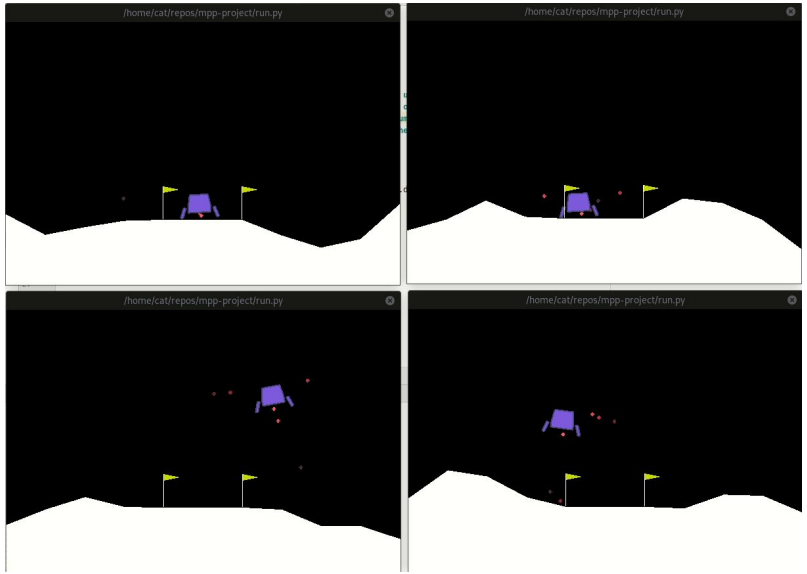
- We first wrote code to train a model to play Atari games using Deep Q-learning
  - Compared using the “LunarLander” game, but many games to choose from
- Implemented in Python, as it has many useful libraries for ML:
  - Torch (for neural networks)
  - OpenAI gym (the Atari game simulation)
- While the model eventually solves the game, it can take a while to train, depending on hardware, if just 1 processor is used



A single agent first starting to train

How to train faster?

- Use concurrency!
- We leveraged many of the ideas in the Gorila framework [1]
- Create *multiple agents* that play the game in parallel to explore the state space more quickly and converge to optimal network weights faster
- Occasionally synchronize network weights
- How to do this?



Multiple agents training in parallel

[1] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. arXiv preprint arXiv:1507.04296, 15 July 2015.

Python's shared memory model

- "Multithreading" vs. "Multiprocessing" library
- Multiprocessing provides true concurrency, but each process is given a separate memory space
  - Synchronizing shared data between threads a challenge
  - Not as easy as Java, where objects on heap can be shared by multiple Threads
- How to synchronize data?
  - Python provides a few ways to communicate data like Pipes and Queues
  - A few synchronization primitives like Locks and Barriers
  - Torch multiprocessing module also helpful, can mark some objects, such as Tensors, as "shared_memory"
  - Still some challenges here...

How to synchronize network weights?

- During training, each agent has their own network whose weights get updated
  - But to train faster, want to synchronize the network weights of all agents periodically
- Naïve approach:
  - Use mutual exclusion lock to make sure only one agent updates global set of parameters at a time
  - This becomes a sequential bottleneck, because there's quite a bit of computation that happens within the lock
  - Can train in less total episodes of the game, but each episode takes much longer because of threads contending for lock

```
# Acquire mutual exclusion lock
self.l.acquire()
try:
    self.global_optimizer.zero_grad() # zero out old gradients

    # Copy gradients from local network to global network
    for local_params, global_params in zip(self.local_network.parameters(), self.global_network.parameters()):
        global_params._grad = local_params._grad

    # Run SGD or other optimization algorithm - this will update global network parameters
    self.global_optimizer.step()
finally:
    self.l.release() # release lock
```

HogWild!

- An approach to *lock-free parallel SGD* (Stochastic Gradient Descent)
- Surprisingly simple:
  - If data is sparse, it will be rare enough for processors to overwrite one another as updates happen
  - And if they do, this shouldn't hurt results much
- Theoretical guarantees / proofs given in paper, but provides near linear speedup

```
# In agent constructor
# Create SGD optimizer locally that will operate on global network parameters
self.local_optimizer = optim.SGD(self.global_network.parameters(), lr=lr)

#----------------------------

# Later on, to update global network weights
loss = self.compute_loss(experiences) # Compute loss w.r.t. global network

self.local_optimizer.zero_grad() # Zero out gradients locally
loss.backward() # backpropagate loss to global network
self.local_optimizer.step() # Run SGD to update global network parameters
```

Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, pages 693–701, 2011.

Parameter Server

- The Gorila framework proposes a parameter server
- Each process sends its gradient updates to server, and then server applies gradients accumulated from many learners
- Local network parameters then synchronized with parameter server's values
- Server can be "sharded" for more parallelism
    - If gradients are held in a vector of size k, can create k shards that operate separately on each gradient

Other approaches / our approach

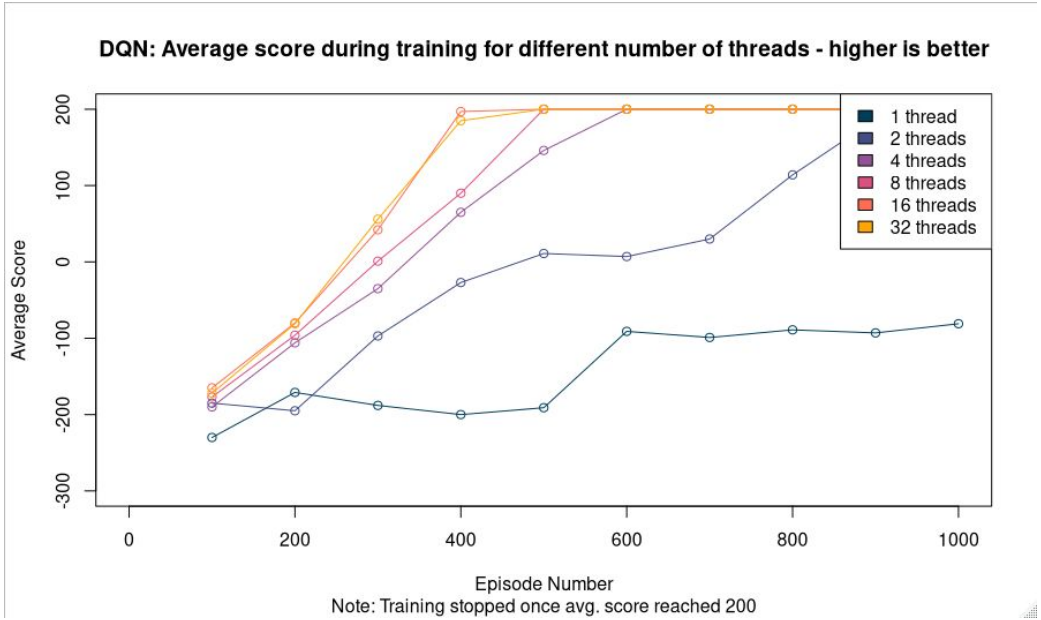
- Many other approaches to concurrent optimization exist
  - Hogwild++ , DogWild!, etc…
- Tried to implement parameter server
  - Created a Queue for agents to place gradient updates in (also tried sharded approach, creating k Queues)
  - Parameter server pulled from queue and applied to global parameters
  - Trained in less episodes but took too long to run
    - Perhaps having to copy tensors in and out of shared memory to place in and remove from queue?
- Went with HogWild! approach
  - Much better results: more processes = faster training
  - Found it important to synchronize less frequently with more threads
    - While no locking, overhead due to operating on shared memory
    - Couldn't find much documentation on Torch's shared memory, but could be interesting to look at in future how this is implemented and what the overheads are

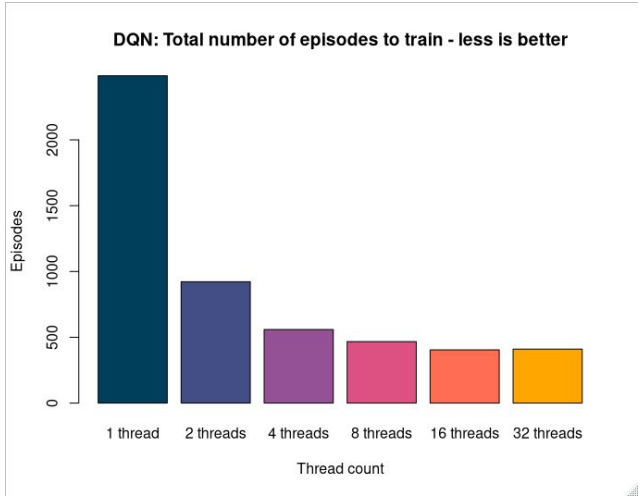
Local vs. global replay memory

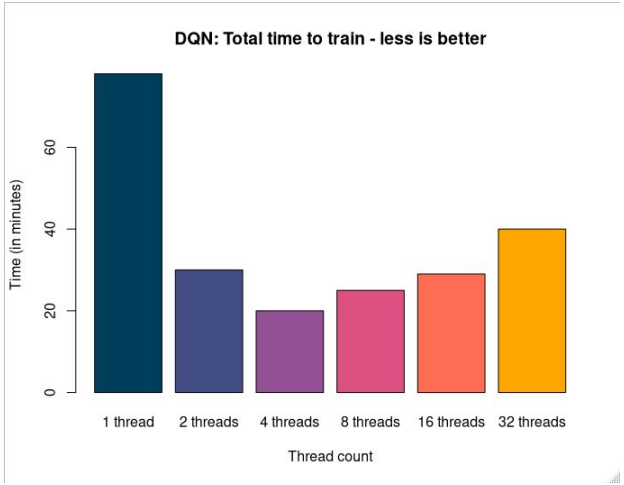
- As individual agents run, add experiences to a “replay memory”
- Could combine these into a global memory
- Would likely improve results to share a global memory
    - For now, kept each agent’s memory local to avoid having to deal with sharing memory

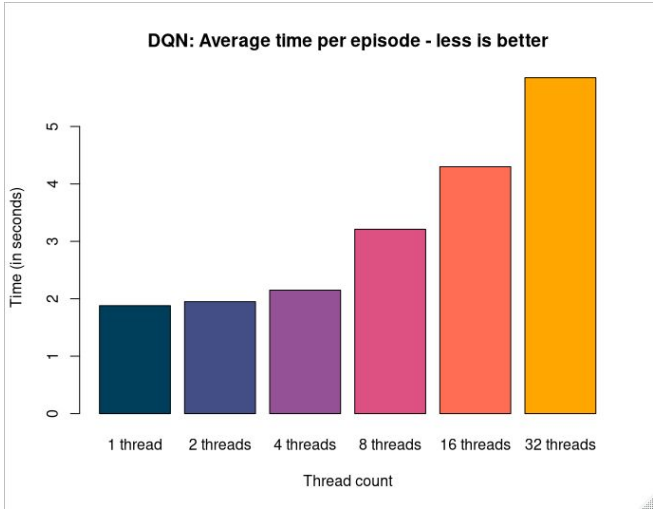
Evaluation of Parallel Deep Q-learning

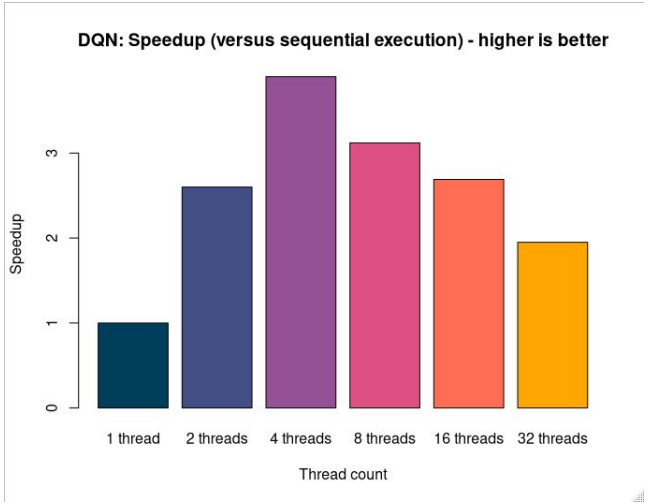
- Evaluation was done on Rlogin (40 cores) for varying number of threads
- Measured number of episodes to train and total time to complete




DQN: Total number of episodes to train - less is better
Episodes
0
500
1000
1500
2000
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count


DQN: Total time to train - less is better
Time (in minutes)
0
20
40
60
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count


DQN: Average time per episode - less is better
Time (in seconds)
0
1
2
3
4
5
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count


DQN: Speedup (versus sequential execution) - higher is better
Speedup
0
1
2
3
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count

Takeaways

- Using multiple processors allowed training to complete in less episodes
  - Except with 32 - actually took slightly more episodes than with 16 (410 vs. 406 episodes)
- But time per episode increased due to overhead of shared memory
- Improving how network weights are synchronized (with better concurrent optimization algorithm) could give better performance
- This was the most straightforward to implement of all the algorithms we looked at
  - However, techniques learned during DQN implementation helped in implementing the others

Asynchronous Advantage Actor Critic (A3C)

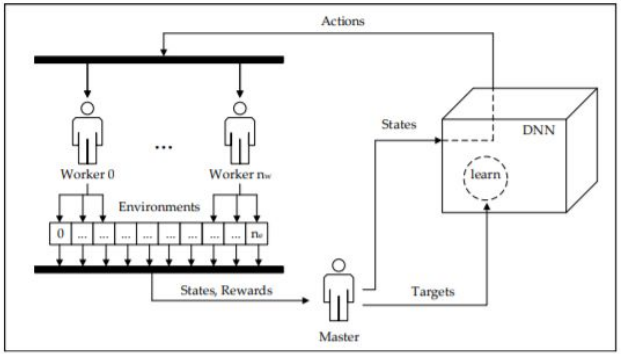
- We now have 2 local networks for each agent
  - One network estimates the value function, while another updates the policy
- Similarly to DQN, we created 2 global networks in which to synchronize local networks with for faster training
- Having implemented parallel DQN, the implementation of this algorithm was fairly straightforward. One approach we considered, but did not implement, was splitting each actor into 2 processes, allowing one to perform computation on the actor network, while the other performed computation on the critic network.

PARALLEL FRAMEWORK FOr A3C

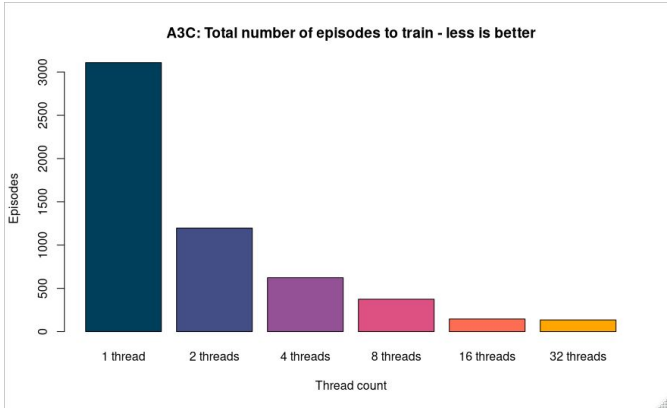
A general framework for deep reinforcement learning, where multiple actors can be trained synchronously on a single machine. A set of ne environment instances are maintained, where actions for all environment instances are generated from the policy. By having multiple environments instances in parallel it is likely that they will be exploring different locations of the state space at any given time, which reduces the correlation of encountered states and helps stabilize training.

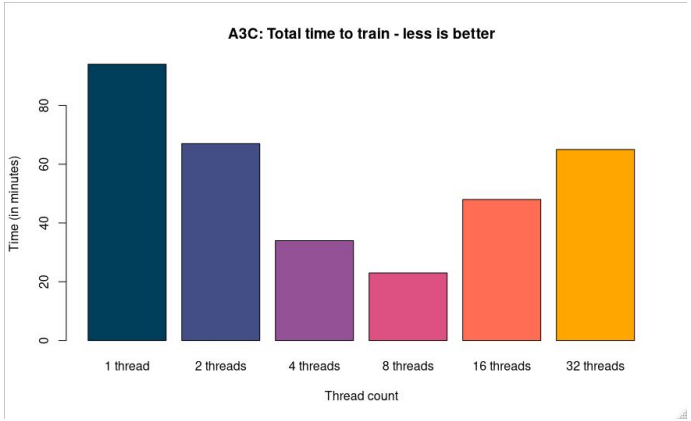
A3C algorithm maintains a policy π and an estimate value function, both approximated by deep neural network.

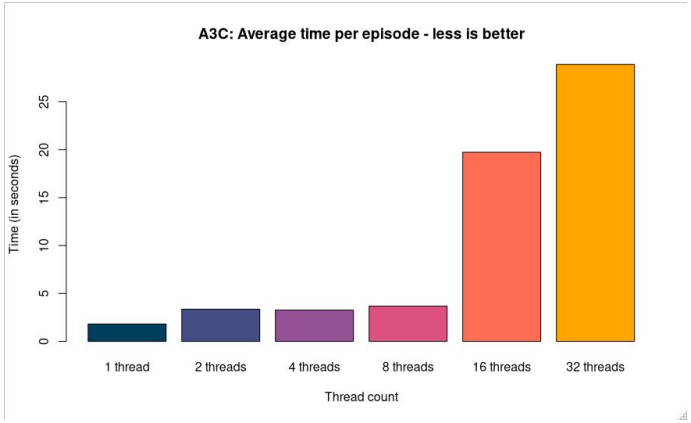
Even in this system the actor-learners compute gradients in parallel and shared parameters are updated in a hogwild fashion.

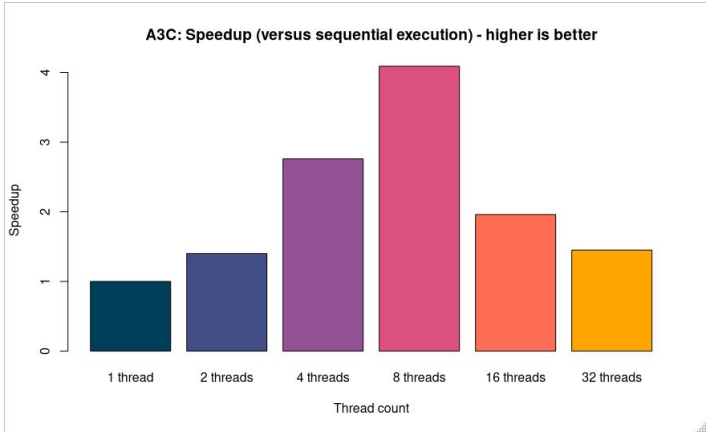


Evaluation of A3C









Model-based methods

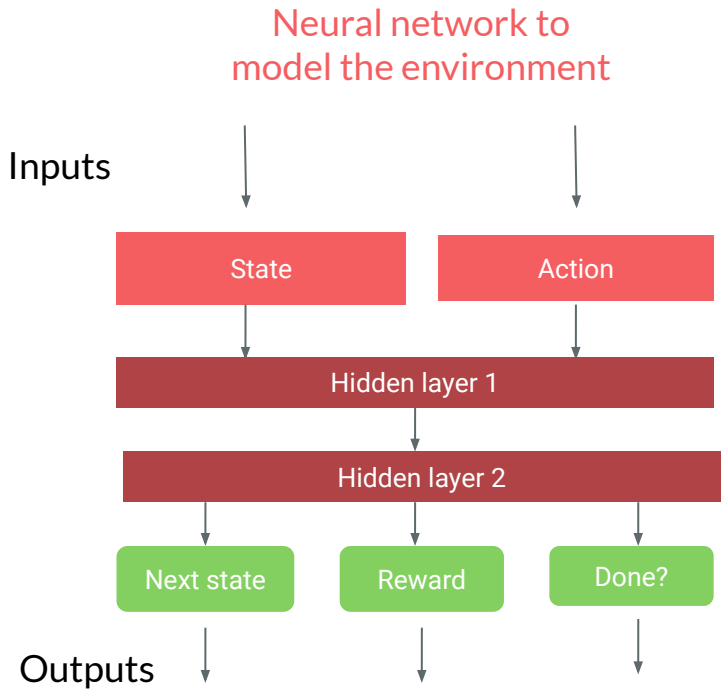
- Having multiple agents works fine for playing Atari games
  - We can have as many instances as we like
- What if there was only one “real” environment to explore?
  - i.e. stock market prediction
- Here, model-based methods such as Dyna-Q can be useful
  - An agent to interacts with the real environment,
  - Simultaneously trains a *model* of the environment (in addition to Q-network)
  - Then, additional exploration of the simulated model can occur
  - Exploration using the simulated model adds additional experiences which can used to train faster

Deep Dyna-Q

- We can think of our model as a neural network which maps:

  (state, action) -> (next_state, reward, done)

- So, our trained model (world model) would represent the Atari game itself



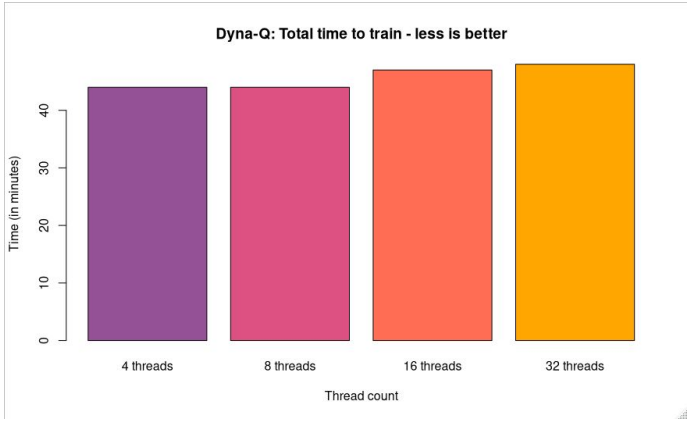
Parallel Deep Dyna-Q

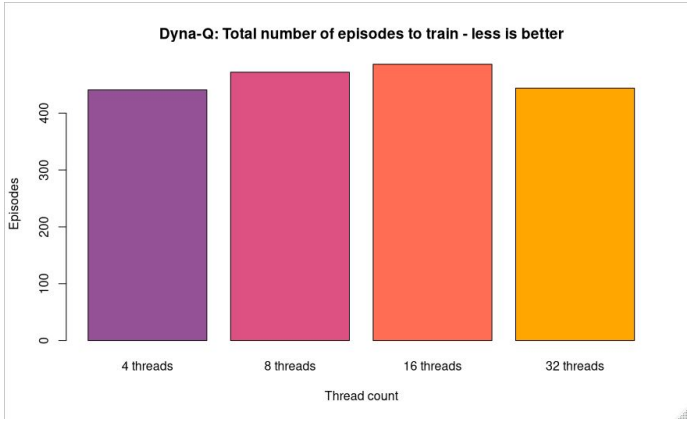
- Just one thread acts on real environment
  - Updates Q-network
- Multiple threads train world model and generate additional experiences using world model
  - Additional experiences then used to update the Q-network
- Needed to synchronize world model network parameters between agent threads
  - Did this the same as in DQN
- Need to communicate simulated experiences to other agents
  - Could create global replay memory
  - Instead used Queue to pass experiences to agents for training

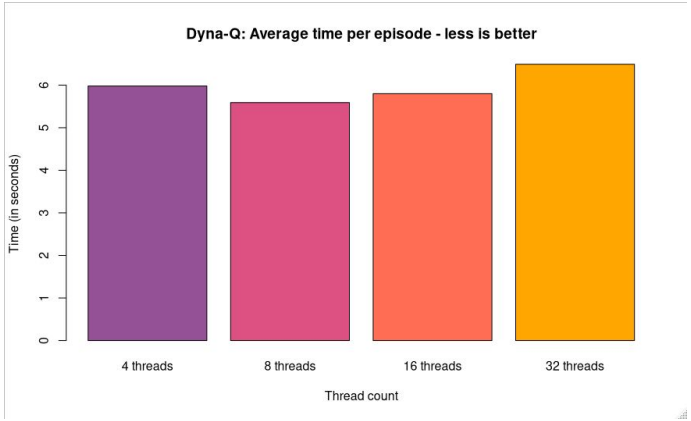
Evaluation of parallel deep Dyna-Q

- Did not evaluate using 1 thread
  - Split code for agent acting on real environment and agents acting on world model to different processes
- Using 2-3 threads did not work well
  - World model did not train quickly enough to provide useful simulated experiences
- Using too many threads added too many simulated experiences (vs. real experiences) to Q-network training
  - Limited number of updates of simulated experiences
  - Could make adjustments in the future to improve this ration
- Could be useful to add prioritized sweeping in the future
  - Have agents adding simulated experiences focus on exploring more unknown/interesting parts of the state space to update Q-network, rather than randomly or greedily exploring


Dyna-Q: Average score during training for different number of threads - higher is better
4 threads
8 threads
16 threads
32 threads
Average Score
200
100
0
-100
-200
-300
0
100
200
300
400
500
Episode Number
Note: Training stopped once avg. score reached 200


Dyna-Q: Total time to train - less is better
Time (in minutes)
0
10
20
30
40
4 threads
8 threads
16 threads
32 threads
Thread count


Dyna-Q: Total number of episodes to train - less is better
Episodes
0
100
200
300
400
4 threads
8 threads
16 threads
32 threads
Thread count


Dyna-Q: Average time per episode - less is better
Time (in seconds)
0
1
2
3
4
5
6
4 threads
8 threads
16 threads
32 threads
Thread count

Takeaways of Dyna-Q

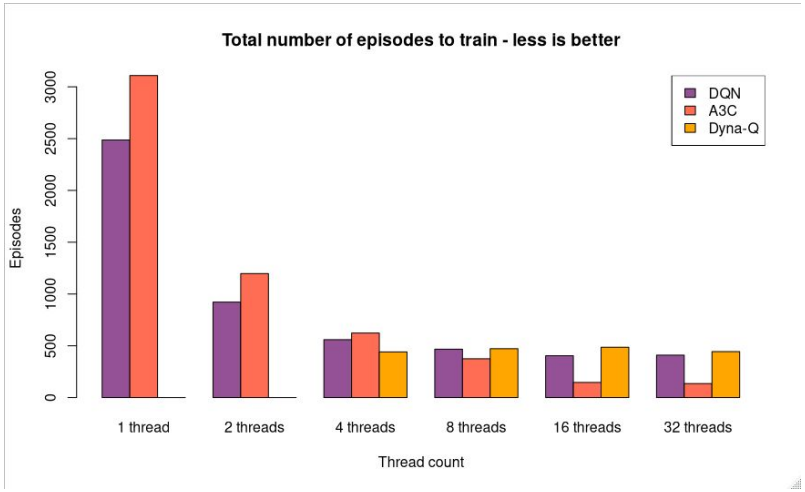
- DynaQ not necessarily a great RL algorithm to use for Atari games, but could be useful in other domains
- Think there is potential for this to work well, but needs adjustment to better determine how to balance real/simulated experiences
  - Can use world agents to balance exploration/exploitation as in [1]
  - Prioritized sweeping [2] also looks promising

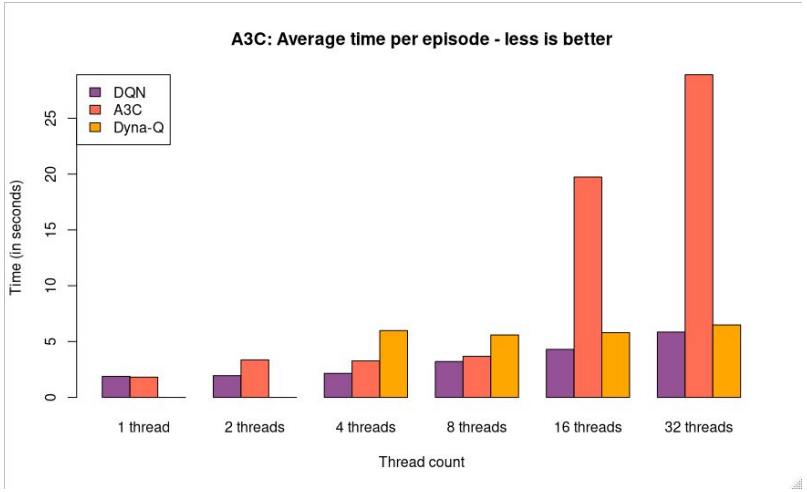
[1] Takeshi Tateyama, Seiichi Kawata and Toshiki Shimomura, "Parallel Reinforcement Learning Systems Using Exploration Agents and Dyna-Q Algorithm," *SICE Annual Conference 2007*, Takamatsu, 2007, pp. 2774-2778. doi: 10.1109/SICE.2007.4421460

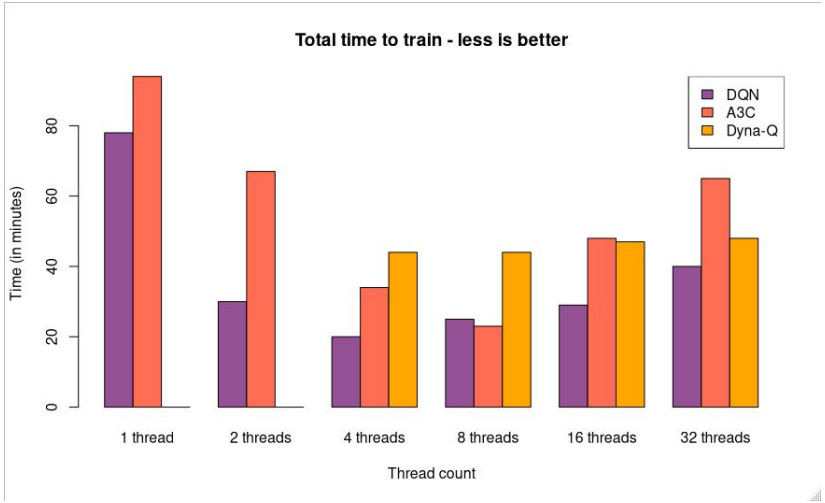
[2] Andrew W. Moore and Christopher G. Atkeson. 1993. Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time. *Mach. Learn. 13*, 1 (October 1993), 103-130. DOI: https://doi.org/10.1023/A:1022635613229

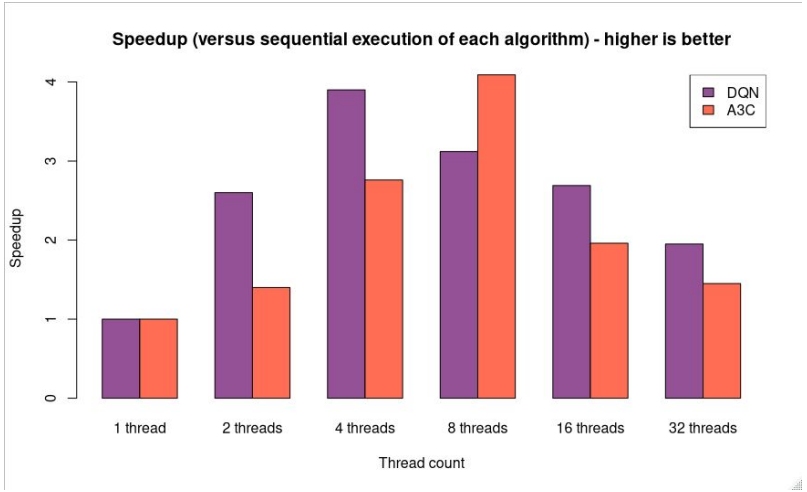
Evaluation of all algorithms

- What worked best depended on number of threads
  - 4-8 threads generally best overall
- DQN generally faster than A3C or Dyna-Q


Total number of episodes to train - less is better
Episodes
0
500
1000
1500
2000
2500
3000
DQN
A3C
Dyna-Q
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count


A3C: Average time per episode - less is better
Time (in seconds)
0
5
10
15
20
25
DQN
A3C
Dyna-Q
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count


Total time to train - less is better
Time (in minutes)
0
20
40
60
80
DQN
A3C
Dyna-Q
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count


Speedup (versus sequential execution of each algorithm) - higher is better
Speedup
0
1
2
3
4
DQN
A3C
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
Thread count

Future Improvements

- Use Torch's C++ library instead
  - Could make things run faster, and possibly make sharing memory easier
- Examine other concurrent optimization algorithms in order to synchronize global network parameters with less overhead
  - Many options here, some interesting ones include:
  - Asynchronous Decentralized Parallel Stochastic Gradient Descent [1]
  - A Linearly Convergent Proximal Gradient Algorithm for Decentralized Optimization [2]
- Run on GPU instead of CPU
- Try using global replay memory vs. local memory per actor
- Try optimizing hyperparameters selected (like learning rate, etc…)
- Try these algorithms on other types of problems besides Atari games, or try implementing different RL algorithms

[1] Xiangru Lian, Wei Zhang, Ce Zhang, Ji Liu. (2018). Asynchronous Decentralized Parallel Stochastic Gradient Descent. *ICML 2018*: 3049-3058
[2] Alghunaim, Sulaiman & Yuan, Kun & Sayed, Ali. (2019). A Linearly Convergent Proximal Gradient Algorithm for Decentralized Optimization. *33rd Conference on Neural Information Processing Systems*.

Thank you!! :)

- Questions?