Distributed Deep Q-Learning

Hao Yi Ong

joint work with K. Chavez, A. Hong

Stanford University

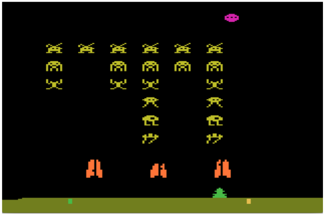
Box, 6/3/15

Outline

Motivation

- long-standing challenge of reinforcement learning (RL)
  - control with high-dimensional sensory inputs (*e.g.*, vision, speech)
  - shift away from reliance on hand-crafted features
- utilize breakthroughs in deep learning for RL [M$^+$13, M$^+$15]
  - extract high-level features from raw sensory data
  - learn better representations than handcrafted features with neural network architectures used in supervised and unsupervised learning
- create fast learning algorithm
  - train efficiently with stochastic gradient descent (SGD)
  - distribute training process to accelerate learning [DCM$^+$12]

Success with Atari games



Theoretical complications

deep learning algorithms require

- huge training datasets
  - sparse, noisy, and delayed reward signal in RL
  - delay of $\sim 10^3$ time steps between actions and resulting rewards
  - *cf.* direct association between inputs and targets in supervised learning
- independence between samples
  - sequences of highly correlated states in RL problems
- fixed underlying data distribution
  - distribution changes as RL algorithm learns new behaviors

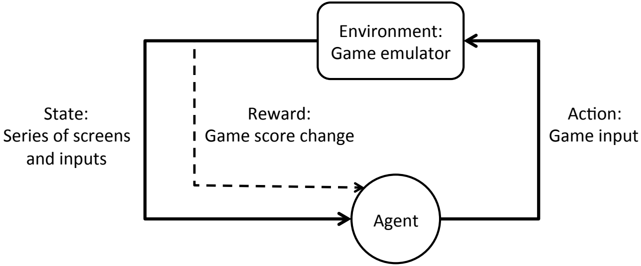
Goals

distributed deep RL algorithm

- robust neural network agent
  - must succeed in challenging test problems
- control policies with high-dimensional sensory input
  - obtain better internal representations than handcrafted features
- fast training algorithm
  - efficiently produce, use, and process training data

Outline

Playing games



objective: learned policy maximizes future rewards

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'},$$

- discount factor γ
- reward change at time t' $r_{t'}$

State-action value function

- basic idea behind RL is to estimate

$$Q^\star(s,a) = \max_\pi \mathbf{E}\left[R_t \mid s_t = s, a_t = a, \pi\right],$$

where π maps states to actions (or distributions over actions)

- optimal value function obeys Bellman equation

$$Q^\star(s,a) = \mathop{\mathbf{E}}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^\star(s', a') \mid s, a\right],$$

where $\mathcal{E}$ is the MDP environment

Value approximation

- typically, a linear function approximator is used to estimate $Q^\star$

$$Q\left(s, a; \theta\right) \approx Q^\star\left(s, a\right),$$

  which is parameterized by θ

- we introduce the Q-network
  - nonlinear neural network state-action value function approximator
  - "Q" for Q-learning

Q-network

- trained by minimizing a sequence of loss functions

$$L_i(\theta_i) = \mathop{\mathbf{E}}_{s,a\sim\rho(\cdot)} \left[\left(y_i - Q(s,a;\theta_i) \right)^2 \right],$$

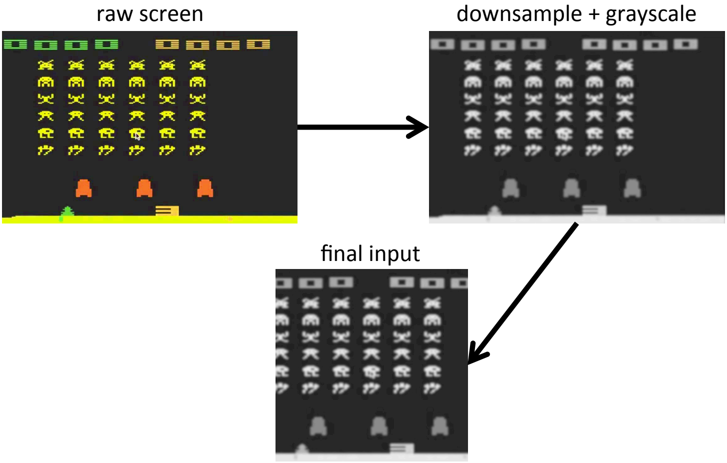
  with

  - iteration number i
  - target $y_i = \mathbf{E}_{s'\sim\mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a \right]$
  - “behavior distribution” (exploration policy) $\rho(s,a)$

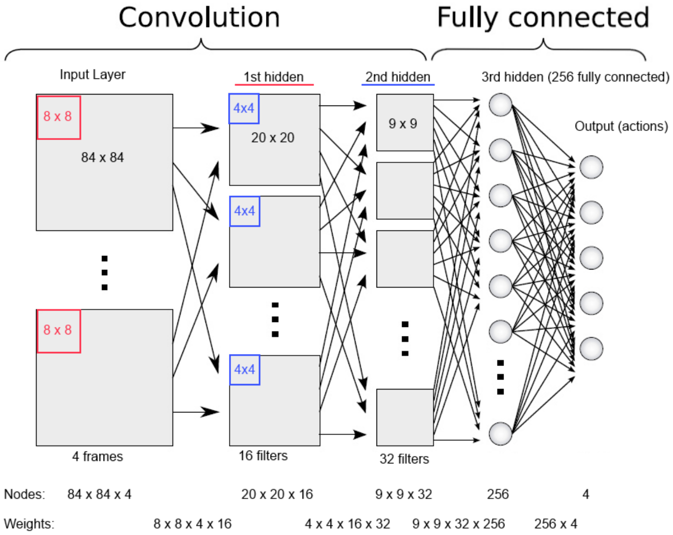
- architecture varies according to application

Outline

Preprocessing

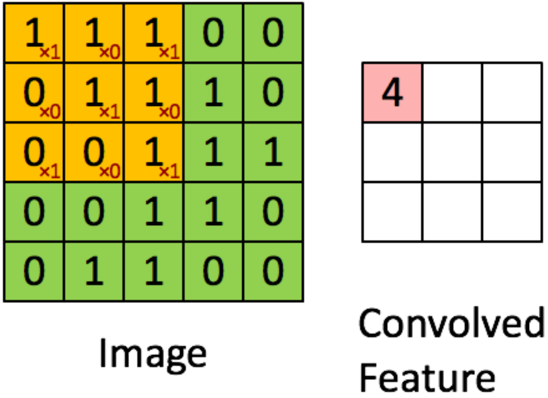


Network architecture



Convolutional neural network

- biologically-inspired by the visual cortex
- CNN example: single layer, single frame to single filter, stride $= 1$



Stochastic gradient descent

- optimize Q-network loss function by gradient descent

$$Q\left(s,a;\theta\right) := Q\left(s,a;\theta\right) + \alpha \nabla_{\theta} Q\left(s,a;\theta\right),$$

  with

  - learning rate α

- for computational expedience

  - update weights after every time step
  - avoid computing full expectations
  - replace with single samples from ρ and $\mathcal{E}$

Q-learning

$$Q(s,a) := Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

- model free RL
  - avoids estimating $\mathcal{E}$
- off-policy
  - learns policy $a = \operatorname{argmax}_a Q(s,a;\theta)$
  - uses behavior distribution selected by an ϵ-greedy strategy

Experience replay

a kind of short-term memory

- trains optimal policy using "behavior policy" (off-policy)
  - learns policy $\pi^{\star}(s) = \operatorname{argmax}_a Q(s, a; \theta)$
  - uses an ϵ-greedy strategy (behavior policy) for state-space exploration
- store agent's experiences at each time step

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

  - experiences form a replay memory dataset with fixed capacity
  - execute Q-learning updates with random samples of experience

Serial deep Q-learning

given replay memory $\mathcal{D}$ with capacity N

initialize Q-networks Q, $\hat{Q}$ with same random weights θ

repeat until timeout

initialize frame sequence $s_1 = \{x_1\}$ and preprocessed state $\phi_1 = \phi(s_1)$

for $t = 1, \ldots, T$

1. select action $a_t = \begin{cases} \max_a Q(\phi(s_t), a; \theta) & \text{w.p. } 1-\epsilon \\ \text{random action} & \text{otherwise} \end{cases}$
2. execute action a_t and observe reward r_t and frame x_{t+1}
3. append $s_{t+1} = (s_t, a_t, x_{t+1})$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
4. store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
5. uniformly sample minibatch $(\phi_j, a_j, r_j, \phi_{j+1}) \sim \mathcal{D}$
6. set $y_j = \begin{cases} r_j & \text{if } \phi_{j+1} \text{ terminal} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$
7. perform gradient descent step for Q on minibatch
8. every C steps reset $\hat{Q} = Q$

Theoretical complications

deep learning algorithms require

- huge training datasets
- independence between samples
- fixed underlying data distribution

Deep Q-learning

avoids theoretical complications

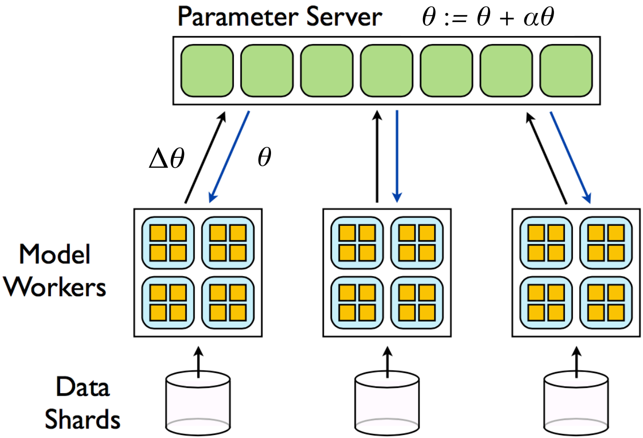
- greater data efficiency
  - each experience potentially used in many weight udpates
- reduce correlations between samples
  - randomizing samples breaks correlations from consecutive samples
- experience replay averages behavior distribution over states
  - smooths out learning
  - avoids oscillations or divergence in gradient descent

Cat video

Outline

Data parallelism

downpour SGD: generic asynchronous distributed SGD



Model parallelism

on each worker machine

- computation of gradient is pushed down to hardware
  - parallelized according to available CPU/GPU resources
  - uses the Caffe deep learning framework
- complexity scales linearly with number of parameters
  - GPU provides speedup, but limits model size
  - CPU slower, but model can be much larger

Implementation

- data shards are generated locally on each model worker in real-time
  - data is stored independently for each worker
  - since game emulation is simple, generating data is fast
  - simple fault tolerance approach: regenerate data if worker dies
- algorithm scales very well with data
  - since data lives locally on workers, no data is sent
- update parameter with gradients using RMSprop or AdaGrad
- communication pattern: multiple asynchronous all-reduces
  - one-to-all and all-to-one, but asynchronous for every minibatch

Implementation

- bottleneck is parameter update time on parameter server
  - *e.g.*, if parameter server gradient update takes 10 ms, then we can only do up to 100 updates per second (using buffers, etc.)
- trade-off between parallel updates and model staleness
  - because worker is likely using a stale model, the updates are "noisy" and not of the same quality as in serial implementation

Outline

Evaluation

Snake

- parameters
  - snake length grows with number of apples eaten ($+1$ reward)
  - one apple at any time, regenerated once eaten
  - $n \times n$ array, with walled-off world (-1 if snake dies)
  - want to maximize score, equal to apples eaten (minus 1)
- complexity
  - four possible states for each cell: {empty, head, body, apple}
  - state space cardinality is $O\left(n^4 2^{n^2}\right)$ (-ish)
  - four possible actions: {north, south, east, west}

Software

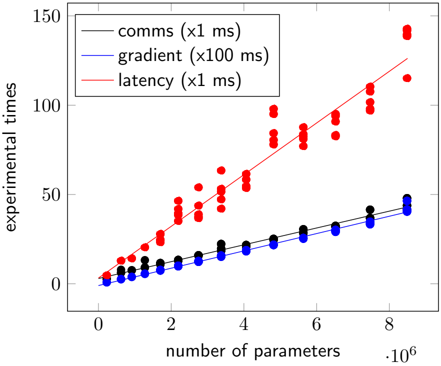
- at initialization, broadcast neural network architecture
  - each worker spawns Caffe with architecture
  - populates replay dataset with experiences via random policy
- for some number of iterations:
  - workers fetch latest parameters for Q network from server
  - compute and send gradient update
  - parameters updated on server with RMSprop or AdaGrad (requires $O(p)$ memory and time)
- Lightweight use of Spark
  - shipping required files and serialized code to worker machines
  - partitioning and scheduling number of updates to do on each worker
  - coordinating identities of worker/server machines
  - partial implementation of generic interface between Caffe and Spark
- ran on dual core Intel i7 clocked at 2.2 GHz, 12 GB RAM

Complexity analysis

- model complexity
  - determined by architecture; roughly on the order of number of parameters
- gradient calculation via backpropagation
  - distributed across worker's CPU/GPU, linear with model size
- communication time and cost
  - for each update, linear with model size

Compute/communicate times

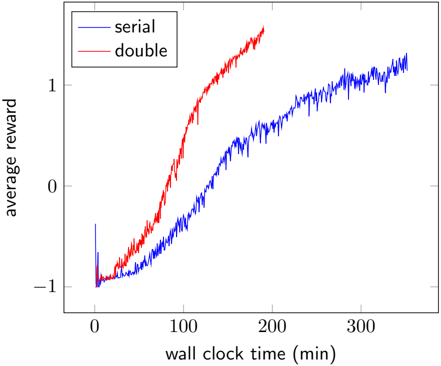
- compute/communicate time scales linearly with model size



- process is compute-bound by gradient calculations
- upper bound on update rate inversely proportional to model size
- with many workers in parallel, independent of batch size

Serial vs. distributed

- performance scales linearly with number of workers



Example game play

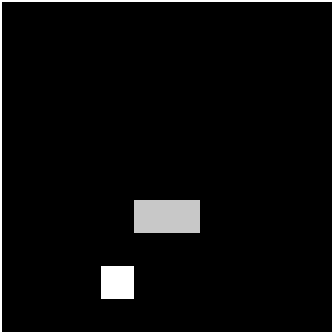


Figure: Dumb snake.

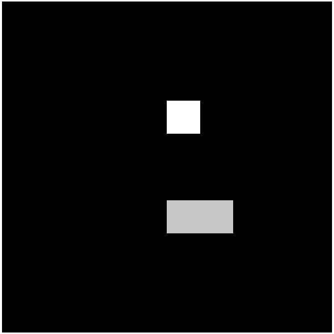


Figure: Smart snake.

Outline

Summary

- deep Q-learning [M+13, M+15] scales well via DistBelief [DCM+12]
- asynchronous model updates accelerate training despite lower update quality (vs. serial)

Contact

questions, code, ideas, go-karting, swing dancing, ...

haoyi.ong@gmail.com

References

- Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al.
  Large scale distributed deep networks.
  In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.
- V. Mnih et al.
  Playing Atari with deep reinforcement learning.
  arXiv preprint arXiv:1312.5602, 2013.
- V. Mnih et al.
  Human-level control through deep reinforcement learning.
  Nature, 518(7540):529–533, 2015.