IE 398/CS 398 Deep Learning

University of Illinois at Urbana-Champaign

Spring 2018

Lecture 11

Distributed training of deep learning models.

- Data parallelization
- Model parallelization

Synchronous gradient descent.

- For $k = 1, 2, \ldots, K$:
  - Data batches $d_1, d_2, \ldots, d_N \subset \mathcal{D}$ are loaded onto the CPUs of the nodes $n = 1, 2, \ldots, N$.[1] Each d_n is itself composed of a minibatch of M data samples.
  - Each node holds a copy of the neural network model M_θ where θ is the parameter vector.
  - Transfer d_n to node n's GPU.
  - The computation of the gradients g_n is parallelized using node n's GPU. The gradient $g_n \equiv \frac{\partial E(d_n, M_\theta)}{\partial \theta}$ where $E(d, M_\theta)$ is the error of the model M_θ for the data batch d.
  - The gradients $g_1, g_2, \ldots, g_N$ are averaged. Let $\tilde{g} = \frac{1}{N} \sum_{n=1}^{N} g_n$.
  - The model M_θ is updated across all nodes using the gradient $\tilde{g}$.

[1] $\mathcal{D}$ is the original dataset which is stored on Online storage and is too large to fit into memory all at once; d_n are random subsets of $\mathcal{D}$.

- Let T be the time required to complete the task using 1 node.
- Let T_N be the time required to complete the task using N nodes.
- The **efficiency** is

$$E_N = \frac{T}{N \times T_N} \times 100\%. \tag{1}$$

$$E_N = \frac{T}{N \times T_N} \times 100\%. \tag{2}$$

- If $T_N = \frac{T}{N}$, $E_N = 100\%$.
- If $T_N = \frac{2T}{N}$, $E_N = 50\%$.
- "Perfectly parallel" tasks requiring no communication have 100% efficiency. An example is Monte Carlo simulation.
- Tasks requiring heavy communication will have less than 100% efficiency.
- Efficiency may be a function of N.

- **Total minibatch size** is $N \times M$.
- Therefore, a larger learning rate can be used, which leads to faster convergence.
- There is an upper limit on the size of the learning rate (i.e., gradient descent on the entire dataset).
- Therefore, Synchronous Gradient Descent cannot achieve linear scaling efficiency for all N.

- Heavy communication
- Node i must communicate with all other nodes $j = 1, 2, \ldots, N$.
- Larger models incur large communication costs
- Update occurs only once all of the workers finish.

Asynchronous stochastic gradient descent.

- Initialize model M_θ on the "parameter server" node 0.
- For $n = 1, 2, \ldots, N$ **(in parallel)**:
  - For $k = 1, 2, \ldots, K$:
    - Copy θ from node 0 and set $\theta_n = \theta$.
    - Load minibatch of data d_n.
    - Calculate the gradient

$$g_n = \frac{\partial E(d_n, M_{\theta_n})}{\partial \theta}. \tag{3}$$

    - Update M_θ on node 0:

$$\theta = \theta - \alpha g_n. \tag{4}$$

- Asynchronous SGD does not have to wait for "slow" workers: rapid, noisy updates.
- Communication bottlenecks at parameter server
- Total communication cost grows linearly in number of nodes N
- Communication cost for synchronous gradient descent is $\log(N)$.
- See "Optimization of Collective Communication Operations in MPICH" by Thakur, Rabenseifner, and Gropp (2005).

- Asynchronous SGD is biased.
- Communication times increase with the number of nodes N.
- Bias grows with N.
- Convergence rate can actually slow down if N is too large.

"Large Scale Distributed Deep Networks" by Dean et al. (2015).

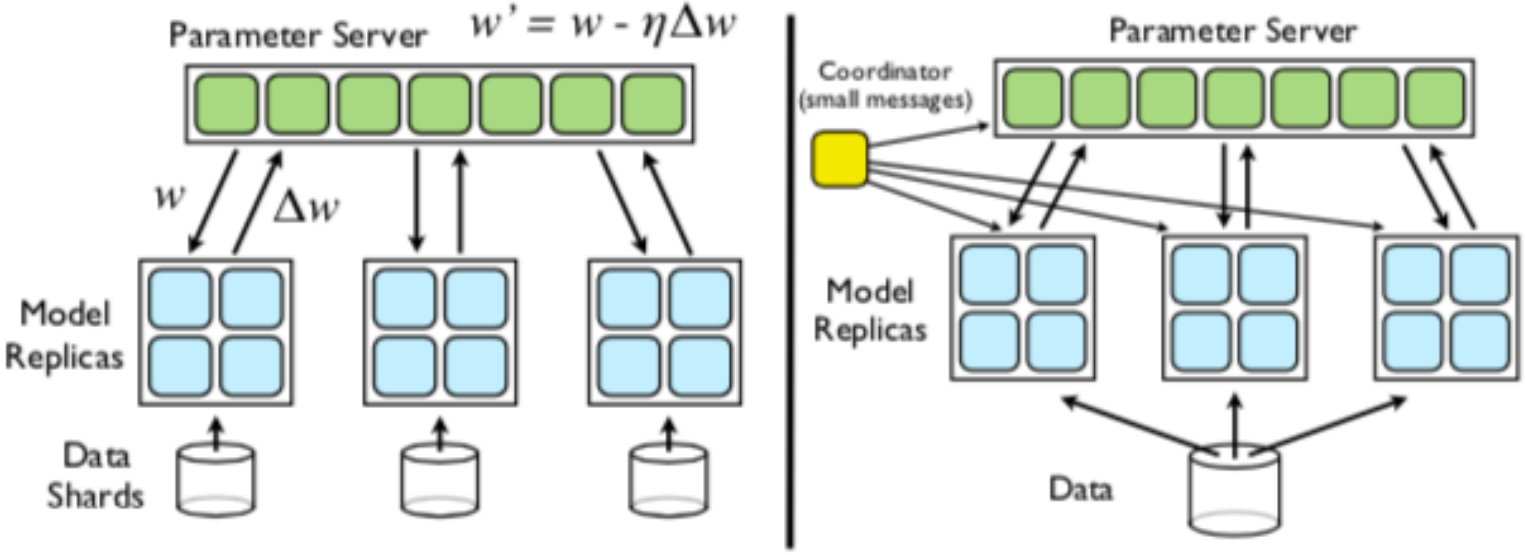


Figure 2: Left: Downpour SGD. Model replicas asynchronously fetch parameters w and push gradients Δw to the parameter server. Right: Sandblaster L-BFGS. A single 'coordinator' sends small messages to replicas and the parameter server to orchestrate batch optimization.

Source: "Large Scale Distributed Deep Networks" by Dean et al. (2015).

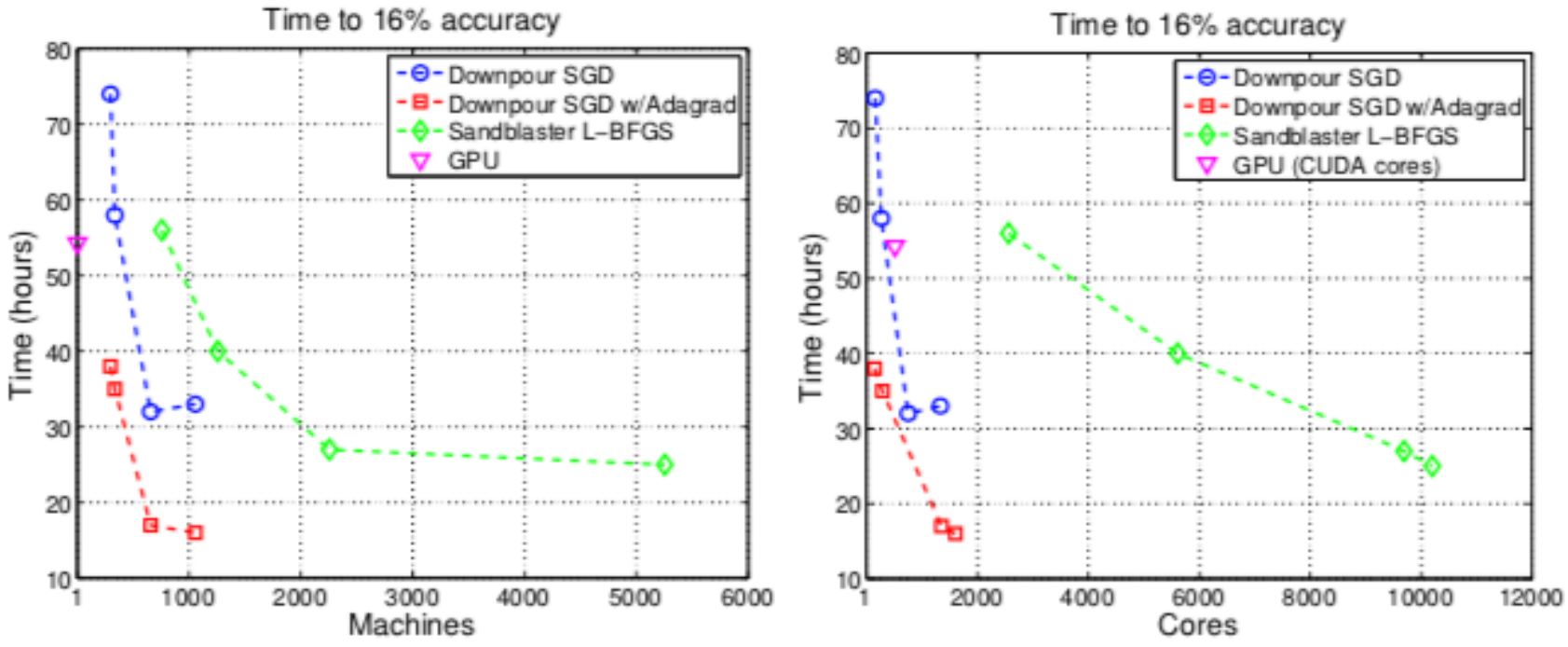


Figure 5: Time to reach a fixed accuracy (16%) for different optimization strategies as a function of number of the machines (left) and cores (right).

Source: "Large Scale Distributed Deep Networks" by Dean et al. (2015).

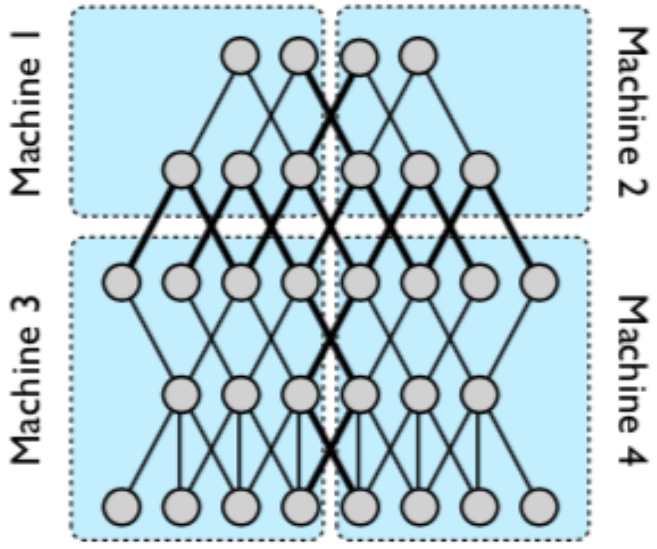


Figure 1: An example of model parallelism in DistBelief. A five layer deep neural network with local connectivity is shown here, partitioned across four machines (blue rectangles). Only those nodes with edges that cross partition boundaries (thick lines) will need to have their state transmitted between machines. Even in cases where a node has multiple edges crossing a partition boundary, its state is only sent to the machine on the other side of that boundary once. Within each partition, computation for individual nodes will the parallelized across all available CPU cores.

Source: "Large Scale Distributed Deep Networks" by Dean et al. (2015).

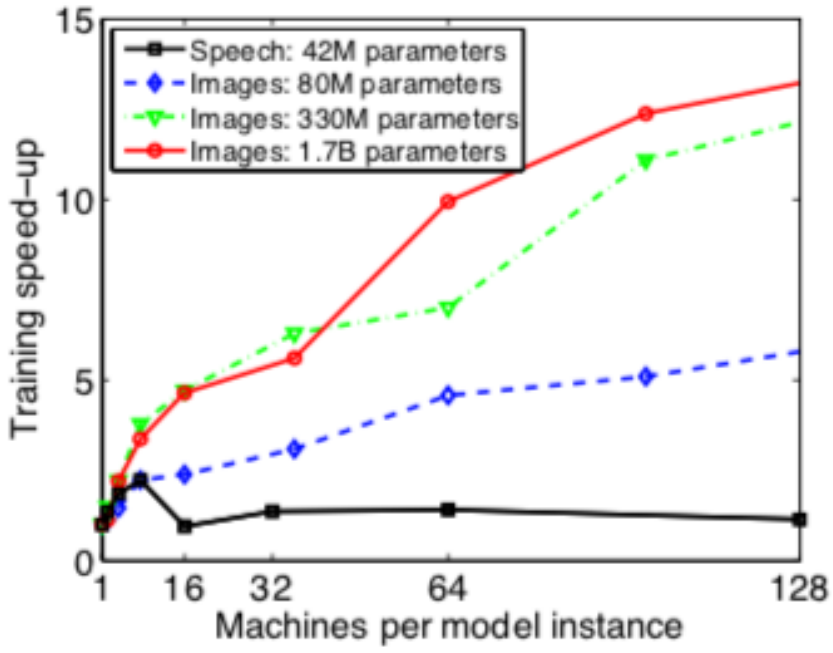


Figure 3: Training speed-up for four different deep networks as a function of machines allocated to a single DistBelief model instance. Models with more parameters benefit more from the use of additional machines than do models with fewer parameters.

Source: "Large Scale Distributed Deep Networks" by Dean et al. (2015).

Reinforcement learning.

- State of the system X_t.
- Action A_t.
- Reward R_t
- $X_{t+1} = h(X_t, A_t) + \epsilon_X$ (unknown transition function h, possibly random)
- $R_t = f(X_t, A_t) + \epsilon_R$ (unknown reward function f, possibly random)

- At time t, we would like to select an action $A_t = a$ that maximizes

$$\mathbb{E}\left[\sum_{\tau=t}^{T} R_\tau \middle| X_{t' \leq t}, A_{t' \leq t}, R_{t' < t}\right]. \tag{5}$$

- If we have an infinite time horizon, we would like to select an action $A_t = a$ that maximizes

$$\mathbb{E}\left[\sum_{\tau=t}^{\infty} \gamma^{t-\tau} R_\tau \middle| X_{t' \leq t}, A_{t' \leq t}, R_{t' < t}\right], \tag{6}$$

  where $0 < \gamma < 1$.

- This actually requires planning ahead regarding $A_{t+1}, A_{t+2}, \ldots$!

Note that the system is Markov! Therefore, it is sufficient to select an action $A_t = a$ that maximizes

$$\mathbb{E}\left[\sum_{\tau=t}^{T} R_\tau \middle| X_t, A_t = a\right]. \tag{7}$$

$$\mathbb{E}\left[\sum_{\tau=t}^{\infty} \gamma^{t-\tau} R_\tau \middle| X_t, A_t = a\right], \tag{8}$$

where $0 < \gamma < 1$. In particular, this means we would like to learn a strategy $A_t = g(X_t)$.

- Selecting the optimal action A_t requires planning ahead regarding $A_{t+1}, A_{t+2}, \ldots$.
- Learning a strategy $A_t = g(X_t)$ automatically does this.
- Need to learn the function g given observations of (X_t, A_t, R_t, X_{t+1}).

Let's consider the simplified case of $T = 0$, that is:

$$\mathbb{E}\Big[R\Big|X, A\Big], \tag{9}$$

where $R = f(X, A)$ (unknown, deterministic reward).

- Let $A \in \mathcal{A} = \{a_1, a_2, \ldots, a_K\}$ and $X \in \mathbb{R}^d$.
- Let $p(a, x; \theta)$ be our model for the action A.
- For example $p(a, x; \theta)$ is a neural network with its final layer being a softmax function.
- The objective function becomes

$$\max_{\theta} \mathbb{E}_{A,X}\Big[f(X, A)\Big], \tag{10}$$

where $\mathbb{P}[A = a|X] = p(a, X; \theta)$.

$$\begin{aligned}
\max_{\theta} \mathbb{E}_{A,X}\Big[f(X,A)\Big] &= \max_{\theta} \mathbb{E}\Big[\mathbb{E}\big[f(X,A)\big|X\big]\Big] \\
&= \max_{\theta} \mathbb{E}\Big[\sum_{a\in\mathcal{A}} f(X,a)\mathbb{P}[A=a|X]\Big] \\
&= \max_{\theta} \mathbb{E}\Big[\sum_{a\in\mathcal{A}} f(X,a)p(a,X;\theta)\Big]. \quad (11)
\end{aligned}$$

How can we numerically optimize this objective function?

$$\nabla_{\theta}\mathbb{E}_{A,X}\Big[f(X,A)\Big] = \mathbb{E}\Big[\sum_{a\in\mathcal{A}} f(X,a)\nabla_{\theta}p(a,X;\theta)\Big] \quad (12)$$

$$\nabla_\theta \mathbb{E}_{A,X}\Big[f(X,A)\Big] = \mathbb{E}\Big[\sum_{a\in\mathcal{A}} f(X,a)\nabla_\theta p(a,X;\theta)\Big] \tag{13}$$

- Requires knowledge of $f(X,a)_{a\in\mathcal{A}}$, i.e. repeatedly taking the different actions for the same state X and observing the rewards.
- Computationally costly if K is large.
- More importantly, this information may not be available (for example, in online learning).
- In typical settings, we do not have complete control over the sampling of X.

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{A,X}\Big[f(X,A)\Big] &= \mathbb{E}\Big[\sum_{a\in\mathcal{A}} f(X,a)\nabla_\theta p(a,X;\theta)\Big] \\
&= \mathbb{E}\Big[\sum_{a\in\mathcal{A}} f(X,a)\frac{\nabla_\theta p(a,X;\theta)}{p(a,X;\theta)}p(a,X;\theta)\Big] \\
&= \mathbb{E}_X\Big[\mathbb{E}_{A|X\sim p(a,X;\theta)}\big[f(X,A)\frac{\nabla_\theta p(A,X;\theta)}{p(A,X;\theta)}|X\big]\Big]. \\
&= \mathbb{E}_X\Big[\mathbb{E}_{A|X\sim p(a,X;\theta)}\big[R\frac{\nabla_\theta p(A,X;\theta)}{p(A,X;\theta)}|X\big]\Big]. \quad (14)
\end{aligned}
$$

$$\nabla_\theta \mathbb{E}_{A,X}\Big[f(X,A)\Big] = \mathbb{E}_X\Big[\mathbb{E}_{A|X\sim p(a,X;\theta)}\big[R\frac{\nabla_\theta p(A,X;\theta)}{p(A,X;\theta)}|X\big]\Big]. \quad (15)$$

Therefore, an unbiased estimate of $\nabla_\theta \mathbb{E}_{A,X}\Big[f(X,A)\Big]$ is

- Sample an X
- Sample an $A \sim p(a,X;\theta)$.
- Observe reward R.
- Calculate

$$g = R\frac{\nabla_\theta p(A,X;\theta)}{p(A,X;\theta)} \quad (16)$$

Stochastic gradient descent:

- Sample an X
- Sample an $A \sim p(a, X; \theta)$.
- Observe reward R.
- Calculate

$$g = R \frac{\nabla_\theta p(A, X; \theta)}{p(A, X; \theta)} \tag{17}$$

- Update parameters θ with

$$\theta = \theta + \alpha g. \tag{18}$$

Suppose that $\mathcal{A} = \mathbb{R}$, i.e. we have a continuous action space. Then,

- $p(a, x; \theta)$ is our model for the action A. Given the input x, $p(a, x; \theta)$ is a probability density function over a.
- **Mixture of Gaussians** model:

$$p(a, x; \theta) = \sum_{i=1}^{M} c_i \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(- \frac{(a - \mu_i(x; \theta_i))^2}{2\sigma^2} \right), \quad (19)$$

where $(c_1, \ldots, c_M)$ is a convex combination, i.e.

$$\begin{aligned} c_m &\geq 0, \\ \sum_{m=1}^{M} c_m &= 1. \end{aligned} \quad (20)$$

More generally,

$$p(a, x; \theta, \nu) = \sum_{i=1}^{M} c_i(x; \nu) \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(- \frac{(a - \mu_i(x; \theta_i))^2}{2\sigma^2} \right), \quad (21)$$

where we would like to learn both parameters (θ, ν).

- Let $c_i(x; \nu)$ be the i-th output of a neural network $c(x; \nu) : \mathbb{R}^d \to M$, where the final layer is a softmax layer.
- Note that a probability distribution is a convex combination!

Our objective function becomes

$$\begin{aligned}
\max_{\theta} \mathbb{E}_{A,X}\Big[f(X,A)\Big] &= \max_{\theta} \mathbb{E}\Big[\mathbb{E}\big[f(X,A)\big|X\big]\Big] \\
&= \max_{\theta} \mathbb{E}\Big[\int_a f(X,a)p(A \in da|X)\Big] \\
&= \max_{\theta} \mathbb{E}\Big[\int_a f(X,a)p(a,X;\theta)da\Big]. \quad (22)
\end{aligned}$$

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{A,X}\Big[f(X,A)\Big] &= \mathbb{E}\Big[\int_a f(X,a)\nabla_\theta p(a,X;\theta)da\Big] \\
&= \mathbb{E}\Big[\int_a f(X,a)\frac{\nabla_\theta p(a,X;\theta)}{p(a,X;\theta)}p(a,X;\theta)da\Big] \\
&= \mathbb{E}_X\Big[\mathbb{E}_{A|X\sim p(a,X;\theta)}\big[f(X,A)\frac{\nabla_\theta p(A,X;\theta)}{p(A,X;\theta)}|X\big]\Big]. \\
&= \mathbb{E}_X\Big[\mathbb{E}_{A|X\sim p(a,X;\theta)}\big[R\frac{\nabla_\theta p(A,X;\theta)}{p(A,X;\theta)}|X\big]\Big]. \quad (23)
\end{aligned}
$$

Stochastic gradient descent:

- Sample an X
- Sample an $A \sim p(a, X; \theta)$.
- Observe reward R.
- Calculate

$$g = R \frac{\nabla_\theta p(A, X; \theta)}{p(A, X; \theta)} \tag{24}$$

- Update parameters θ with

$$\theta = \theta + \alpha g. \tag{25}$$

Suppose that reward $R = f(X, A) + \epsilon$ where $\mathbb{E}[\epsilon|X, A] = 0$. The objective function becomes

$$\max_{\theta} \mathbb{E}_{A,X}\Big[R\Big], \tag{26}$$

which, by iterated expectations, is simply

$$\max_{\theta} \mathbb{E}_{A,X}\Big[f(X, A)\Big]. \tag{27}$$

Therefore,

$$\begin{aligned} \nabla_\theta \mathbb{E}_{A,X}\Big[f(X, A)\Big] &= \mathbb{E}\Big[\sum_a f(X, a)\nabla_\theta p(a, X; \theta)\Big] \\ &= \cdots \\ &= \mathbb{E}_X\Big[\mathbb{E}_{A|X\sim p(a,X;\theta)}\big[R\frac{\nabla_\theta p(A, X; \theta)}{p(A, X; \theta)}|X\big]\Big]. \end{aligned} \tag{28}$$

Therefore, an unbiased estimate of $\nabla_\theta \mathbb{E}_{A,X}\Big[f(X,A)\Big]$ is

- Sample an X
- Sample an $A \sim p(a, X; \theta)$.
- Observe reward R (which equals $f(X,A) + \epsilon$).
- Calculate

$$g = R\frac{\nabla_\theta p(A, X; \theta)}{p(A, X; \theta)} \tag{29}$$

For $t = 0, 1, \ldots, T$:

- Observe X_t.
- Select $A_t \sim p(a, X_t; \theta)$.
- Observe R_t

Take a stochastic gradient descent step:

- Set $G = 0$.
- For $t = 0, 1, \ldots, T$:

$$\begin{aligned} G &= G + \tilde{R}_t \frac{\nabla_\theta p(A_t, X_t; \theta)}{p(A_t, X_t; \theta)}, \\ \tilde{R}_t &= \sum_{\tau=t}^{T} R_\tau. \end{aligned} \tag{30}$$

- Update θ via

$$\theta = \theta + \alpha G. \tag{31}$$

For $t = 0, 1, \ldots, T$:

- Observe X_t.
- Select $A_t \sim p(a, X_t; \theta)$.
- Observe R_t

Take a stochastic gradient descent step:

- Set $G = 0$.
- For $t = 0, 1, \ldots, T$:

$$\begin{aligned} G &= G + \tilde{R}_t \frac{\nabla_\theta p(A_t, X_t; \theta)}{p(A_t, X_t; \theta)}, \\ \tilde{R}_t &= \sum_{\tau=t}^{T} \gamma^{\tau - t} R_\tau. \end{aligned} \tag{32}$$

- Update θ via

$$\theta = \theta + \alpha G. \tag{33}$$

Hamilton-Jacobi Bellman (HJB) equation.

Let the value function satisfy

$$V(x,a) = \mathbb{E}\left[R(X_t, A_t) + \sum_{\tau=t+1}^{\infty} \gamma^{\tau-t} R(X_\tau, A_\tau^*) \middle| X_t = x, A_t = a \right], \quad (34)$$

where $A_\tau^* = g(X_t)$ is the strategy which maximizes

$$\mathbb{E}\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(X_\tau, A_\tau^*) \middle| X_t = x \right]. \quad (35)$$

The value at state x is

$$V(x) = \max_{a \in \mathcal{A}} V(x,a). \quad (36)$$

The optimal strategy at state x is

$$g(x) = \arg\max_{a \in \mathcal{A}} V(x,a). \quad (37)$$

$R(x, a)$ is the (possibly random) reward given state x and action a.

Define

$$r(x, a) = \mathbb{E}\left[R(X_t, A_t)\middle| X_t = x, A_t = a\right]. \tag{38}$$

Then,

$$\begin{aligned} V(x,a) &= r(x,a) + \mathbb{E}\left[\sum_{\tau=t+1}^{\infty} \gamma^{t-\tau} R(X_\tau, A_\tau^*) \middle| X_t = x, A_t = a \right] \\ &= r(x,a) + \gamma \mathbb{E}\left[\sum_{\tau=t+1}^{\infty} \gamma^{t-\tau-1} R(X_\tau, A_\tau^*) \middle| X_t = x, A_t = a \right] \\ &= r(x,a) + \gamma \mathbb{E}\left[\max_{a'} V(X_{t+1}, a') \middle| X_t = x, A_t = a \right]. \quad (39) \end{aligned}$$

The HJB equation is

$$r(x,a) + \gamma \mathbb{E}_{X_{t+1}}\left[\max_{a'} V(X_{t+1}, a') \middle| X_t = x, A_t = a \right] - V(x,a) = 0, \quad (40)$$

for every $(x,a) \in \mathcal{X} \times \mathcal{A}$.

A more detailed derivation:

$$V(x,a) = \mathbb{E}\left[R(X_t, A_t) + \sum_{\tau=t+1}^{\infty} \gamma^{\tau-t} R(X_\tau, A_\tau^*) \middle| X_t = x, A_t = a \right]. \quad (41)$$

We also know that

$$\begin{aligned} & \mathbb{E}\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(X_\tau, A_\tau^*) \middle| X_t = x \right] \\ = \quad & \mathbb{E}\left[R(X_t, A_t) + \sum_{\tau=t+1}^{\infty} \gamma^{\tau-t} R(X_\tau, A_\tau^*) \middle| X_t = x, A_t = A_t^* \right] \\ = \quad & \max_{a \in \mathcal{A}} V(x,a). \end{aligned} \quad (42)$$

$$\mathbb{E}\Big[\sum_{\tau=t+1}^{\infty}\gamma^{t-\tau-1}R(X_\tau,A_\tau^*)\Big|X_t=x,A_t=a\Big]$$

$$= \mathbb{E}\Big[\mathbb{E}\big[\sum_{\tau=t+1}^{\infty}\gamma^{t-\tau-1}R(X_\tau,A_\tau^*)\big|X_{t+1}\big]\Big|X_t=x,A_t=a\Big] \quad (43)$$

Since (X_t, A_t^*) is a Markov chain,

$$\mathbb{E}\big[\sum_{\tau=t+1}^{\infty}\gamma^{t-\tau-1}R(X_\tau,A_\tau^*)\big|X_{t+1}=x\big]$$

$$= \mathbb{E}\big[\sum_{\tau=t}^{\infty}\gamma^{t-\tau}R(X_\tau,A_\tau^*)\big|X_t=x\big]$$

$$= \max_{a\in\mathcal{A}} V(x,a). \quad (44)$$

Therefore,

$$\begin{aligned} & \mathbb{E}\left[\sum_{\tau=t+1}^{\infty} \gamma^{t-\tau-1} R(X_\tau, A_\tau^*) \middle| X_t = x, A_t = a\right] \\ = \quad & \mathbb{E}\left[\max_{a \in \mathcal{A}} V(X_{t+1}, a) \middle| X_t = x, A_t = a\right]. \end{aligned} \tag{45}$$

This yields the HJB equation

$$r(x,a) + \gamma \mathbb{E}_{X_{t+1}}\left[\max_{a'} V(X_{t+1}, a') \middle| X_t = x, A_t = a\right] - V(x,a) = 0, \tag{46}$$

for every $(x, a) \in \mathcal{X} \times \mathcal{A}$.

Practical challenges:

- State space can be high-dimensional.
- Action space can be high-dimensional.

Dynamic programming versus reinforcement learning (RL):

- In RL, the transition probability

$$p(x'|x,a) = \mathbb{P}[X_{t+1} = x'|X_t = x, A_t = a] \tag{47}$$

  is **unknown**.

Function approximation.

$$J(x,a;\theta) = \left(r(x,a) + \gamma \mathbb{E}_{X_{t+1}} \left[\max_{a'} Q(X_{t+1}, a'; \theta) \middle| X_t = x, A_t = a \right] - Q(x,a;\theta) \right)^2. \quad (48)$$

Minimize

$$\mathcal{L}(\theta) = \sum_{x,a} J(x,a;\theta). \quad (49)$$

If $\mathcal{L}(\theta) = 0$, $V(x,a) = Q(x,a;\theta)$!

Q-learning.

- Select A_t.
- Sample $\left(X_{t+1}, R(X_t, A_t)\right)$ given X_{t-1}, A_{t-1}.
- Define

$$\tilde{J}(\theta) = \left(R(X_t, A_t) + \gamma \max_{a'} Q(X_{t+1}, a'; \theta) - Q(X_t, A_t; \theta) \right)^2 . \quad (50)$$

- Take a stochastic gradient descent step:

$$\theta = \theta - \alpha \nabla_\theta \tilde{J}(\theta). \quad (51)$$

How to select action A_t from $\mathcal{A} = \{a_1, a_2, \ldots, a_K\}$?

- Greedy action: $A_t = \arg\max_{a \in \mathcal{A}} Q(X_t, a'; \theta)$.
- Pure exploration: $A_t \sim \text{Uniform}(a_1, a_2, \ldots, a_K)$.
- ϵ-Greedy algorithm:

$$A_t = \begin{cases} \text{Uniform}(a_1, a_2, \ldots, a_K) & \text{with prob. } \epsilon \\ \arg\max_{a \in \mathcal{A}} Q(X_t, a'; \theta) & \text{with prob. } 1 - \epsilon. \end{cases}$$

ϵ-Greedy algorithm typically has an $\epsilon_m \to 0$ as the number of training epochs m increases:

$$A_t = \begin{cases} \text{Uniform}(a_1, a_2, \ldots, a_K) & \text{with prob. } \epsilon_m \\ \arg\max_{a \in \mathcal{A}} Q(X_t, a'; \theta) & \text{with prob. } 1 - \epsilon_m. \end{cases}$$

As the model $Q(x, a; \theta)$ becomes more accurate, we would like to more frequently take the greedy action.

$$\begin{aligned} \tilde{J}(\theta) &= \Big(Y_t - Q(X_t, A_t; \theta) \Big)^2, \\ Y_t &= R(X_t, A_t) + \gamma \max_{a'} Q(X_{t+1}, a'; \theta) \end{aligned} \tag{52}$$

In SGD, treat Y_t as a constant. That is,

$$\theta = \theta + \alpha \Big(Y_t - Q(X_t, A_t; \theta) \Big) \nabla_\theta Q(X_t, A_t; \theta). \tag{53}$$

In summary, the Q-learning algorithm is:

- Select A_t according to the ϵ-greedy algorithm.
- Observe $\big(X_{t+1}, R(X_t, A_t)\big)$.
- Let

$$Y_t \quad = \quad R(X_t, A_t) + \gamma \max_{a'} Q(X_{t+1}, a'; \theta). \tag{54}$$

- Take a stochastic gradient descent step:

$$\theta = \theta + \alpha \bigg(Y_t - Q(X_t, A_t; \theta) \bigg) \nabla_\theta Q(X_t, A_t; \theta). \tag{55}$$

Convergence.

The loss function is

$$\mathcal{L}(\theta) = \mathbb{E}_{R,X_t,A_t,X_{t+1}} \left(R(X_t, A_t) + \gamma \max_{a'} Q(X_{t+1}, a'; \theta) - Q(X_t, A_t; \theta) \right)^2.$$

Note that

$$\begin{aligned} & \mathbb{E}_{R,X_t,A_t,X_{t+1}} \left[\left(R(X_t, A_t) + \gamma \max_{a'} Q(X_{t+1}, a'; \theta) - Q(X_t, A_t; \theta) \right) \right. \\ \times \quad & \left. \nabla_\theta Q(X_t, A_t; \theta). \right] \end{aligned} \tag{57}$$

is not a descent direction for $\mathcal{L}(\theta)$.

In addition,

$$\nabla_\theta \mathbb{E}_{R,X_t,A_t,X_{t+1}} \left[\left(R(X_t, A_t) + \gamma \max_{a'} Q(X_{t+1}, a'; \theta) - Q(X_t, A_t; \theta) \right) \right]$$

is not Lipschitz continuous.

In tabular Q-learning, i.e. $\mathcal{X}$ is a discrete space and the function to be learned is $Q(x, a)$, we can show convergence for the scheme

$$Q(X_t, A_t) = Q(X_t, A_t) + \left(R(X_t, A_t) + \gamma \max_{a'} Q(X_{t+1}, a') - Q(X_t, A_t) \right).$$

For serious applications, tabular Q-learning is impractical due to the curse of dimensionality.

If A_t visits each action $a \in \mathcal{A}$ infinitely often,

$$Q(x, a) \to V(x, a), \tag{59}$$

as $t \to \infty$.

Results for function approximation.

- Let $A_t = p(a|X_t)$ (i.e., an *a priori* fixed policy for the actions A_t).

- Estimate value function $Q(x)$ with $Q(x;\theta) = \theta^\top \phi(x)$ with updates

$$\begin{aligned} \theta^{t+1} &= \theta^t + \alpha\Big(Y_t - Q(X_t;\theta^t)\Big)\nabla_\theta Q(X_t;\theta^t), \\ Y_t &= R(X_t, A_t) + \gamma Q(X_{t+1};\theta^t). \end{aligned} \tag{60}$$

  Of course, $\nabla_\theta Q(X_t;\theta) = \phi(X_t)$.

- Then, θ^t converges as $t \to \infty$.

- See Tsitsiklis and Van Roy (1997).

- If the control A_t is continuous, i.e. $\mathcal{A} = \mathbb{R}^K$, then Q-learning is not computationally feasible.

- The optimization problem

$$\arg\max_{a \in \mathcal{A}} Q(x, a; \theta) \tag{61}$$

  would need to be repeatedly solved.

- This requires optimizing over a non-convex neural network!

Actor-critic.

- Actor: a network $p(x;\theta^A) : \mathbb{R}^d \to \mathbb{R}^K$ gives the action (**continuous control**).
- Critic: a network $Q(a, x;\theta^Q)$ gives the value of the state-action pair (a, x).
- The critic minimizes the objective function

$$\begin{aligned} J(\theta^Q) &= \Big(Y_t - Q(X_t, A_t;\theta^Q)\Big)^2, \\ Y_t &= R(X_t, A_t) + \gamma Q(X_{t+1}, p(X_{t+1};\theta^A);\theta^Q). \end{aligned} \quad (62)$$

  Y_t is treated as a constant.
- The actor maximizes the objective function

$$G(\theta^A) = Q(X_t, p(X_t;\theta^A);\theta^Q). \quad (63)$$

The SGD algorithm is:

- Select action $A_t = p(X_t; \theta^A) + \text{Noise}$.
- Observe reward $R(X_t, A_t)$ and X_{t+1}.
- Update critic.
- Update actor.

Replay buffer.

- Strong correlations between X_t and X_{t+1} can slow convergence.
- Store a subset of observations over all previous times.
- At every parameter update, sample a mini-batch from the replay buffer, and calculate gradient on this mini-batch.

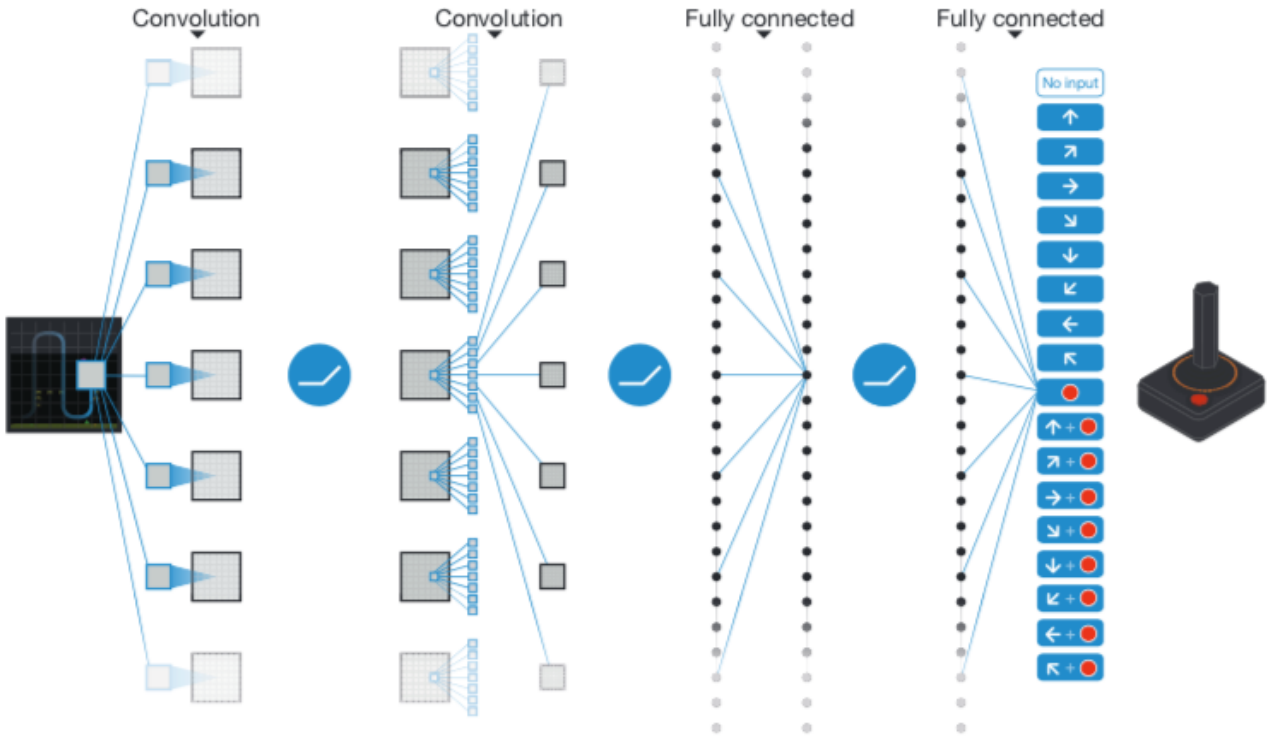


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0,x)$).

Source: "Human-level control through deep reinforcement learning" by Mnih et al. (2015).

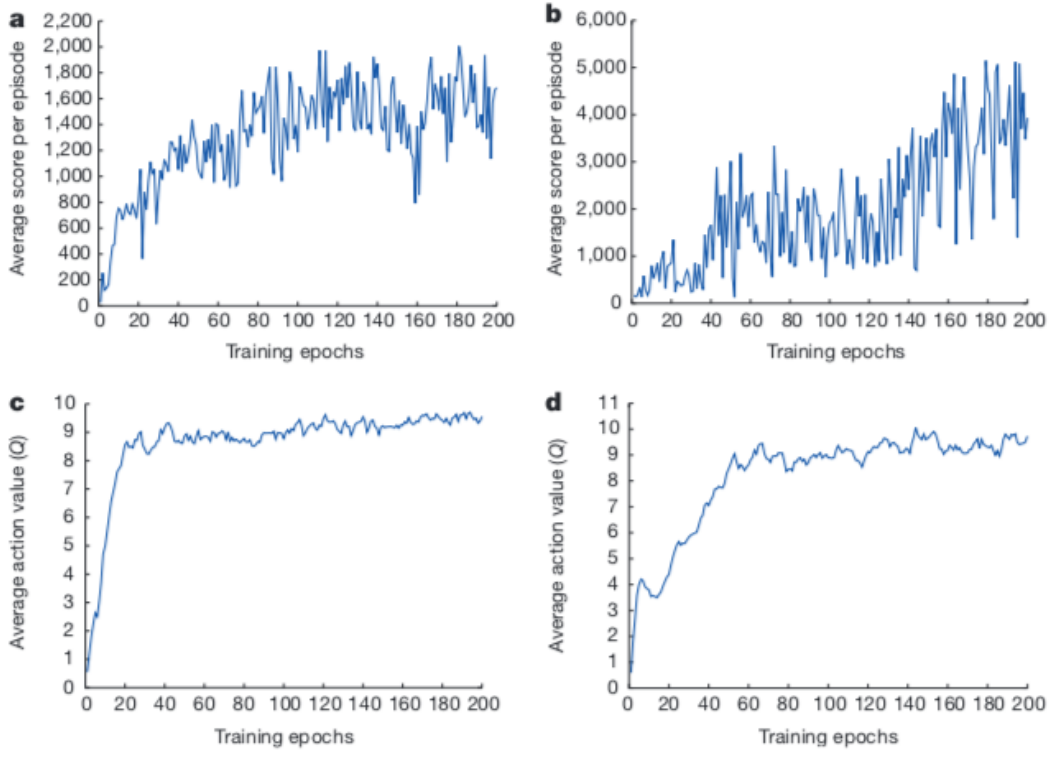


Figure 2 | Training curves tracking the agent's average score and average predicted action-value. **a**, Each point is the average score achieved per episode after the agent is run with ε-greedy policy ($\varepsilon = 0.05$) for 520 k frames on Space Invaders. **b**, Average score achieved per episode for Seaquest. **c**, Average predicted action-value on a held-out set of states on Space Invaders. Each point on the curve is the average of the action-value Q computed over the held-out set of states. Note that Q-values are scaled due to clipping of rewards (see Methods). **d**, Average predicted action-value on Seaquest. See Supplementary Discussion for details.

Source: "Human-level control through deep reinforcement learning" by Mnih et al. (2015).

Extended Data Table 3 | The effects of replay and separating the target Q-network

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

DQN agents were trained for 10 million frames using standard hyperparameters for all possible combinations of turning replay on or off, using or not using a separate target Q-network, and three different learning rates. Each agent was evaluated every 250,000 training frames for 135,000 validation frames and the highest average episode score is reported. Note that these evaluation episodes were not truncated at 5 min leading to higher scores on Enduro than the ones reported in Extended Data Table 2. Note also that the number of training frames was shorter (10 million frames) as compared to the main results presented in Extended Data Table 2 (50 million frames).

Source: "Human-level control through deep reinforcement learning" by Mnih et al. (2015).

Extended Data Table 4 | Comparison of DQN performance with linear function approximator

Game	DQN	Linear
Breakout	316.8	3.00
Enduro	1006.3	62.0
River Raid	7446.6	2346.9
Seaquest	2894.4	656.9
Space Invaders	1088.9	301.3

Source: "Human-level control through deep reinforcement learning" by Mnih et al. (2015).

Extended Data Table 2 | Comparison of games scores obtained by DQN agents with methods from the literature[12,15] and a professional human games tester

Game	Random Play	Best Linear Learner	Contingency (SARSA)	Human	DQN (± std)	Normalized DQN (% Human)
Alien	227.8	939.2	103.2	6875	3069 (±1093)	42.7%
Amidar	5.8	103.4	183.6	1676	739.5 (±3024)	43.9%
Assault	222.4	628	537	1496	3359(±775)	246.2%
Asterix	210	987.3	1332	8503	6012 (±1744)	70.0%
Asteroids	719.1	907.3	89	13157	1629 (±542)	7.3%
Atlantis	12850	62687	852.9	29028	85641(±17600)	449.9%
Bank Heist	14.2	190.8	67.4	734.4	429.7 (±650)	57.7%
Battle Zone	2360	15820	16.2	37800	26300 (±7725)	67.6%
Beam Rider	363.9	929.4	1743	5775	6846 (±1619)	119.8%
Bowling	23.1	43.9	36.4	154.8	42.4 (±88)	14.7%
Boxing	0.1	44	9.8	4.3	71.8 (±8.4)	1707.9%
Breakout	1.7	5.2	6.1	31.8	401.2 (±26.9)	1327.2%
Centipede	2091	8803	4647	11963	8309(±5237)	63.0%
Chopper Command	811	1582	16.9	9882	6687 (±2916)	64.8%
Crazy Climber	10781	23411	149.8	35411	114103 (±22797)	419.5%
Demon Attack	152.1	520.5	0	3401	9711 (±2406)	294.2%
Double Dunk	-18.6	-13.1	-16	-15.5	-18.1 (±2.6)	17.1%
Enduro	0	129.1	159.4	309.6	301.8 (±24.6)	97.5%
Fishing Derby	-91.7	-89.5	-85.1	5.5	-0.8 (±19.0)	93.5%
Freeway	0	19.1	19.7	29.6	30.3 (±0.7)	102.4%
Frostbite	65.2	216.9	180.9	4335	328.3 (±250.5)	6.2%
Gopher	257.6	1288	2368	2321	8520 (±3279)	400.4%
Gravitar	173	387.7	429	2672	306.7 (±223.9)	5.3%
H.E.R.O.	1027	6459	7295	25763	19950 (±158)	76.5%
Ice Hockey	-11.2	-9.5	-3.2	0.9	-1.6 (±2.5)	79.3%
James Bond	29	202.8	354.1	406.7	576.7 (±175.5)	145.0%

Source: "Human-level control through deep reinforcement learning" by

Double Q-learning.

- Deep Reinforcement Learning with Double Q-Learning by Hasselt, Guez, and Silver (2016).
- Q-learning has empirically been observed to over-estimate the value of action-state pairs.
- Deep Q-learning algorithm:

$$\begin{aligned} \tilde{J}(\theta_t) &= \left(Y_t - Q(X_t, A_t; \theta_t) \right)^2, \\ Y_t &= R(X_t, A_t) + \gamma Q\left(X_{t+1}, \arg\max_a Q(X_{t+1}, a; \theta_t); \theta_t \right). \end{aligned} \tag{64}$$

In SGD, treat Y_t as a constant.

Double Q-learning:

- Modify deep Q-learning algorithm:

$$\begin{aligned} \tilde{J}(\theta_t) &= \Big(Y_t - Q(X_t, A_t; \theta_t)\Big)^2, \\ Y_t &= R(X_t, A_t) + \gamma Q\Big(X_{t+1}, \arg\max_a Q(X_{t+1}, a; \theta_t); \theta_t^-\Big). \end{aligned} \tag{65}$$

  In SGD, treat Y_t as a constant.

- θ_t^- is periodically updated. I.E., every τ parameter updates, set $\theta_t^- = \theta_t$.

- Intuition: $\arg\max_a Q(X_{t+1}, a; \theta_t^-)$ maximizes $Q\left(X_{t+1}, \cdot; \theta_t^-\right)$.
- We instead use $\arg\max_a Q(X_{t+1}, a; \theta_t)$.
- This can hopefully help address the issue of over-estimation.

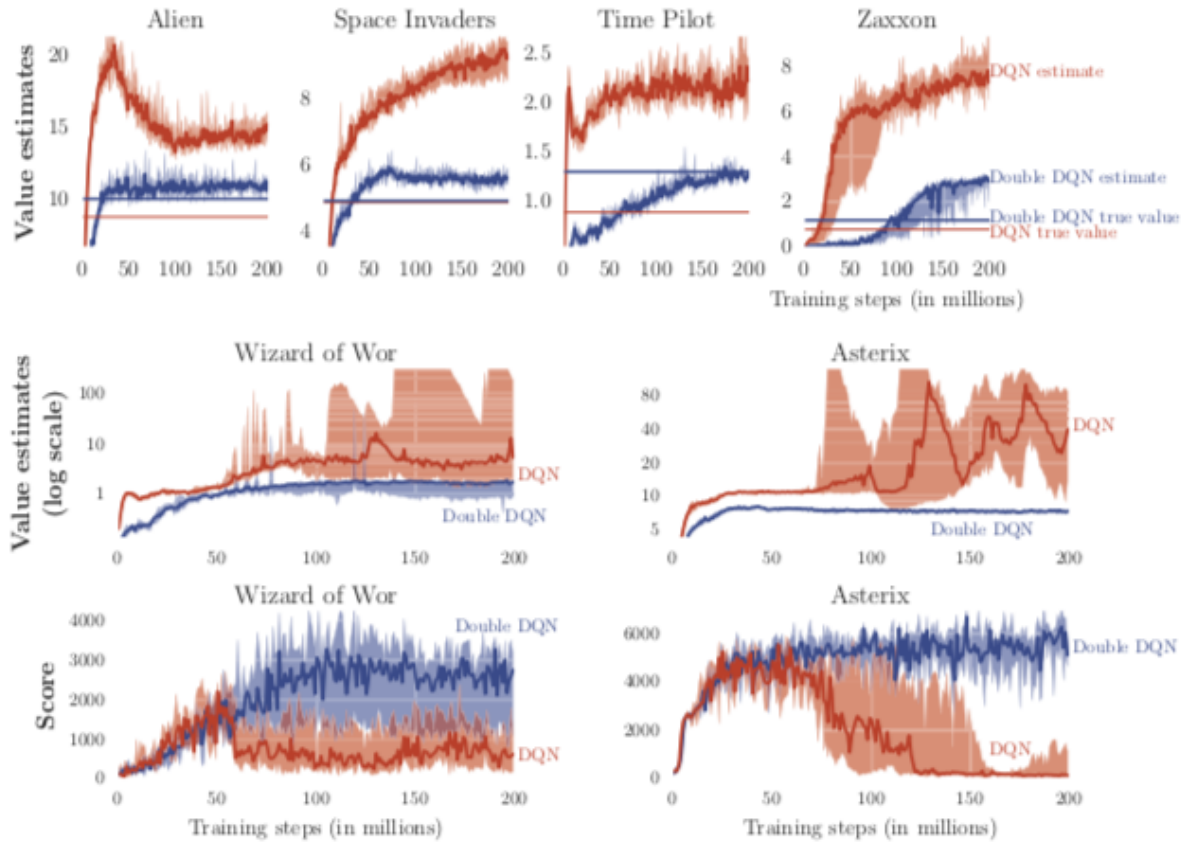
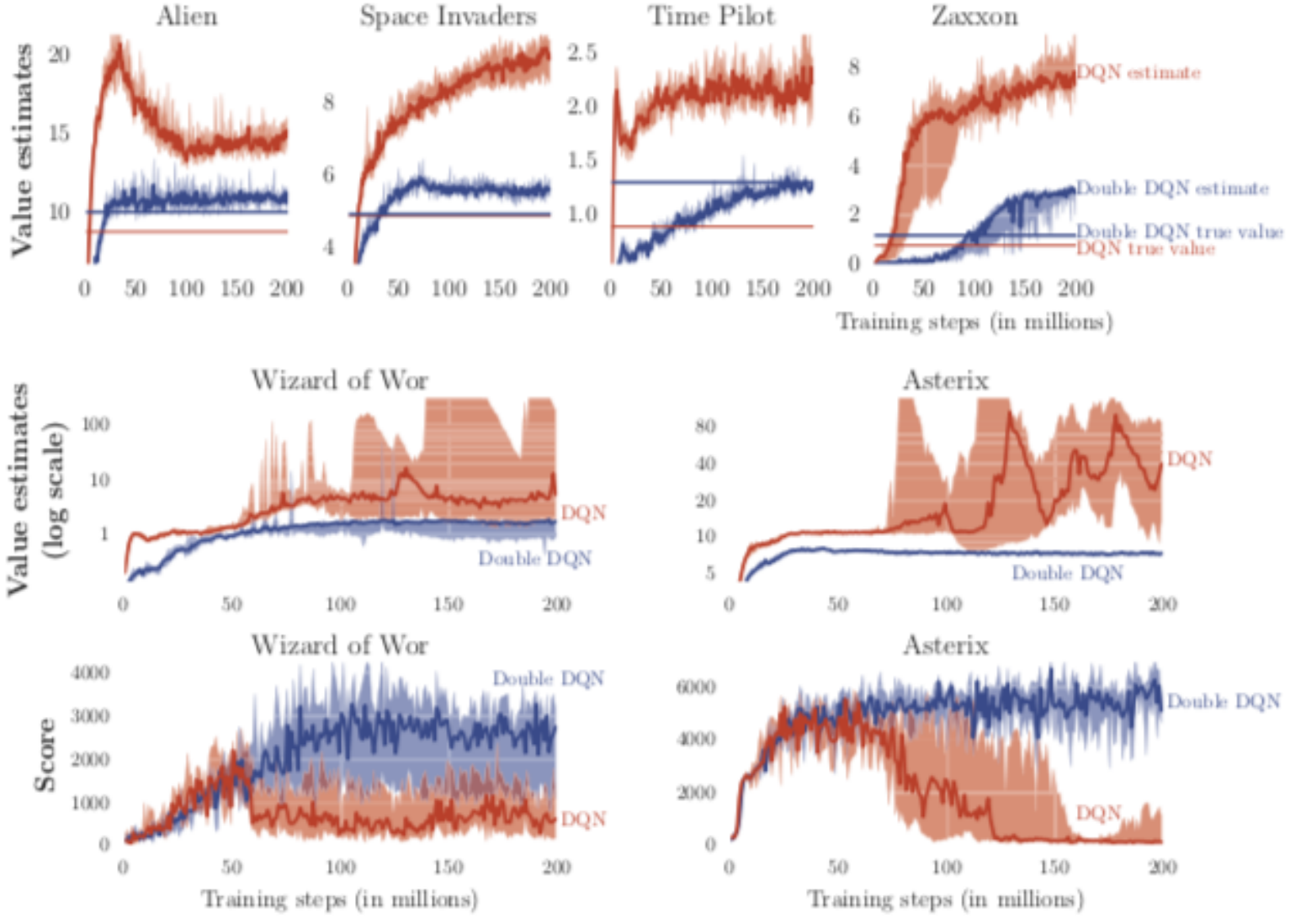
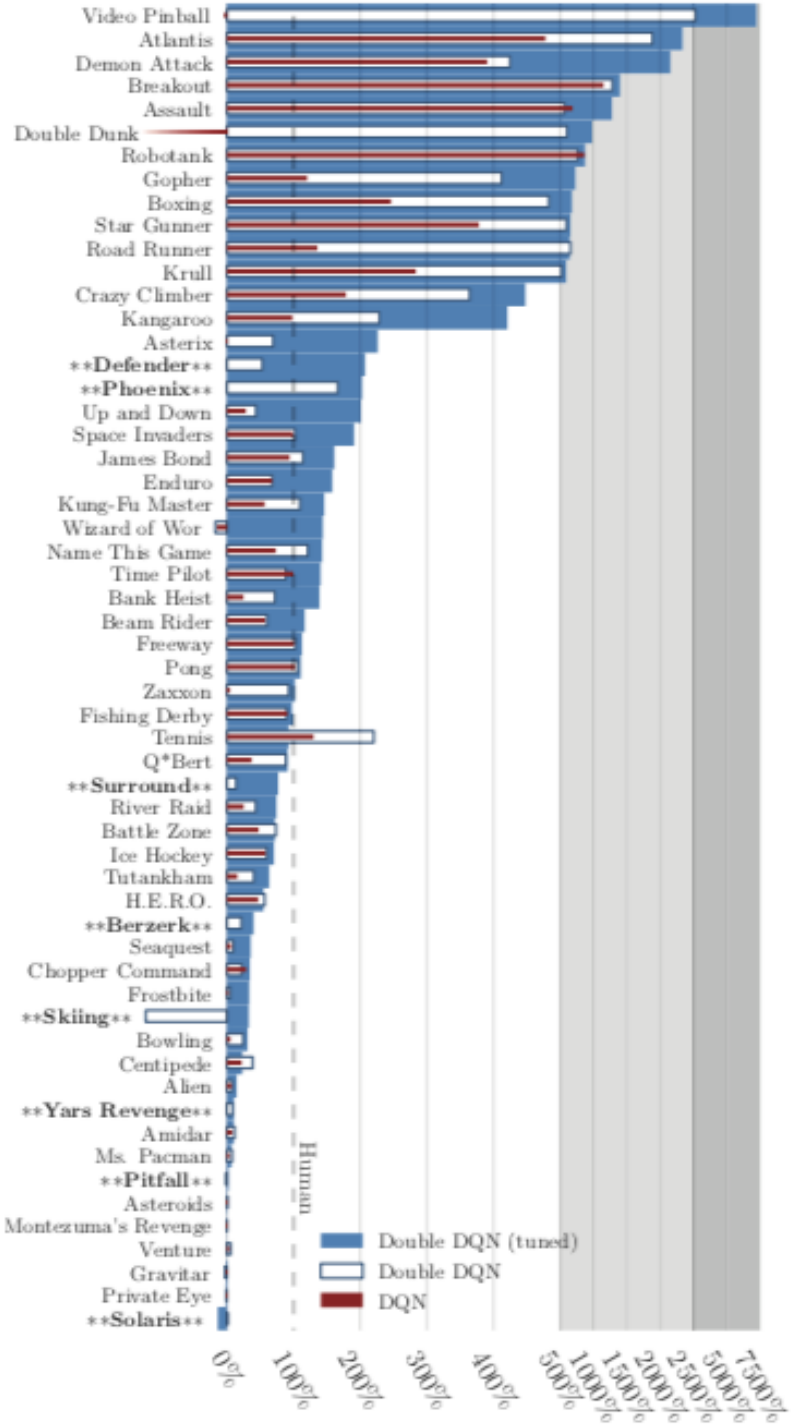


Figure 3: The **top** and **middle** rows show value estimates by DQN (orange) and Double DQN (blue) on six Atari games. The results are obtained by running DQN and Double DQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over seeds and we average the two extreme values to obtain the shaded area (i.e., 10% and 90% quantiles with linear interpolation). The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias. The **middle** row shows the value estimates (in log scale) for two games in which DQN's overoptimism is quite extreme. The **bottom** row shows the detrimental effect of this on the score achieved by the agent as it is evaluated during training: the scores drop when the overestimations begin. Learning with Double DQN is much more stable.

Source: "Deep Reinforcement Learning with Double Q-Learning" by



Source: "Deep Reinforcement Learning with Double Q-Learning" by Hasselt et al. (2016).


Video Pinball
Atlantis
Demon Attack
Breakout
Assault
Double Dunk
Robotank
Gopher
Boxing
Star Gunner
Road Runner
Krull
Crazy Climber
Kangaroo
Asterix
Defender
Phoenix
Up and Down
Space Invaders
James Bond
Enduro
Kung-Fu Master
Wizard of Wor
Name This Game
Time Pilot
Bank Heist
Beam Rider
Freeway
Pong
Zaxxon
Fishing Derby
Tennis
Q*Bert
Surround
River Raid
Battle Zone
Ice Hockey
Tutankham
H.E.R.O.
Berzerk
Seaquest
Chopper Command
Frostbite
Skiing
Bowling
Centipede
Alien
Yars Revenge
Amidar
Ms. Pacman
Pitfall
Asteroids
Montezuma's Revenge
Venture
Gravitar
Private Eye
Solaris
Human
Double DQN (tuned)
Double DQN
DQN
0%
100%
200%
300%
400%
500%
1000%
1500%
2000%
2500%
5000%
7500%

Topics we will cover next week:

- Actor-critic methods.
- Deep exploration.
- AlphaGo algorithm.

Layer normalization vs. batch normalization.

- Recall that batch normalization estimates normalization statistics (mean and variance) across a minibatch of data samples $1, 2, \ldots, M$.
- Layer normalization calculates normalization statistics (mean and variance) across the hidden units $1, 2, \ldots, d_H$ for a **single** data sample.
- Exact normalization versus approximate normalization.
- Easier implementation for evaluation/test set.
- Layer normalization be used for online predictions (or training) with mini-batch size 1.
- See "Layer Normalization" by Ba, Kiros, and Hinton (2016).

Standard layer (no normalization):

$$\begin{aligned} A^{\ell+1} &= W^{\ell+1} H^{\ell}, \\ H^{\ell+1} &= \sigma(A^{\ell+1} + b^{\ell+1}). \end{aligned} \tag{66}$$

Layer normalization:

$$\begin{aligned} A^{\ell+1} &= W^{\ell+1} H^{\ell}, \\ \bar{A}^{\ell+1} &= \gamma^{\ell+1} \odot \frac{A^{\ell+1} - \mu^{\ell+1}}{\beta^{\ell+1}}, \\ H^{\ell+1} &= \sigma(\bar{A}^{\ell+1} + b^{\ell+1}), \end{aligned} \tag{67}$$

where $\gamma^{\ell+1}$ is an additional parameter. The normalization statistics are:

$$\begin{aligned} \mu^{\ell+1} &= \frac{1}{d_H} \sum_{i=1}^{d_H} A_i^{\ell+1}, \\ \beta^{\ell+1} &= \sqrt{\frac{1}{d_H} \sum_{i=1}^{d_H} (A_i^{\ell+1} - \mu^{\ell+1})}. \end{aligned} \tag{68}$$

Midterm.

- **Everything covered in the lectures, course notes, and homeworks can be on the midterm.**
- 1 question where you will derive backpropagation algorithm for a neural network architecture (either a fully-connected network or a fully-connected network with dropout).
- 1 PyTorch question
- Conceptual questions regarding deep learning models, algorithms, and training methods.
- Questions involving quick calculations
- No proofs.

Backpropagation question will either be

- Multi-layer fully-connected network,
- Multi-layer fully-connected network with dropout, or

An example of a possible PyTorch question might be

- If $\mathrm{optimizer.zero\_grad()}$ is not included, what happens?
- What is the difference between PyTorch and TensorFlow?
- If GPU is "out of memory", how can we address this?
- How do we move the model from the CPU to the GPU? (Specific PyTorch command.)
- Why does PyTorch use Float32 instead of Float64 (double)?

- Answers should be mathematically precise as possible.
- For example, a possible question could be to mathematically state a GRU unit or a convolution layer (with stride, padding, input channels, output channels).
- Residual networks
- Generative adversarial networks
- Convolution networks
- Good references: lecture slides, course notes, book by Goodfellow et al., homework assignments.

- Different types of regularization
- Optimization algorithms (RMSprop, ADAM, SGD, etc.)
- Non-convexity of neural networks
- What does the universal approximation theorem say? How does it relate to SGD methods for estimating neural networks?
- Why use softmax function instead of $\arg\max$?

- Xavier initialization
- Batch normalization
- Layer normalization
- What is the vanishing gradient problem?
- What are ways to try to address the vanishing gradient problem?

- Data augmentation
- Transfer learning
- Distributed training (synchronous, asynchronous, data parallelization, model parallelization, efficiency)
- What is the advantage of GPU over CPU for machine learning applications?

- Fundamental functions in deep learning: types of hidden units, different types of objective functions, softmax functions, convolutions, etc.

- Training LSTMs and recurrent networks (backpropagation through time, truncated backpropagation through time).

- Why is the dropout algorithm (when used on the test set) *ad hoc*?

- Comparison of different optimization algorithms (RMSprop, ADAM, AdaGrad, SGD, etc.). What are the advantages of algorithm A over algorithm B?

Midterm at 8 AM on Tuesday, April 2 in Lecture.

- 6 questions, each question with multiple parts.
- If possible, provide mathematically precise answers. (For example, if the questions asks to describe the dropout algorithm, write the mathematical equations for the dropout algorithm.)
- 1 hour and 20 minutes.

Example Midterm Exam questions from 2016 and 2017.

1. Derive the backpropagation algorithm for a single-layer fully-connected neural network with dropout.

$$
\begin{aligned}
Z^1 &= W^1 X + b^1, \\
A_i &= R_i \sigma(Z_i^1), \\
Z^2 &= W^2 A + b^2, \\
\mathbb{P}[Y = m|X] &= \frac{e^{Z_m}}{\sum_{m'=0}^{1} e^{Z_{m'}}},
\end{aligned}
$$

where $X \in \mathbb{R}^d$, $W^1 \in \mathbb{R}^{L \times d}$, $b^1 \in \mathbb{R}^L$, $A \in \mathbb{R}^L$, $W^2 \in \mathbb{R}^{2 \times L}$, $b^2 \in \mathbb{R}^2$, $\sigma : \mathbb{R} \to \mathbb{R}$, $Z^2 \in \mathbb{R}^2$, and $Y \in \{0, 1\}$. The loss function is the cross-entropy error (i.e., the negative log-likelihood). The parameters are $\theta = \{W^1, W^2, b^1, b^2\}$. R_i is a Bernoulli random variable. Also, write the stochastic gradient descent algorithm for training this network.

AdaGrad.

$$\begin{aligned} r_i &\leftarrow r_i + g_i^2, \\ \Delta\theta_i &= -\frac{\epsilon}{\delta + \sqrt{r_i}} g_i, \\ \theta_i &\leftarrow \theta_i + \Delta\theta_i, \end{aligned}$$

where θ_i is the i-th parameter.

The gradient g is

$$g = \nabla_\theta \frac{1}{M} \sum_{m=1}^{M} \rho\big(f(x^m;\theta), y^m\big). \tag{69}$$

RMSprop.

$$
\begin{aligned}
r_i &\leftarrow \rho r_i + (1-\rho) g_i^2, \\
\Delta\theta_i &= -\frac{\epsilon}{\sqrt{\delta + r_i}} g_i, \\
\theta_i &\leftarrow \theta_i + \Delta\theta_i,
\end{aligned}
$$

where θ_i is the i-th parameter.

RMSprop (element-wise notation).

$$
\begin{aligned}
r &\leftarrow \rho r + (1-\rho) g \odot g, \\
\Delta\theta_i &= -\frac{\epsilon}{\sqrt{\delta + r}} \odot g, \\
\theta &\leftarrow \theta + \Delta\theta,
\end{aligned}
$$

where $\sqrt{\cdot}$ is applied element-wise.

ADAM (element-wise notation).

$$
\begin{aligned}
t &\leftarrow t+1, \\
s &\leftarrow \rho_1 s + (1-\rho_1) g, \\
r &\leftarrow \rho_2 r + (1-\rho_2) g \odot g, \\
\hat{s} &\leftarrow \frac{s}{1-\rho_1^t}, \\
\hat{r} &\leftarrow \frac{r}{1-\rho_2^t}, \\
\Delta\theta_i &= -\frac{\epsilon}{\delta + \sqrt{\hat{r}}} \odot \hat{s}, \\
\theta &\leftarrow \theta + \Delta\theta.
\end{aligned}
$$

Non-uniqueness of global minimum for a neural network.

As an example, consider a single layer neural network with 2 hidden units:

$$\begin{aligned} f(x;\theta) &= C_1\sigma(W_1 x) + C_2\sigma(W_2 x), \\ \mathcal{L}(\theta) &= \mathbb{E}_{X,Y}\Big[\rho\big(f(x;\theta), y\big)\Big]. \end{aligned} \tag{70}$$

Let $\theta = (C_1, W_1, C_2, W_2)$. Suppose that a global minimum is

$$\theta^{*,\mathrm{A}} = (C_1^*, W_1^*, C_2^*, W_2^*). \tag{71}$$

Then, there is of course always another global minimum

$$\theta^{*,\mathrm{B}} = (C_2^*, W_2^*, C_1^*, W_1^*). \tag{72}$$

That is, the global minimum is not unique.