# Non-linear Value Function Approximation: Deep Q-Network

Alina Vereshchaka

CSE4/510 Reinforcement Learning
Fall 2019

*avereshc@buffalo.edu*

October 3, 2019

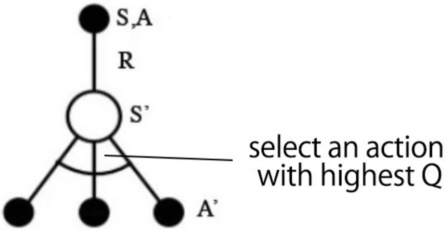
*Slides are based on David Silver's Deep Learning Tutorial, ICML 2016
& Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning."

# Overview

# Table of Contents

# Recap: Q-Learning Algorithm

- Q-learning learns the action-value function $Q(s, a)$: how good to take an action at a particular state.
- From the memory table, we determine the next action $a'$ to take which has the maximum $Q(s', a')$.



Loop for each step of episode:

Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)

Take action $A$, observe $R$, $S'$

$$Q(S,A) \leftarrow Q(S,A) + \alpha\Big[\underbrace{\overbrace{\boxed{\underbrace{R}_{\text{Immediate Reward}} + \gamma \max_a Q(S',a)}}^{\text{Target}} - \overbrace{\boxed{Q(S,A)}}^{\text{Prediction}}}_{\text{loss}}\Big]$$

$S \leftarrow S'$

until $S$ is terminal

# Deep Q-Network (DQN)

- Represent value function by deep Q-network with weights $w$

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

# Deep Q-Network (DQN)

- Represent value function by deep Q-network with weights $w$

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

- Define objective function

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

# Deep Q-Network (DQN)

- Represent value function by deep Q-network with weights $w$

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

- Define objective function

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

- Leading to the following Q-leaning gradient

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right) \frac{\partial Q(s, a, w)}{\partial w}\right]$$

# Deep Q-Network (DQN)

- Represent value function by deep Q-network with weights $w$

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

- Define objective function

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

- Leading to the following Q-leaning gradient

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right) \frac{\partial Q(s, a, w)}{\partial w}\right]$$

- Optimize objective end-to-end by SGD, using $\frac{\partial L(w)}{\partial w}$

# Supervised SGD vs Q-Learning SGD

- SGD update assuming supervision

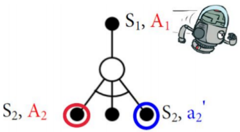
$$J(\mathbf{w}) = \mathbb{E}_\pi \left[ (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2 \right]$$

$$\Delta \mathbf{w} = \alpha (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

- SGD update assuming supervision

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[ (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2 \right]$$

$$\Delta\mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$$




$A_2 \sim$ *epsilon greedy* $(\cdot|S_2)$

$a_2' \sim \text{greedy}(\cdot|S_2)$

$$Q(S_1, A_1) := Q(S_1, A_1) + \alpha\left(R_2 + \gamma \max_{a_2'} Q(S_2, a_2') - Q(S_1, A_1)\right)$$

# Supervised SGD vs Q-Learning SGD

- SGD update assuming supervision

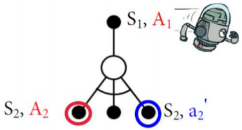
$$J(\mathbf{w}) = \mathbb{E}_\pi \left[ (q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))^2 \right]$$

$$\Delta\mathbf{w} = \alpha(q_\pi(S,A) - \hat{q}(S,A,\mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$$

- SGD update for Q-Learning

$$J(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s',a',w)}_{\text{target}} - Q(s,a,w)\right)^2\right]$$

$$\Delta\mathbf{w} = \left(r + \gamma \max_{a'} Q(s',a',w) - Q(s,a,w)\right)\frac{\partial Q(s,a,w)}{\partial w}$$




$A_2 \sim$ *epsilon greedy* $(\cdot|S_2)$

$a_2' \sim \text{greedy}(\cdot|S_2)$

$$Q(S_1, A_1) := Q(S_1, A_1) + \alpha\left(R_2 + \gamma \max_{a_2'} Q(S_2, a_2') - Q(S_1, A_1)\right)$$

# Stability issues with Deep RL

Naive Q-learning oscillates or diverge with neural nets

1. Data is sequential
   - Successive samples are correlated, non-iid
2. Policy changes rapidly with slight changes to Q-values
   - Policy can oscillate
   - Disctribution of data can swing from one extreme to another
3. Scale of reqards and Q-values is unknown
   - Naive Q-learning gradients can be large unstable when backpropagated

# Deep Q-Networks

DQN provides a stable solution to deep value-based RL

1. Use experience replay
   - Break correlations in data, bring us back to iid setting
   - Learn from all past policies
2. Freeze target Q-network
   - Avoid oscillations
   - Break correlations between Q-network and target
3. Clip rewards or normalize network adaptive to sensible range
   - Robust gradients

# Stable Deep RL (1): Experience Replay

**Problem:** approximation of Q-values using non-linear functions is not stable

**Solution:**

- Take action $a_t$ according to $\epsilon$-greedy policy

# Stable Deep RL (1): Experience Replay

**Problem:** approximation of Q-values using non-linear functions is not stable

**Solution:**

- Take action $a_t$ according to $\epsilon$-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in a replay memory $D$

# Stable Deep RL (1): Experience Replay

**Problem:** approximation of Q-values using non-linear functions is not stable

**Solution:**

- Take action $a_t$ according to $\epsilon$-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in a replay memory $D$
- Sample random mini-batch of transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ from $D$

# Stable Deep RL (1): Experience Replay

**Problem:** approximation of Q-values using non-linear functions is not stable

**Solution:**

- Take action $a_t$ according to $\epsilon$-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in a replay memory $D$
- Sample random mini-batch of transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ from $D$
- Optimize MSE between Q-network and Q-learning targets, e.g.

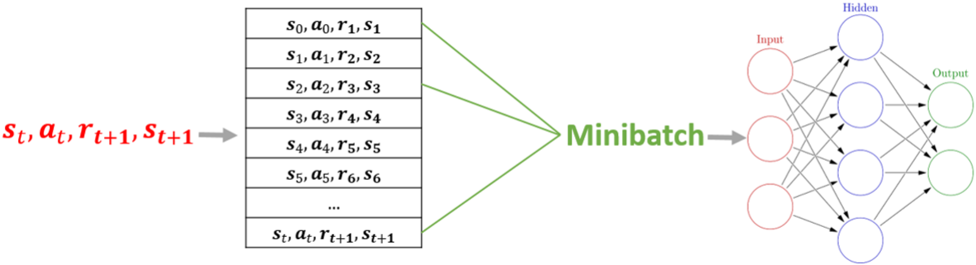
$$\mathcal{L}(w) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[ \left( r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w) \right)^2 \right]$$

This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum.

# Stable Deep RL (1): Experience Replay

**Problem:** approximation of Q-values using non-linear functions is not stable

**Solution:**



# Stable Deep RL (2): Fixed Target Q-Network

- Create two deep networks $w^-$ and $w$

# Stable Deep RL (2): Fixed Target Q-Network

- Create two deep networks $w^-$ and $w$
- Use the first one to retrieve $Q$ values while the second one includes all updates in the training. After $C$ updates synchronize $w^- \leftarrow w$.

  **Motivation:** Fix the Q-value targets temporarily so we don't have a moving target.

# Stable Deep RL (2): Fixed Target Q-Network

To avoid oscillations, fix parameters used in Q-learning target

- Compute Q-learning targets w.r.t. old, fixed parameters $w^-$

$$r + \gamma \max_{a'} Q(s', a', w^-)$$

- Optimize MSE between Q-network and Q-learning targets

$$\mathcal{L}(w) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}} \left[ \left( r + \gamma \max_{a'} Q(s', a', w^-) - Q(s, a, w) \right)^2 \right]$$

- Periodically update fixed parameters $w^- \leftarrow w$

# Stable Deep RL (3): Reward/Value Range

- DQN clips the reward $[-1, +1]$

# Stable Deep RL (3): Reward/Value Range

- DQN clips the reward $[-1, +1]$
- This prevents Q-values from becoming too large

# Stable Deep RL (3): Reward/Value Range

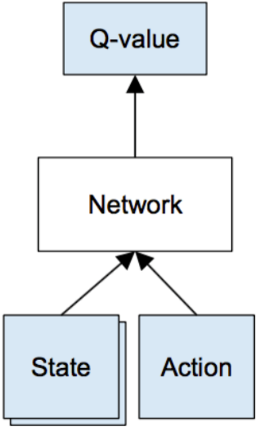
- DQN clips the reward $[-1, +1]$
- This prevents Q-values from becoming too large
- Ensures gradients are well-conditioned

# Stable Deep RL (3): Reward/Value Range

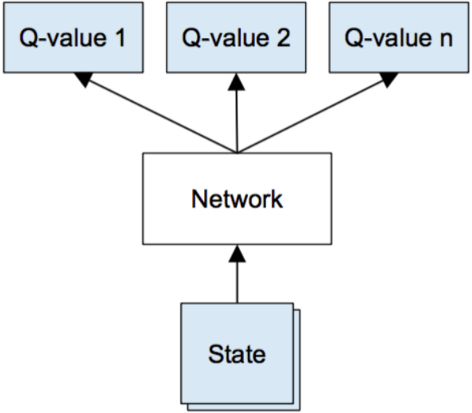
- DQN clips the reward $[-1, +1]$
- This prevents Q-values from becoming too large
- Ensures gradients are well-conditioned
- Can't tell difference between small and large rewards

# Deep Q-Network (DQN) Architecture

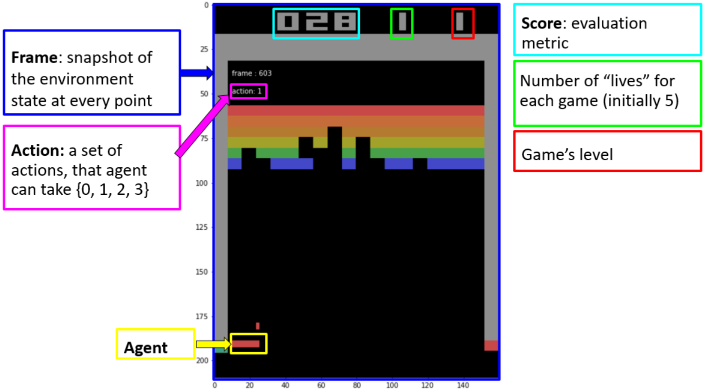
Naive DQN



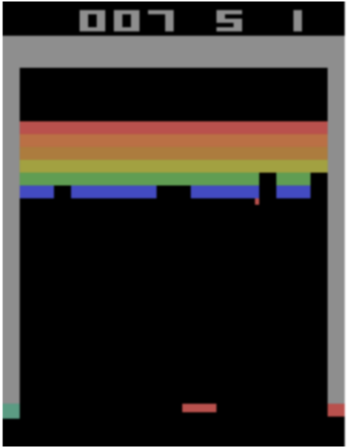

Optimized DQN used by DeepMind

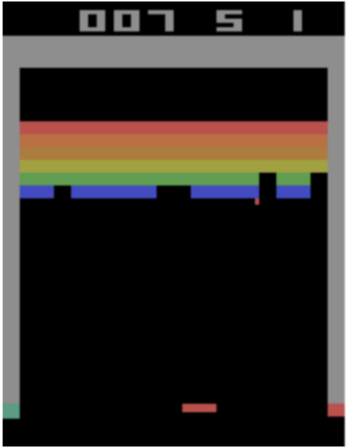


# Reinforcement Learning in Atari



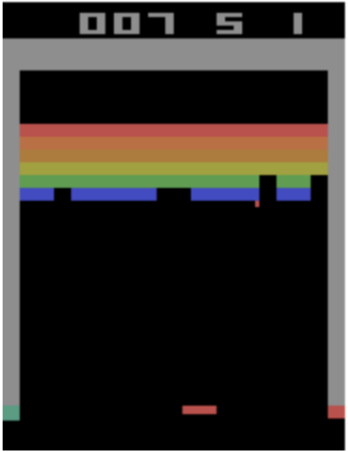
# Reinforcement Learning in Atari

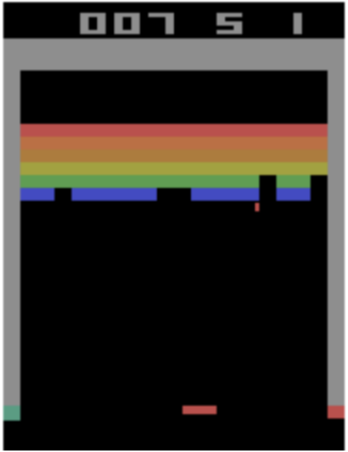




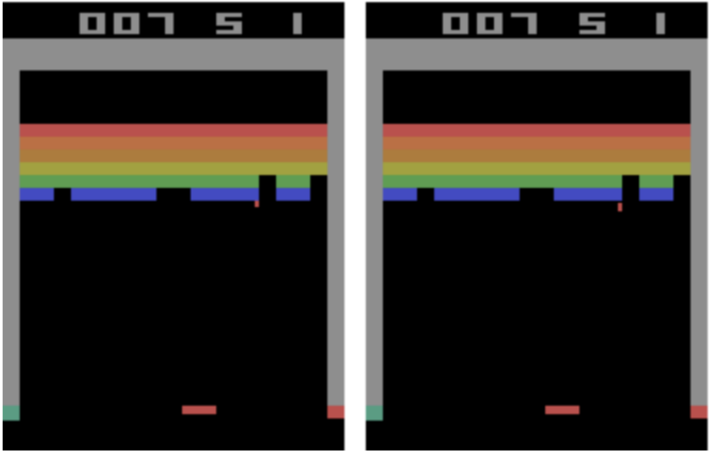

Do we have all the information to start training?

## Reinforcement Learning in Atari



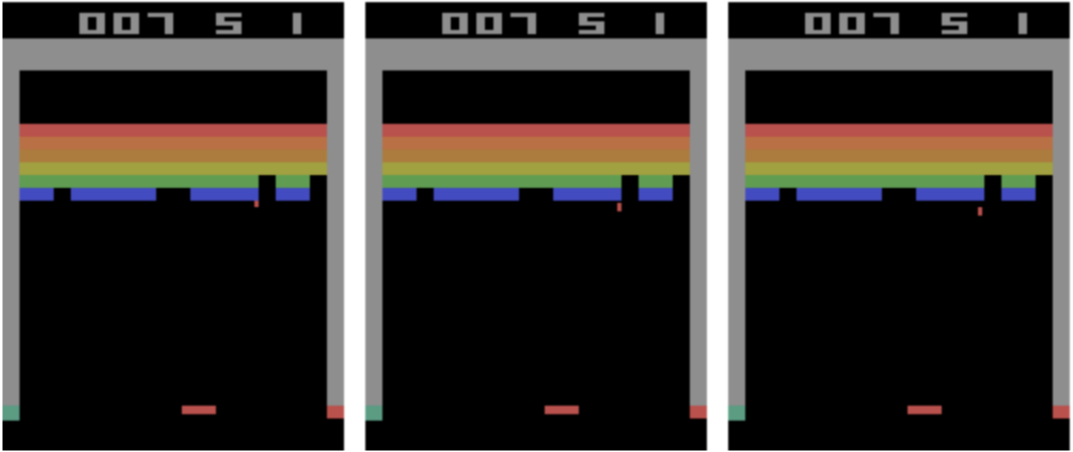



# Reinforcement Learning in Atari



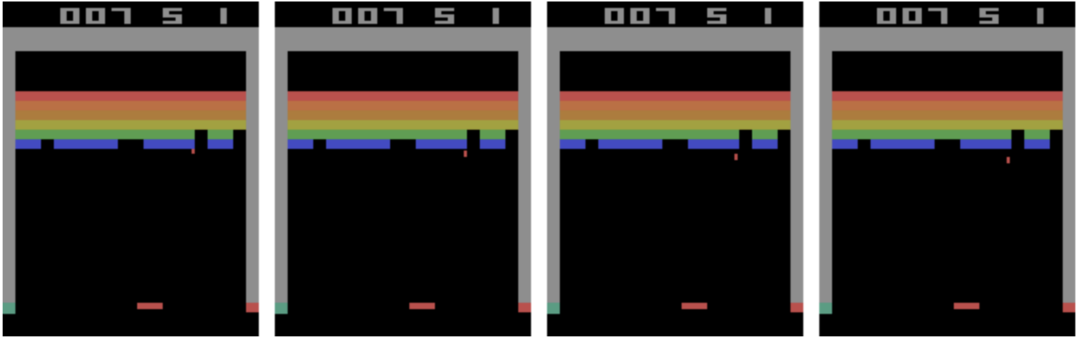

We know the direction and the velocity of the ball. But do we know its acceleration?

# Reinforcement Learning in Atari




Now we can extract all the information about the state.

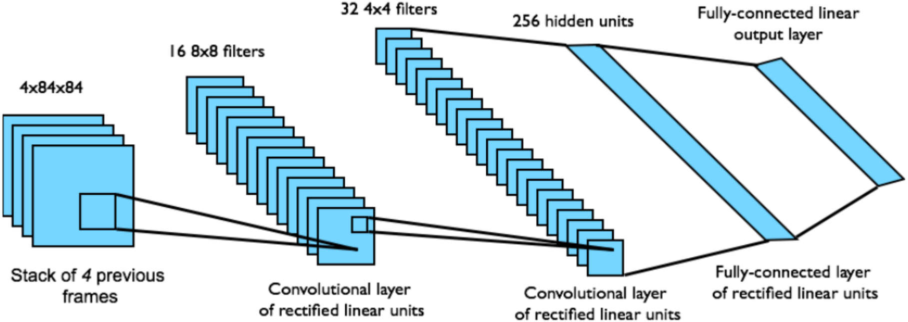
# Reinforcement Learning in Atari



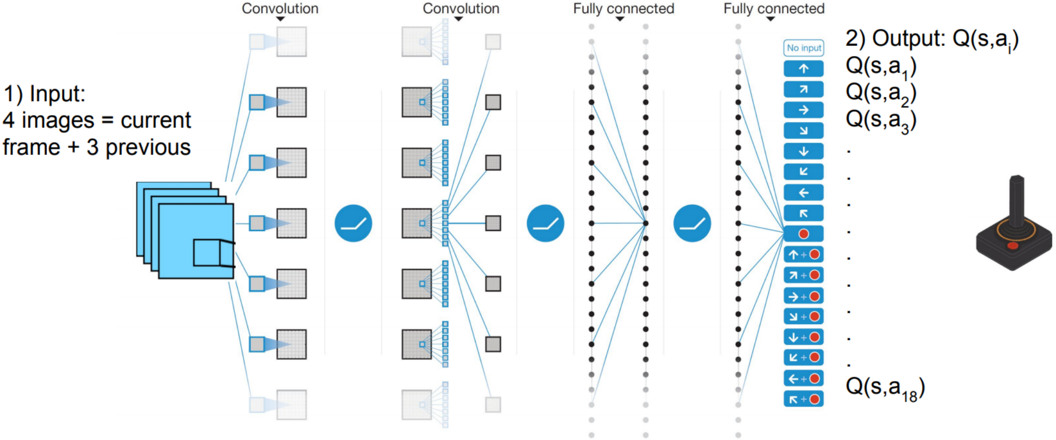

To make sure we can generalize for other games as well, we keep 4 frames as an input.

# DQN in Atari

- End-to-end learning of values $Q(s, a)$ from pixels $s$
- Input state $s$ is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step



# DQN in Atari


1) Input:
4 images = current
frame + 3 previous
Convolution
Convolution
Fully connected
Fully connected
No input
2) Output: Q(s,a_i)
Q(s,a_1)
Q(s,a_2)
Q(s,a_3)
Q(s,a_18)

# DQN Algorithm

**Algorithm 1: deep Q-learning with experience replay.**

Initialize replay memory $D$ to capacity $N$

Initialize action-value function $\boxed{Q}$ with random weights $\theta$

Initialize target action-value function $\boxed{\hat{Q}}$ with weights $\theta^{-} = \theta$

**For** episode $= 1, M$ **do**

# DQN Algorithm

**Algorithm 1: deep Q-learning with experience replay.**

Initialize replay memory $D$ to capacity $N$

Initialize action-value function $\boxed{Q}$ with random weights $\theta$

Initialize target action-value function $\boxed{\hat{Q}}$ with weights $\theta^{-} = \theta$

**For** episode $= 1, M$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

**For** $t = 1$,T **do**

With probability $\varepsilon$ select a random action $a_t$

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

# DQN Algorithm

**Algorithm 1: deep Q-learning with experience replay.**

Initialize replay memory $D$ to capacity $N$

Initialize action-value function $\boxed{Q}$ with random weights $\theta$

Initialize target action-value function $\boxed{\hat{Q}}$ with weights $\theta^{-} = \theta$

**For** episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  **For** $t = 1$,T **do**

    With probability $\varepsilon$ select a random action $a_t$

    otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

# DQN Algorithm

**Algorithm 1: deep Q-learning with experience replay.**

Initialize replay memory $D$ to capacity $N$

Initialize action-value function $Q$ with random weights $\theta$

Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$

**For** episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  **For** $t = 1$,T **do**

    With probability $\varepsilon$ select a random action $a_t$

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from $D$

# DQN Algorithm

**Algorithm 1: deep Q-learning with experience replay.**

Initialize replay memory $D$ to capacity $N$

Initialize action-value function $\boxed{Q}$ with random weights $\theta$

Initialize target action-value function $\boxed{\hat{Q}}$ with weights $\theta^{-} = \theta$

**For** episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  **For** $t = 1$,T **do**

    With probability $\varepsilon$ select a random action $a_t$

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from $D$

    $$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ \boxed{r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right)} & \text{otherwise} \end{cases}$$

    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters $\theta$

    Every $C$ steps reset $\hat{Q} = Q$

  **End For**

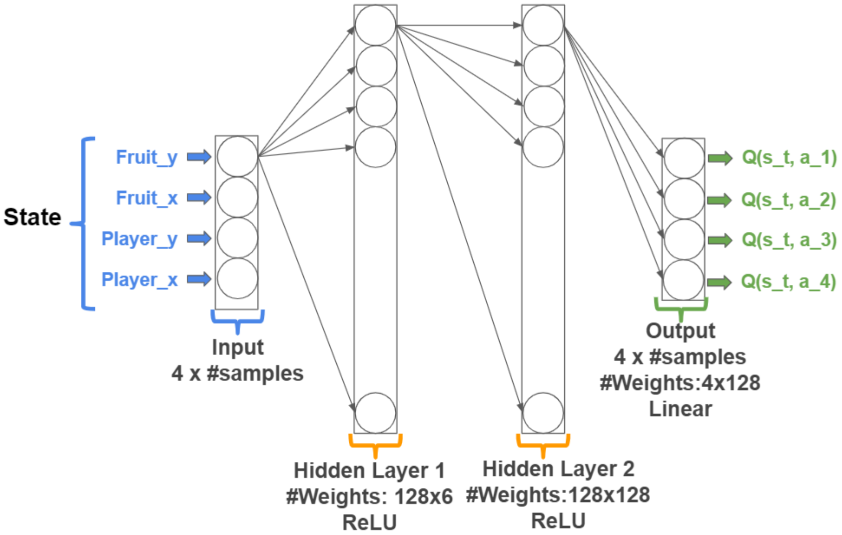
**End For**

# DQN Example

Demo for Project 4, CSE 574 Machine Learning, Fall 2018 [Instructor: Dr. Sargur N. Srihari]
Authors: Alina Vereshchaka and Nathan Margaglio

Tom & Jerry

# DQN Example


State
Fruit_y
Fruit_x
Player_y
Player_x
Input
4 x #samples
Hidden Layer 1
#Weights: 128x6
ReLU
Hidden Layer 2
#Weights:128x128
ReLU
Output
4 x #samples
#Weights:4x128
Linear
Q(s_t, a_1)
Q(s_t, a_2)
Q(s_t, a_3)
Q(s_t, a_4)

# DQN Summary

- DQN: Q-Learning but with a Deep Neural Network as a function approximator

# DQN Summary

- DQN: Q-Learning but with a Deep Neural Network as a function approximator
- Using a non-linear Deep Neural Network is powerful, but training is unstable if we apply it naively

# DQN Summary

- DQN: Q-Learning but with a Deep Neural Network as a function approximator
- Using a non-linear Deep Neural Network is powerful, but training is unstable if we apply it naively
- Experience Replay Trick: Store experience $(S, A, R, S_{next})$ in a replay buffer and sample minibatches from it to train the network. This decorrelates the data and leads to better data efficiency. In the beginning, the replay buffer is filled with random experience

# DQN Summary

- DQN: Q-Learning but with a Deep Neural Network as a function approximator
- Using a non-linear Deep Neural Network is powerful, but training is unstable if we apply it naively
- Experience Replay Trick: Store experience $(S, A, R, S_{next})$ in a replay buffer and sample minibatches from it to train the network. This decorrelates the data and leads to better data efficiency. In the beginning, the replay buffer is filled with random experience
- By using a Convolutional Neural Network as the function approximator on raw pixels of Atari games where the score is the reward we can learn to play many of those games at human-like performance.