Value Function Approximation

Reinforcement Learning · Lecture 5

Christopher Mutschler

University of Technology Nuremberg

12 November 2026

TODAY

Where we are going

1. **Why function approximation?** Generalize beyond a table.
2. **Linear VFA.** Features, regression, semi-gradient updates.
3. **Fitted Q-Iteration.** Turn Bellman backups into supervised learning.
4. **Deep Q-Networks.** Neural approximation plus two stabilizers.

TODAY'S QUESTION

How do we keep Q-learning useful when the state space is too large to enumerate?

PART 1

Why approximate value functions?

Tables remember. Function approximators generalize.

Tabular Q-learning updates one entry at a time

ONE-STEP TARGET

$$_t = r_t + \gamma \max Q(s_{t+1}, a')$$

$$, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[y_t - Q(s_t, a_t) \right]$$

Great when every state-action pair has its own cell. Not so great when there are millions, billions, or infinitely many states.

state	←	→
	0.2	0.3
	−0.1	0.7
	0.0	**1.4**
⋮	⋮	⋮
	?	?

No automatic sharing between similar rows.

CHALLENGE #1

In real world problems, the state space can be large

Backgammon states

approx. number of states:

20

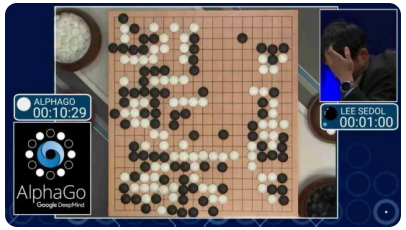


Source: ai.berkeley.edu/lecture_slides.html

Computer Go states

approx. number of states:

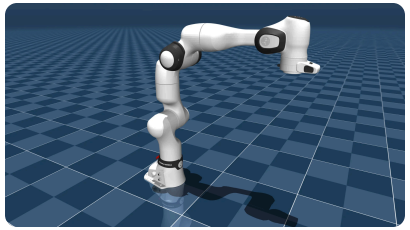
170



Source: youtube.com/watch?v=HT-UZkiOLv8

Robot arm: infinitely many states

approx. number of states:



Source: MuJoCo model documentation

STORAGE

There are too many states and/or actions to store in memory.

DATA

Even if the table fits, visiting enough state-action pairs takes far too long.

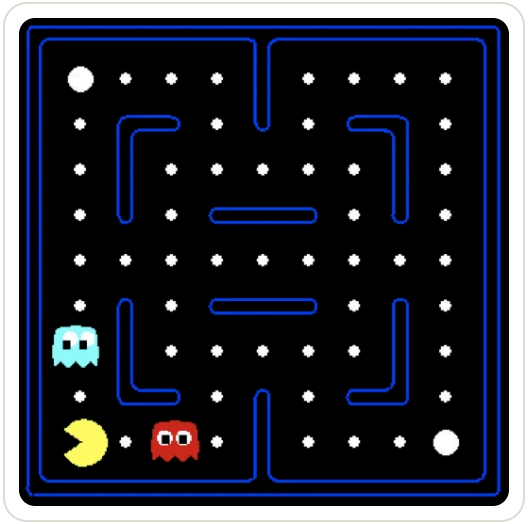
LOCAL UPDATES

Only nearby states are observed, so value information propagates slowly.

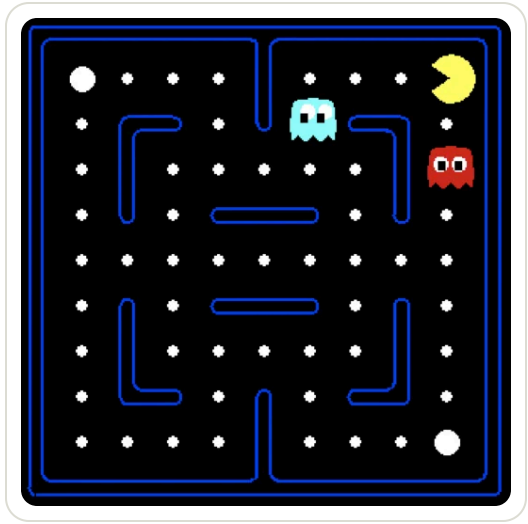
CHALLENGE #2

Generalization across states

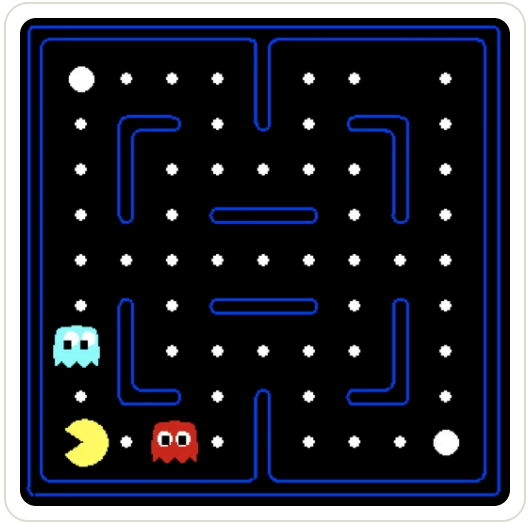
Let’s say we discover through experience that this state is bad:



In naïve Q-learning, we know nothing about this state:



Or even this one!



We need a representation that **generalizes** and speeds up learning across states.

Reference: Pieter Abbeel: *CS 188 Introduction to Artificial Intelligence.* Fall 2018.

From a table to a parameterized function

Value Function Representations

Exact:

- A table with a distinct value for each case
- : one entry per
- : one entry for each (s, a) pair

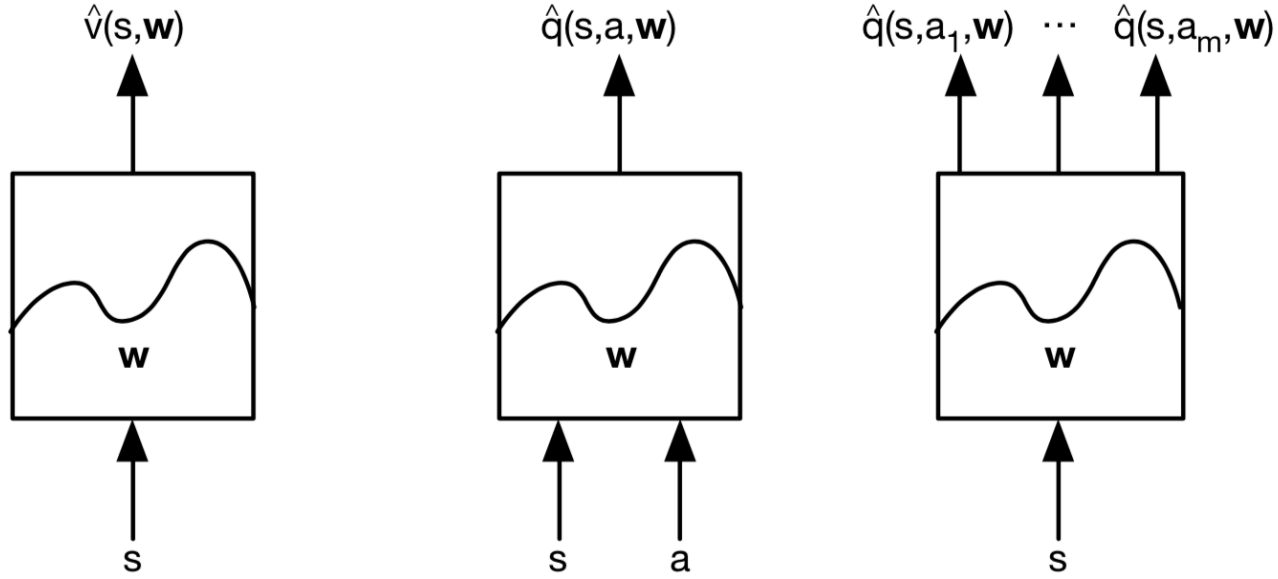
Approximate:

- Approximate or with a function approximator
- e.g., neural networks, polynomials, RBFs, …

$$\hat{v}_\pi(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}_\pi(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

+ We only need to store the approximator parameters

− Convergence properties do not hold anymore in general



David Silver, Reinforcement Learning lecture notes, 2016.

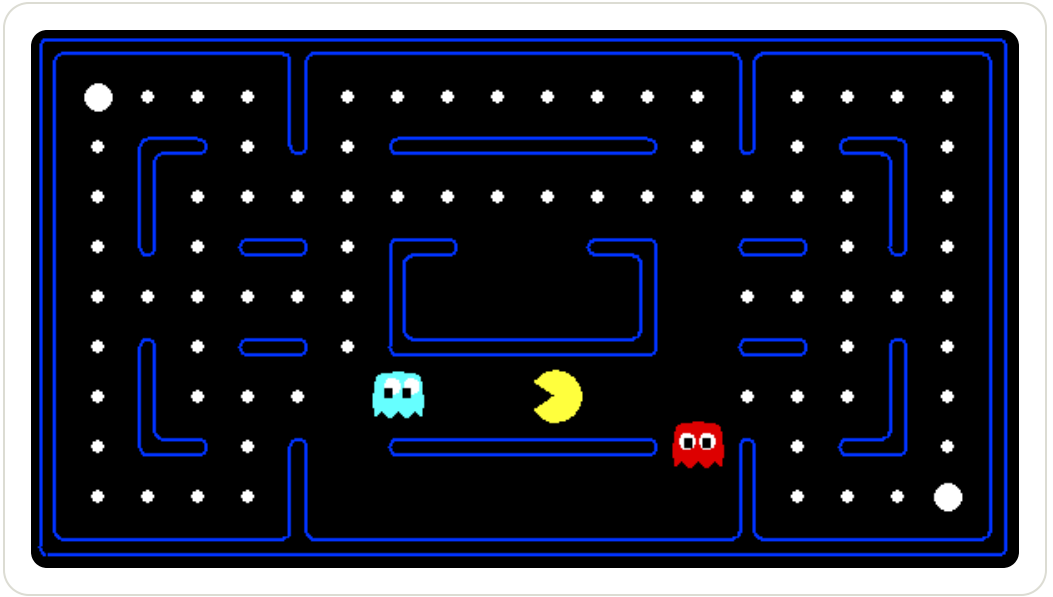
Describe a state using a vector of features

VFA: Describe a state using a vector of features.

Features are functions from states to real numbers that capture important properties of the state.

Example features for Pac-Man:

- Distance to closest ghost
- Distance to closest dot
- Number of ghosts
- …



Example adapted from Pieter Abbeel, CS 188 Introduction to Artificial Intelligence.

Turn value approximation into a regression problem

Our goal is to learn parameters that approximate the target action value well.

$$C = \quad ^{+}(s,a) - \hat{Q}^{\pi}(s,a; \quad)^{2}$$

↓

$$C = \quad ^{+}(s,a) - \phi(s,a)^{\mathsf{T}} \quad ^{2}$$

↓ gradient

$$\frac{\partial C}{\quad} = -2\,\phi(s,a)\,\left[Q^{+}(s,a) - \phi(s,a)^{\mathsf{T}}\mathbf{w}\right]$$

↓ gradient-descent update

$$\leftarrow \mathbf{w} + 2\eta\,\phi(s,a) \quad ^{+}(s,a) - \hat{Q}^{\pi}(s,a;\mathbf{w})\Big]$$

WHERE DOES Q^{+} COME FROM?

Method	Target
Q-learning	$r + \gamma \max_{a'} Q(s',a')$
SARSA	$r + \gamma Q(s',a')$
Monte Carlo	$_{t}$

The regression target is supplied by the RL algorithm.

Linear in the parameters, rich in the features

$$\hat{Q}(s, a; \mathbf{w}) = \phi(s, a)^\top \mathbf{w} = \sum^{d} \phi_j(s, a) w_j$$

WHY WE LIKE IT

Fast updates, interpretable features, and much stronger stability results than arbitrary nonlinear approximators.

BUT

Good performance depends heavily on choosing a feature map that makes the value function easy to represent.

The learner tunes ; the designer decides what can express.

FEATURE DESIGN

Several ways to turn a state into features



Polynomial

Products and powers of coordinates



Fourier

Cosine basis at multiple frequencies



Tile coding

Sparse overlapping partitions



RBF

Localized Gaussian features

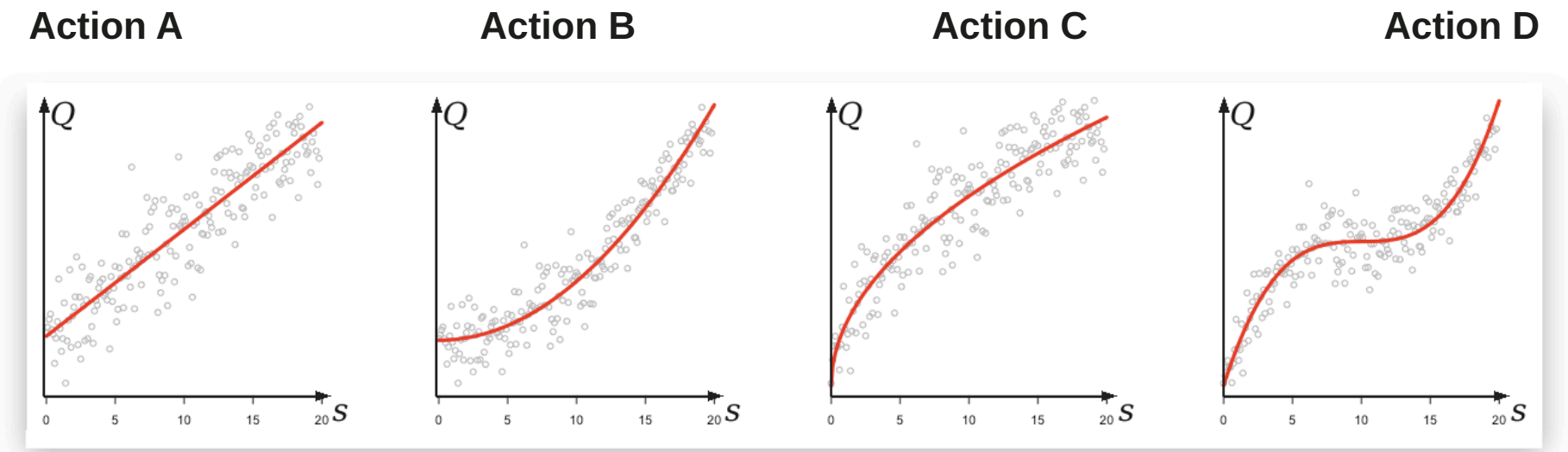
SAME LEARNER, DIFFERENT GEOMETRY

Changing changes which states share parameters — and therefore which states generalize to one another.

The approximation is linear in the parameters . The feature map $a)$ may be highly non-linear.

Polynomial features create non-linear value functions

$$_i(s, a_\ell) = \prod^{k} s_j^{c_{i,j}}, \qquad c_{i,j} \in \{0, 1, \dots, n\}, \quad a_\ell \in \mathcal{A}$$



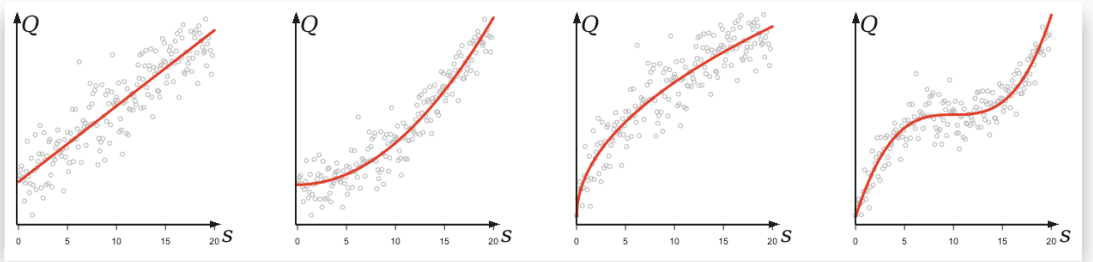
A feature map can lift a low-dimensional state into many monomials

$$, s_2)^\top \mapsto (1, s_1, s_2, s_1 s_2)^\top$$

$$(s_1, s_2)^\top \mapsto (1, s_1, s_2, s_1 s_2, s^2, s_i^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$$

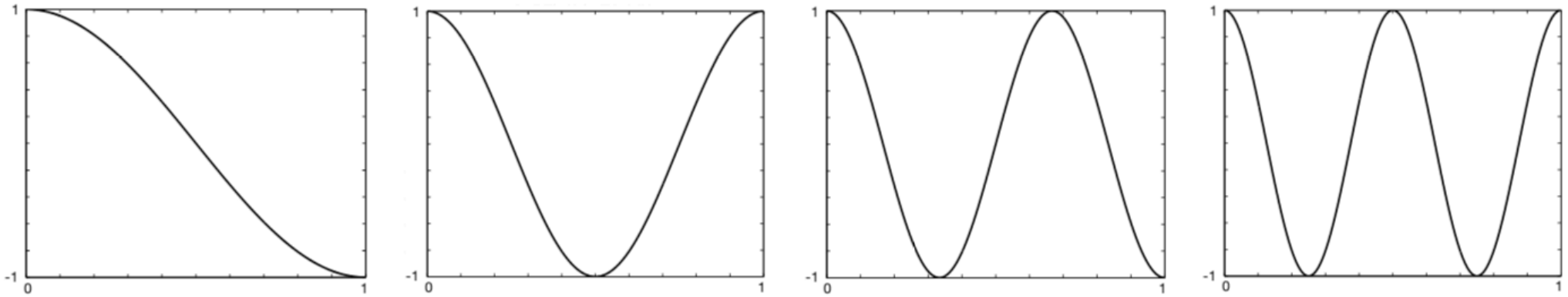
TRADE-OFF

Higher degree increases expressiveness — but the number of features grows quickly with state dimension.



Fourier features represent smooth structure with different frequencies

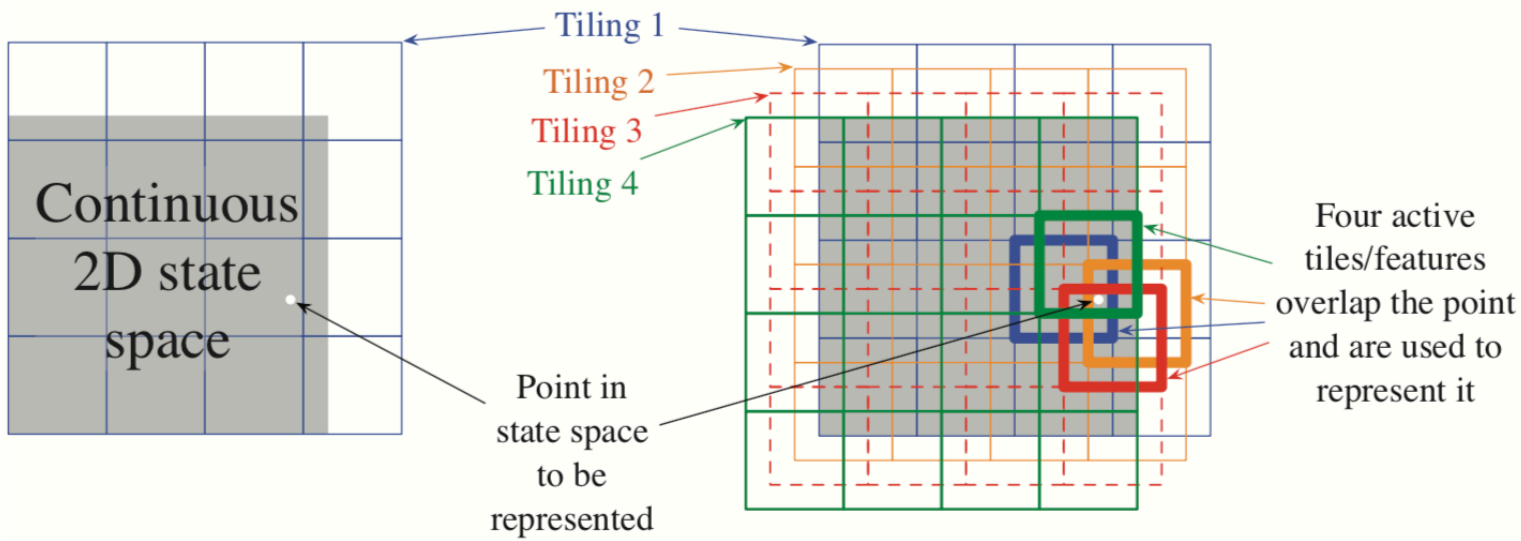
$$\phi_i(s, a\ ) = \cos(i\pi s), \qquad s \in [0, 1], \quad a\ \in \mathcal{A}$$



Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press.

Tile coding turns a continuous state into a sparse binary representation

Several offset tilings cover the same continuous state space. A point activates one tile per tiling.



KEY IDEA

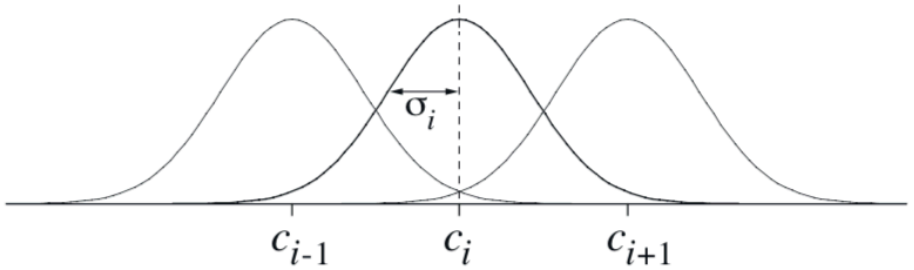
Nearby states share some active tiles → local generalization without a single hard discretization.

Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press.

Radial basis functions create local, smooth features

Example features: Radial Basis Functions (RBFs)

$$_i(s, a_j) \doteq \exp\left(\frac{\|s - c_i\|^2}{2\sigma_i^2}\right), \qquad a_j \in \mathcal{A}$$



Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press.

Convergence of control algorithms

Function approximation lets us generalize, but it changes the convergence story.

Algorithm	Table Lookup	Linear	Non-linear
Monte-Carlo Control	✓	(✓)	✗
SARSA	✓	(✓)	✗
Q-learning	✓	✗	✗

(✓) = chatters around a near-optimal value function

Two important questions

1. Can we approximate any V value function with a linear function approximator?
2. Is it **easy to find** such a linear function approximator?

KEEP BOTH QUESTIONS SEPARATE

Expressiveness and feature discovery are not the same problem.

QUESTION 1 · EXPRESSIVENESS

Can a linear FA approximate any value function?

1. Can we approximate any V value function with a linear FA?

ANSWER

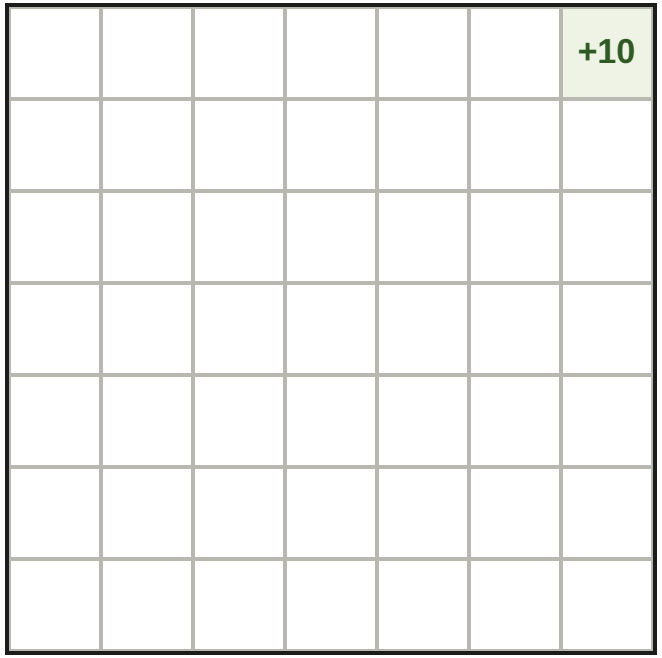
YES! (But the proof is out of scope for this class.)

Is it easy to find such a linear FA?

2. Is it easy to find such a linear FA? → **YES!**

EXAMPLE GRIDWORLD

- No obstacles
- deterministic actions (U/D/L/R)
- no discounting
- reward is −1 everywhere except +10 at the goal



goal at the top-right corner

Represent the state and perform linear VFA

Represent state by a feature vector:

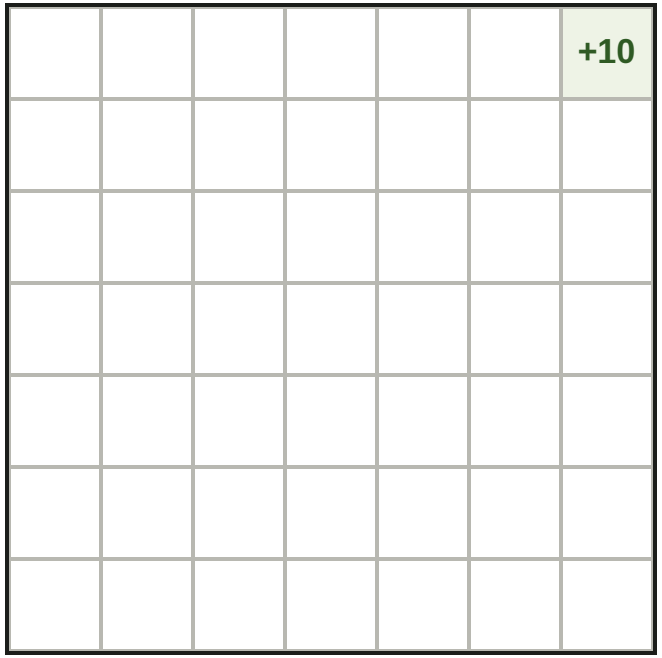
$$\phi(s) = \begin{bmatrix} 1 \\ x \\ y \end{bmatrix}$$

Perform linear value-function approximation:

$$\hat{V}(s, \mathbf{w}) = \phi(s)^\top \mathbf{w}$$

$$= \begin{bmatrix} 1 & x & y \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

$$\hat{V}(s, \mathbf{w}) = w_0 + w_1 x + w_2 y$$



Is there a good linear approximation? → YES

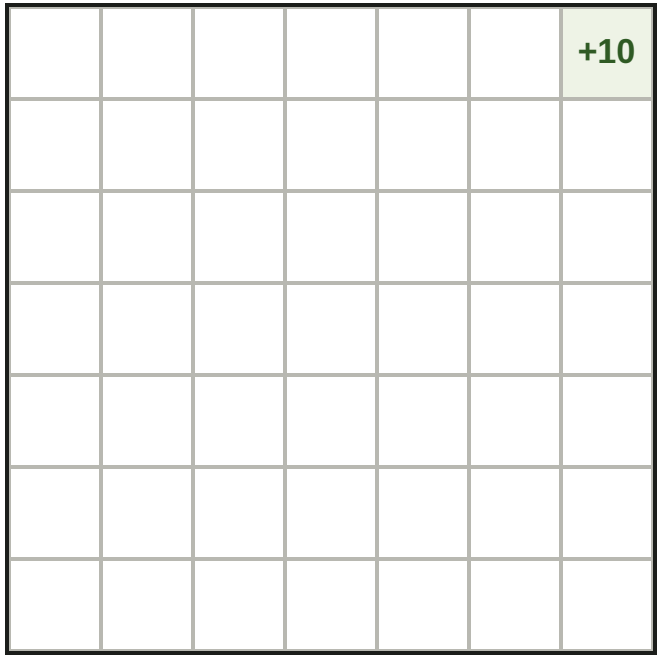
Choose weights that encode distance to the goal:

$$\hat{V}(s;\mathbf{w}) = \begin{bmatrix}1 & x & y\end{bmatrix} \begin{bmatrix}10\\-1\\-1\end{bmatrix}$$

$$\hat{} \qquad \mathbf{w}) = 10 - x - y = 10 - |x + y|$$

INTERPRETATION

The value decreases linearly with the **Manhattan distance** to the goal.



goal value +10

Move the reward: the same features no longer work

$\hat{}$ $\mathbf{w}) = w_0 + w_1 x + w_2 y$

WITH REWARD IN THE INTERIOR

Is there a good linear approximation with only $y]$? → **NO!**

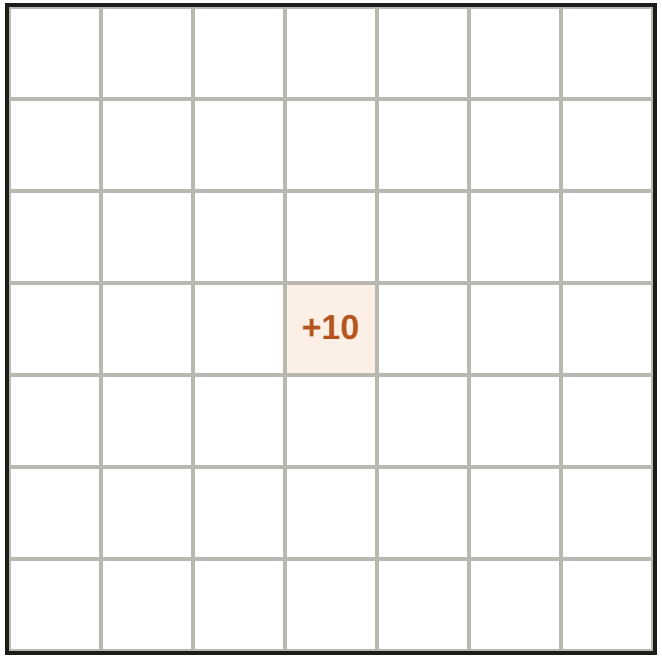
Add a nonlinear feature that measures distance to the new goal:

$z =$ $-$ $+ |3 - y|$

$$\phi(s) = \begin{bmatrix} 1 \\ x \\ y \\ z \end{bmatrix}$$

$\hat{V}$ $= w_0 + w_1 x + w_2 y + w_3 z$

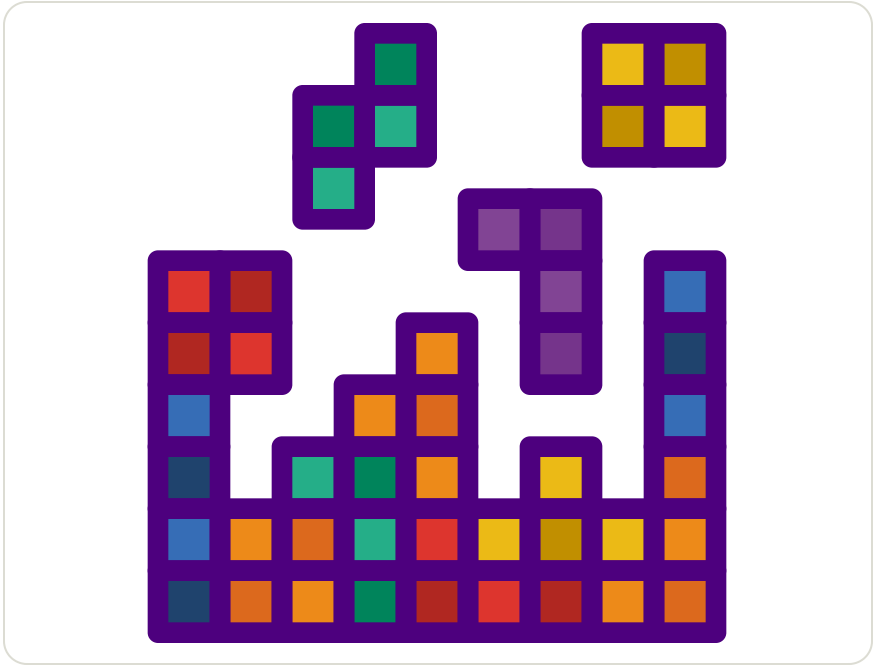
Now: → YES!



reward moved to (3,3)

Is it easy to find a good linear FA? → NO!

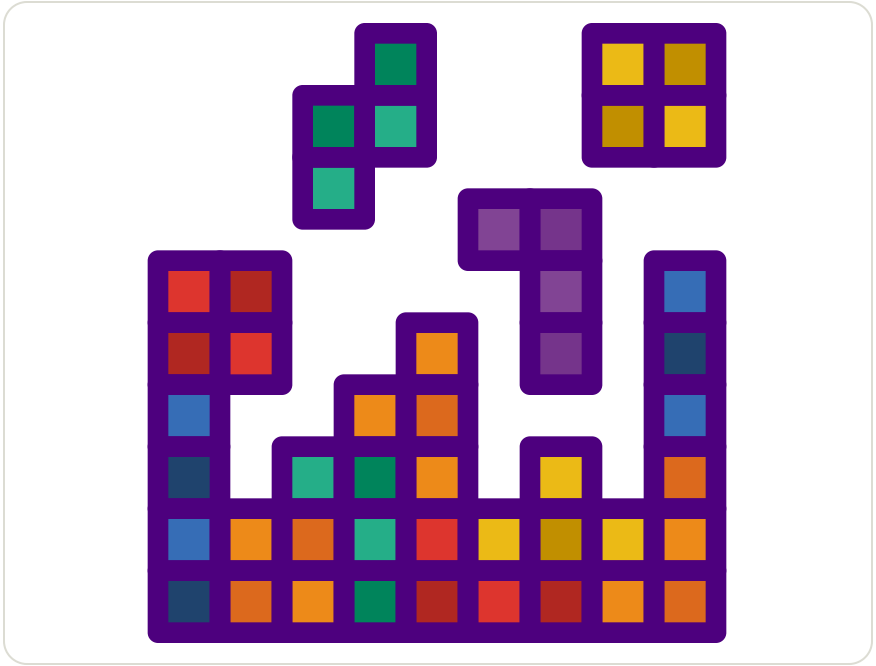
Feature	Description
Landing Height	Height of last piece is added.
Eroded Piece Cells	#rows eliminated in the last move × #bricks eliminated from the last piece.
Row Transitions	#horizontal full↔empty transitions between cells.
Column Transitions	Same idea for vertical transitions.
Holes	#empty cells covered by at least one full cell.



Thierry, C. & Scherrer, B.: Improvements on Learning Tetris with Cross-Entropy. 2010. · Tetris illustration: flaticon.com/free-icon/tetris_1006985

Is it easy to find a good linear FA? → NO!

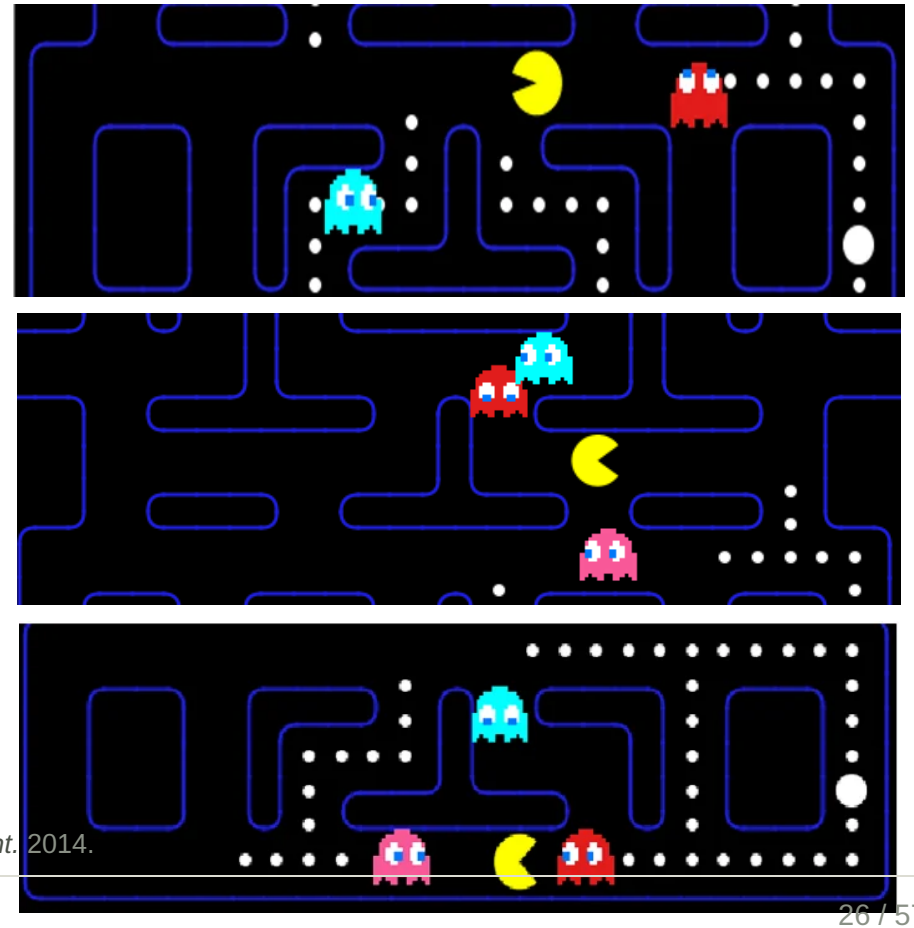
Feature	Description
Board Wells	Sum of well depths (triangular depth penalty).
Column Height	Height of the p-th column.
Column Difference	Absolute height difference between adjacent columns.
Maximum Height	Maximum pile height.
Holes	Empty cells covered by at least one full cell.



Thierry, C. & Scherrer, B.: Improvements on Learning Tetris with Cross-Entropy. 2010. · Tetris illustration: flaticon.com/free-icon/tetris_1006985

Hand-crafted features must encode the relevant geometry

Feature	Description
Nearby Walls	Existence of walls in all four directions.
Nearest Target	Direction of the nearest target where Ms. Pac-Man should move.
Nearby Ghosts	Existence of a ghost in all four directions.
Existence of Escape	Whether Ms. Pac-Man can move freely or is trapped.



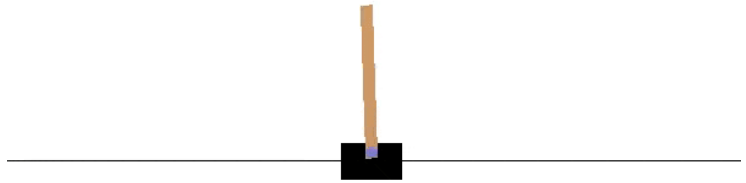
Tziortziotis, N., Tziortziotis, K. & Blekas, K.: *Play Ms. Pac-Man Using an Advanced Reinforcement Learning Agent.* 2014.

A pendulum: 30 carefully placed basis functions

We applied LSPI with a set of 10 basis functions for each of the 3 actions, thus a total of 30 basis functions, to approximate the value function. These 10 basis functions included a constant term and 9 radial basis functions (Gaussians) arranged in a 3×3 grid over the 2-dimensional state space. In particular, for some state $s = (\theta, \dot{\theta})$ and some action a, all basis functions were zero, except the corresponding active block for action a which was

$$\left(1,\ e^{-\frac{\|s-\mu_1\|^2}{2\sigma^2}},\ e^{-\frac{\|s-\mu_2\|^2}{2\sigma^2}},\ e^{-\frac{\|s-\mu_3\|^2}{2\sigma^2}},\ \dots,\ e^{-\frac{\|s-\mu_9\|^2}{2\sigma^2}}\right)^\intercal,$$

where the μ_i's are the 9 points of the grid $\{-\pi/4,\ 0,\ +\pi/4\} \times \{-1,\ 0,\ +1\}$ and $\sigma^2 = 1$.

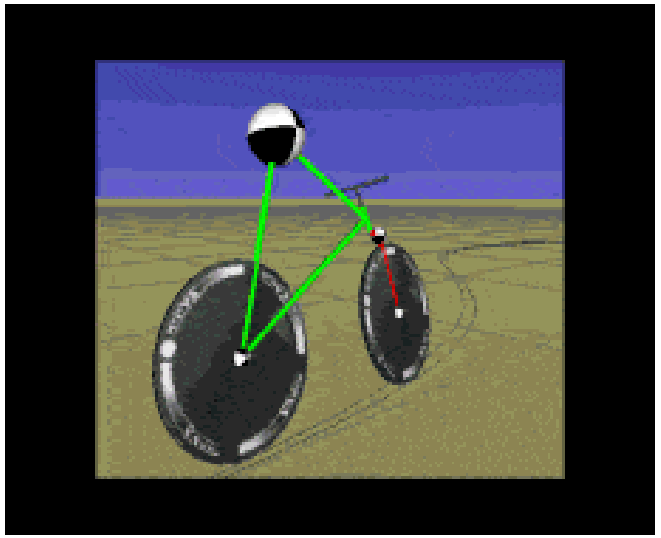


$$\phi(s,a) = \left(1, e^{-\frac{\|s-\mu_1\|^2}{2\sigma^2}}, e^{-\frac{\|s-\mu_2\|^2}{2\sigma^2}}, e^{-\frac{\|s-\mu_3\|^2}{2\sigma^2}}, \dots, e^{-\frac{\|s-\mu_9\|^2}{2\sigma^2}}\right)^\intercal$$

$$\text{with} \quad \mu_i \in \{-\pi/4, 0, +\pi/4\} \times \{-1, 0, +1\}, \quad \sigma^2 = 1$$

Lagoudakis, M. G. & Parr, R.: *Least-Squares Policy Iteration.* JMLR, 2003.

A bicycle: 100 hand-designed basis functions



The goal in the *bicycle balancing and riding* problem (Randløv and Alstrøm, 1998) is to learn to balance and ride a bicycle to a target position located 1 km away from the starting location. Initially, the bicycle's orientation is at an angle of 90° to the goal. The state description is a six-dimensional real-valued vector $(\theta, \dot{\theta}, \omega, \dot{\omega}, \ddot{\omega}, \psi)$, where θ is the angle of the handlebar, ω is the vertical angle of the bicycle, and ψ is the angle of the bicycle to the goal. The actions are the torque τ applied to the handlebar (discretized to $\{-2, 0, +2\}$) and the displacement of the rider υ (discretized to $\{-0.02, 0, +0.02\}$). In our experiments, actions are restricted so that either $\tau = 0$ or $\upsilon = 0$ giving a total of 5 actions.[12] The noise in the system is a uniformly distributed term in $[-0.02, +0.02]$ added to the displacement component of the action. The dynamics of the bicycle are based on the model of Randløv and Alstrøm (1998) and the time step of the simulation is set to 0.01 seconds.

The state-action value function $Q(s, a)$ for a fixed action a is approximated by a linear combination of 20 basis functions:

$$(1,\ \omega,\ \dot{\omega},\ \omega^2,\ \dot{\omega}^2, \omega\dot{\omega},\ \theta,\ \dot{\theta},\ \theta^2,\ \dot{\theta}^2,\ \theta\dot{\theta},\ \omega\theta,\ \omega\theta^2,\ \omega^2\theta,\ \psi,\ \psi^2,\ \psi\theta,\ \bar{\psi},\ \bar{\psi}^2,\ \bar{\psi}\theta\ )^\intercal\ ,$$

where $\bar{\psi} = \pi - \psi$ for $\psi > 0$ and $\bar{\psi} = -\pi - \psi$ for $\psi < 0$. Note that the state variable $\ddot{\omega}$ is completely ignored. This block of basis functions is repeated for each of the 5 actions, giving a total of 100 basis functions (and parameters).

SCALE

20 basis functions per action × 5 actions = **100 parameters**.

Lagoudakis, M. G. & Parr, R.: *Least-Squares Policy Iteration.* JMLR, 2003.

Idea: Why don’t we replace linear approximation with NNs?

- Because theory tells us that this doesn’t work out.

Non-linear function approximation breaks the classical convergence guarantees.

Algorithm	Table Lookup	Linear	Non-linear
Monte-Carlo Control	✓	(✓)	×
SARSA	✓	(✓)	×
Q-learning	✓	×	×

(✓) = chatters around a near-optimal value function

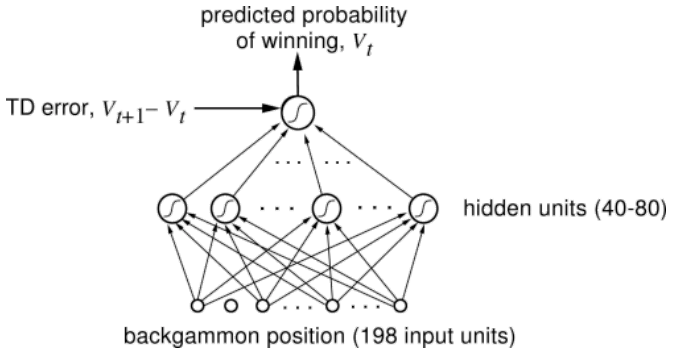
But in some applications, it did 🙂

World's best Backgammon player:

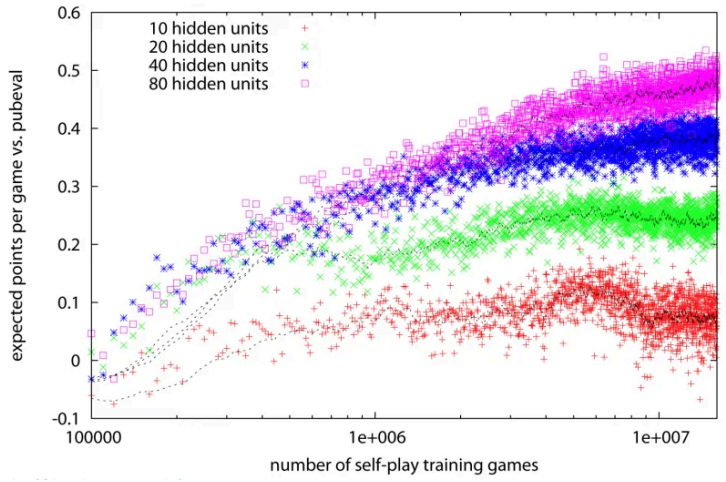
- Neural network with 80 hidden units
- RL from 300,000 games of self-play
- One of the top players in the world



Parenthesis: Gerry Tesauro's TD-Gammon ('92, '94, '95)







$$\leftarrow \mathbf{w} + \alpha \left(V_{t+1} - V_t\right) \sum^{t} \lambda^{t-k} \nabla_{\mathbf{w}} V_k$$

TD error drives a neural value function; more hidden units improved self-play performance.

Tesauro, G.: Temporal difference learning and TD-Gammon. 1995. · Backgammon illustration: flaticon.com/free-icon/backgammon_683899

PART 2

Fitted Q-Iteration

Freeze the Bellman targets. Fit a regressor. Repeat.

Why are one-step online updates not enough?

Use of data

Each update is driven by only one transition, even when we already have a large batch of experience.

Moving targets

Bootstrapped targets change whenever the current value estimate changes.

Model restrictions

Gradient updates tie us to a differentiable parameterization and a particular optimization procedure.

Couldn't we try the problem differently?

Freeze the current Bellman targets, fit one regressor on the whole batch, and repeat.

FITTED Q-ITERATION

Bellman backups become supervised-learning targets

1
Dataset
transitions (s_i, a_i, r_i, s'_i)

→

2
Frozen Bellman labels
compute one target per transition

→

3
Regression
fit Q to those labels

$$y_i^{(k)} = r_i + \gamma \max \hat{Q}_{k-1}(s_i', a')$$

$$\hat{Q}_k = \arg\min_Q \sum \left(Q(s_i, a_i) - y_i^{(k)} \right)^2$$

CONNECTION TO SUPERVISED LEARNING

For one FQI iteration, the targets are fixed. Learning is ordinary regression on Bellman targets; the Bellman error is the regression residual.

What limits Fitted Q-Iteration?

FQI turns reinforcement learning into a sequence of clean regression problems — but each outer iteration is still a batch procedure.

1

Full-batch sweeps

Bellman targets are recomputed for the entire transition dataset at every outer iteration.

2

Refit the regressor

After changing the targets, the function approximator must be fitted again — expensive as the dataset grows.

3

Scaling to rich observations

Hand-designed features and classical regressors become the bottleneck for high-dimensional inputs such as images.

OPEN QUESTION

How can we keep the fixed-target regression idea while learning efficiently from large experience streams and high-dimensional observations?

PART 3

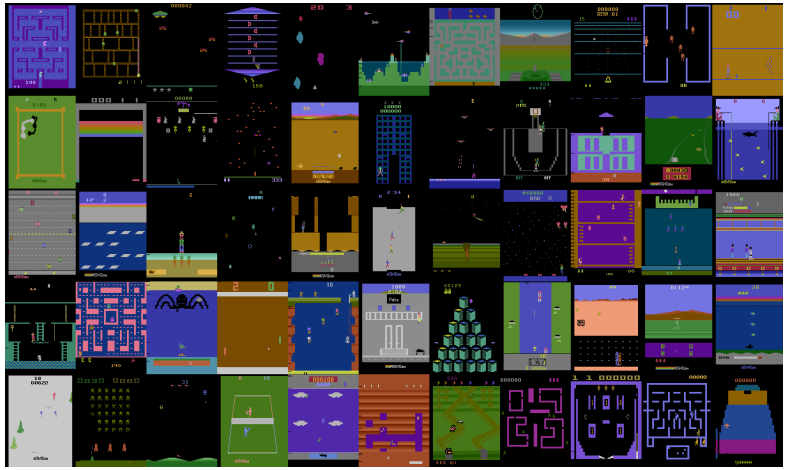
Deep Q-Networks

Use a neural network as the action-value approximator — then stabilize the learning problem.

A game-changing result: DQN from DeepMind

- Then a game-changing result was published in *Nature*:
- **DQN from DeepMind (now Google DeepMind)**
  - Surpassed human performance in many Atari 2600 games.
  - The same RL algorithm learns a policy in each game.
  - End-to-end: only image pixels are used as input.
  - A landmark result that triggered enormous interest in deep reinforcement learning.

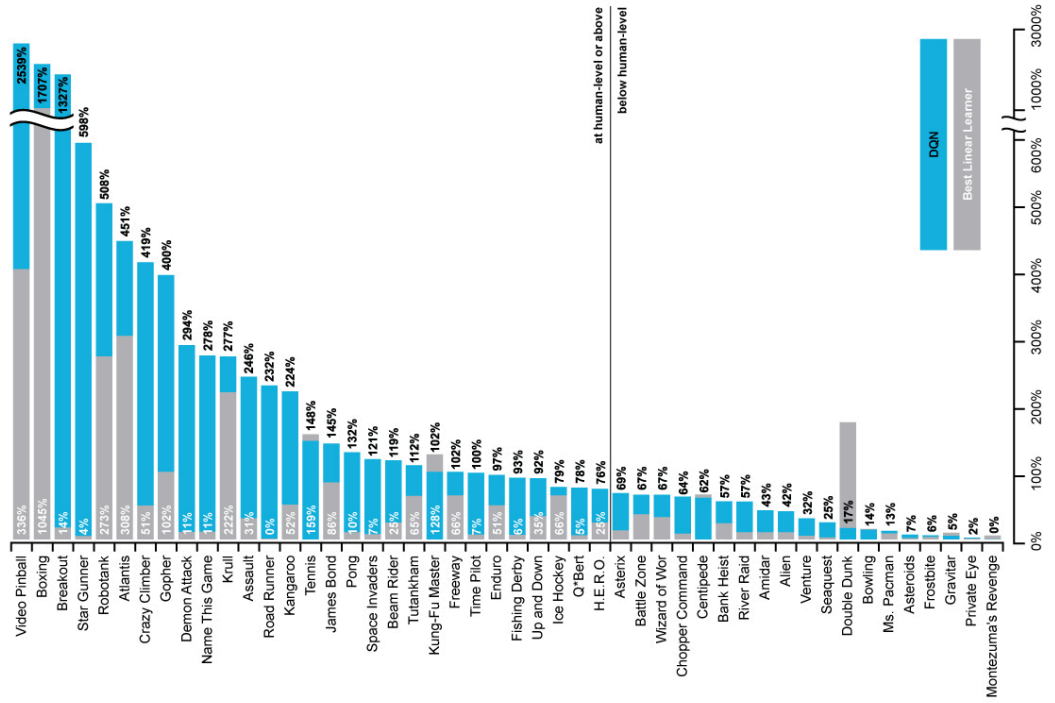




Mnih et al. (2015), Human-level control through deep reinforcement learning, Nature.

DQN from DeepMind: performance across Atari games

One algorithm, one broad architecture, many different games — evaluated against human-normalized performance.



Mnih et al. (2015), Human-level control through deep reinforcement learning, Nature.

How does it work?

- A convolutional neural network reads the game image — typically a stack of the last 4 frames.
- The CNN is a value-function approximator for the action-value function.
- The reward is the game score.
- The network weights are tuned by backpropagating learning signals derived from rewards.



→ CNN learned features → Q-values for all actions

$$\hat{\mathbf{w}}) \approx q_\pi(s, a)$$

Mnih et al. (2015) · David Silver, Reinforcement Learning lecture notes.

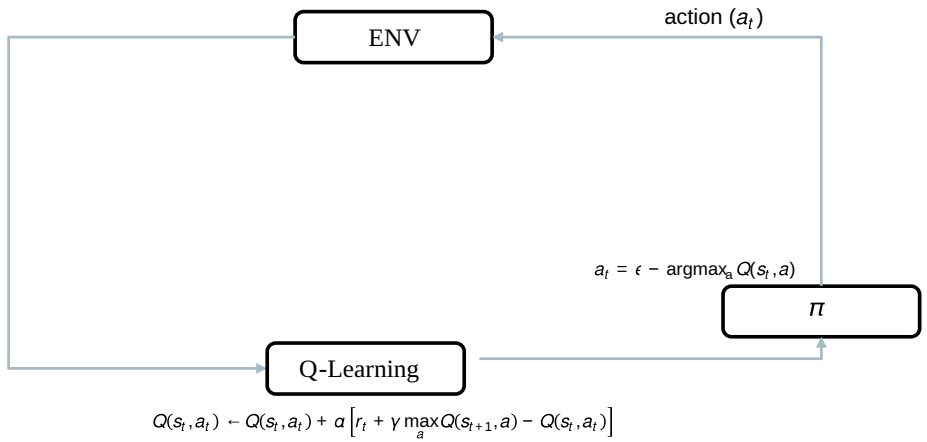
How does it work?

$$L(\mathbf{w}_i) = \mathbb{E}\left[\left(y_i - \hat{Q}(s, a; \mathbf{w}_i)\right)^2\right]$$

	Linear VFA	NN VFA
Approximated Q-function:	$Q(s,a) \approx \mathbf{w}^\mathsf{T}\phi(s,a)$	$Q(s,a) \approx \hat{Q}(s,a;\mathbf{w})$
Target value:	$y_i = r + \gamma \max_{a'} \mathbf{w}_i^\mathsf{T}\phi(s',a')$	$y_i = r + \gamma \max_{a'} \hat{Q}(s',a';\mathbf{w}^-)$
Update:	$\mathbf{w}_{i+1} = \mathbf{w}_i + \alpha\left[y_i - \mathbf{w}_i^\mathsf{T}\phi(s,a)\right]\phi(s,a)$	$\mathbf{w}_{i+1} = \mathbf{w}_i + \alpha\left[y_i - \hat{Q}(s,a;\mathbf{w}_i)\right] \nabla_{\mathbf{w}}\hat{Q}(s,a;\mathbf{w}_i)$

Mnih et al. (2015)

From Q-learning to a neural Q-function



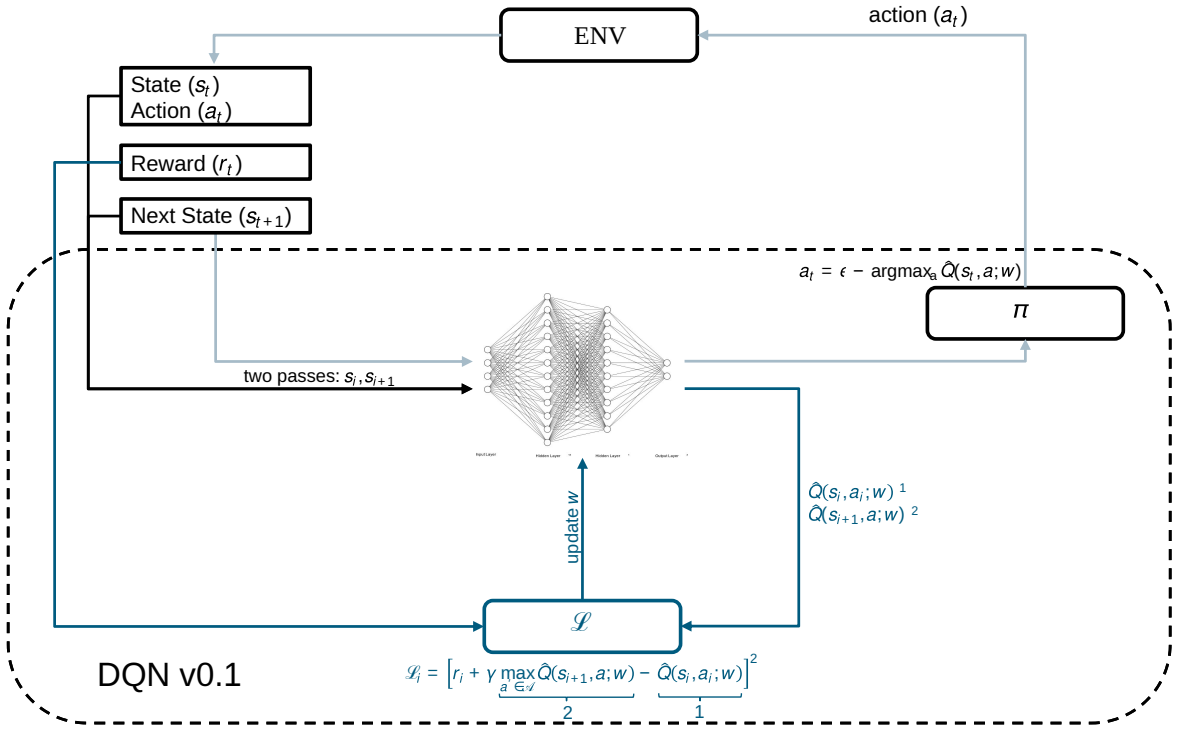
Start from the standard Q-learning control loop

The environment produces the next transition after each action.

The policy π acts ε-greedily with respect to the current Q-function.

Q-learning updates the action value using reward + bootstrap target.

DQN v0.1 · Replace the Q-table by a neural network



The same Q-learning idea, now with parameters w

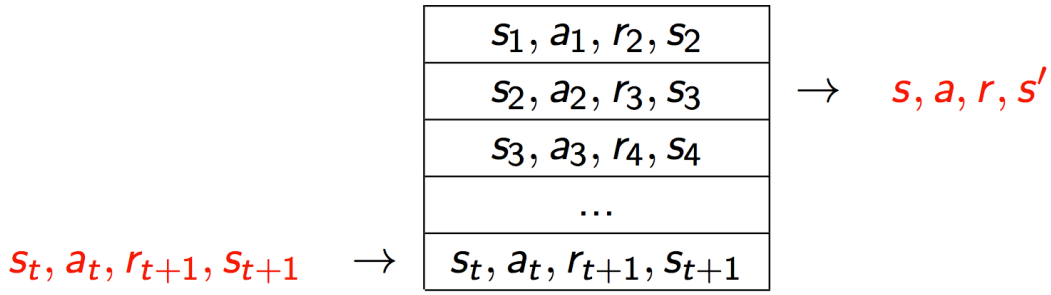
For one transition, evaluate the network at the current state-action pair and at the next state.

The TD loss uses the same network for the prediction and for the bootstrap target.

Backpropagate the loss and update the network parameters w.

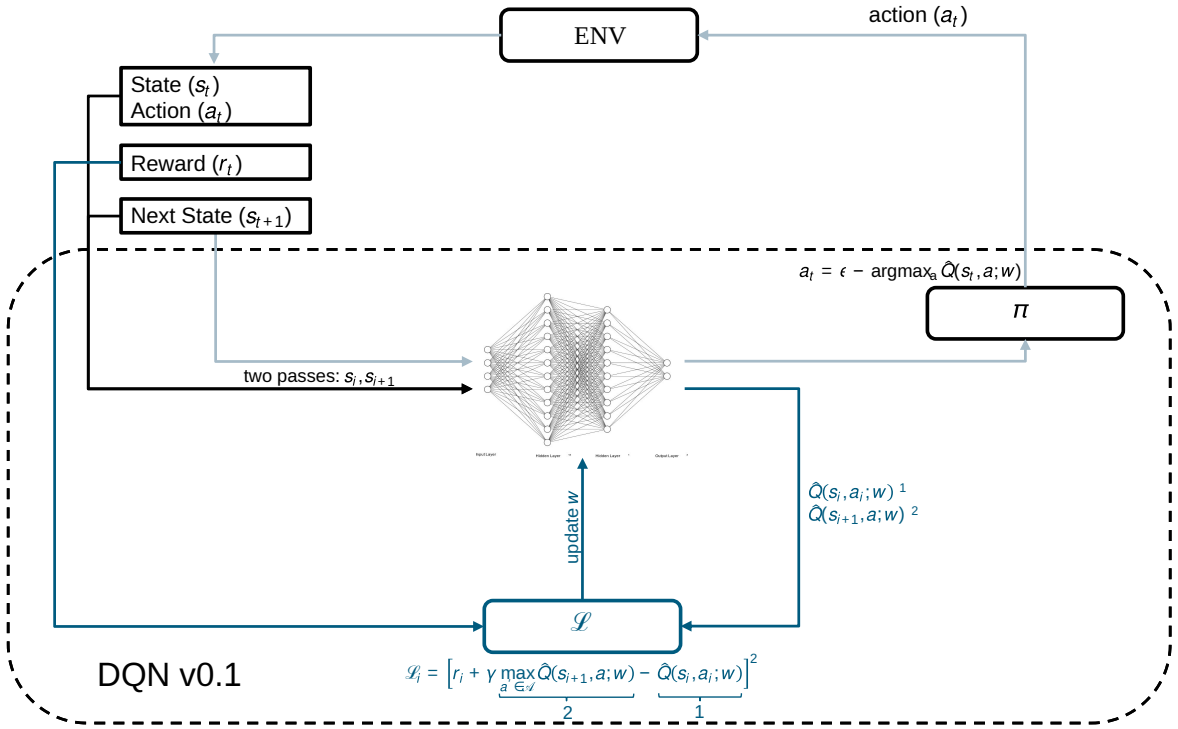
Stabilizer #1 · Experience Replay

- How does it work?
- A **bag of tricks** for stabilizing learning:
  1. Experience Replay:
     - **Problem:** Experiences are correlated over time.
       → *Oscillations* and *divergence* during learning.
     - **Solution:** Random sampling of experience mini-batches from a memory.
       → Samples can be re-used to *increase data efficiency*.
       → Breaking correlations by randomization *reduces variance*.



http://www0.cs.ucl.ac.uk/staff/d.silver/web/Resources_files/deep_rl.pdf

DQN v0.1 · Before experience replay



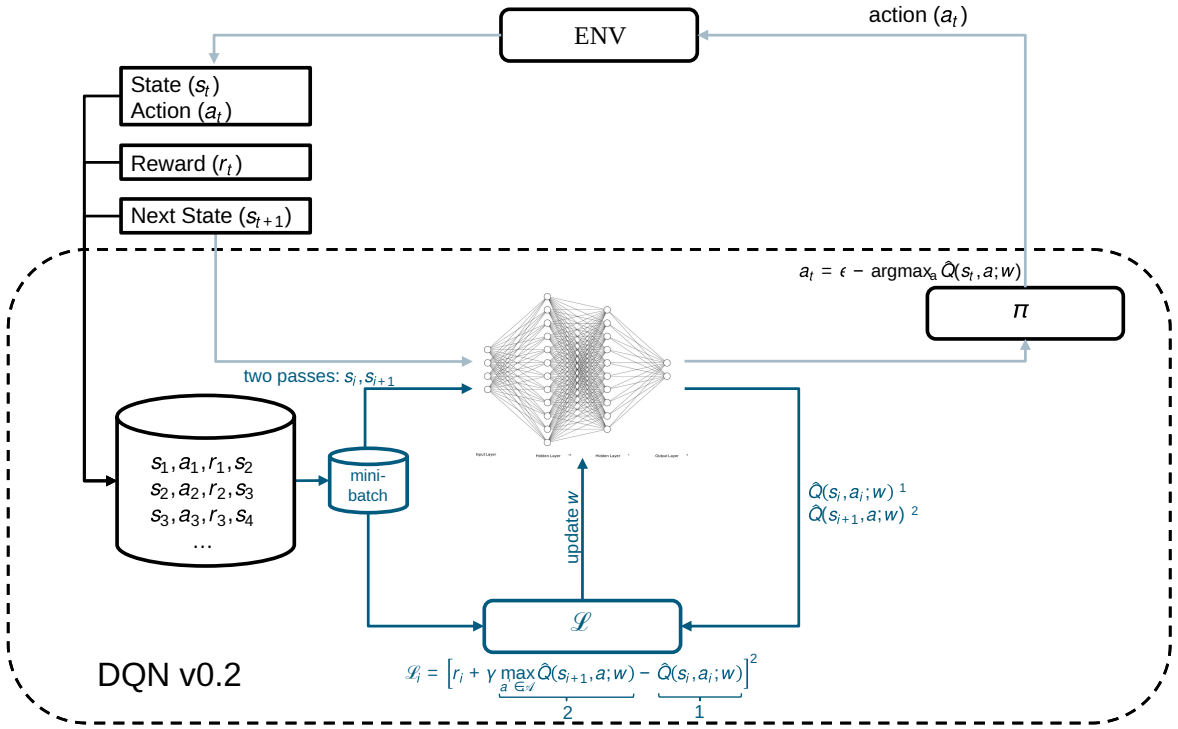
The first neural version still trains on the current stream

Consecutive experiences are strongly correlated.

The same network is used to compute both terms of the TD loss.

Experience replay addresses the first problem.

DQN v0.2 · Add experience replay



Train from random mini-batches sampled from memory

- Store transitions in a replay memory.
- Sample random mini-batches for gradient updates.
- Reuse past experience and break temporal correlations between successive updates.

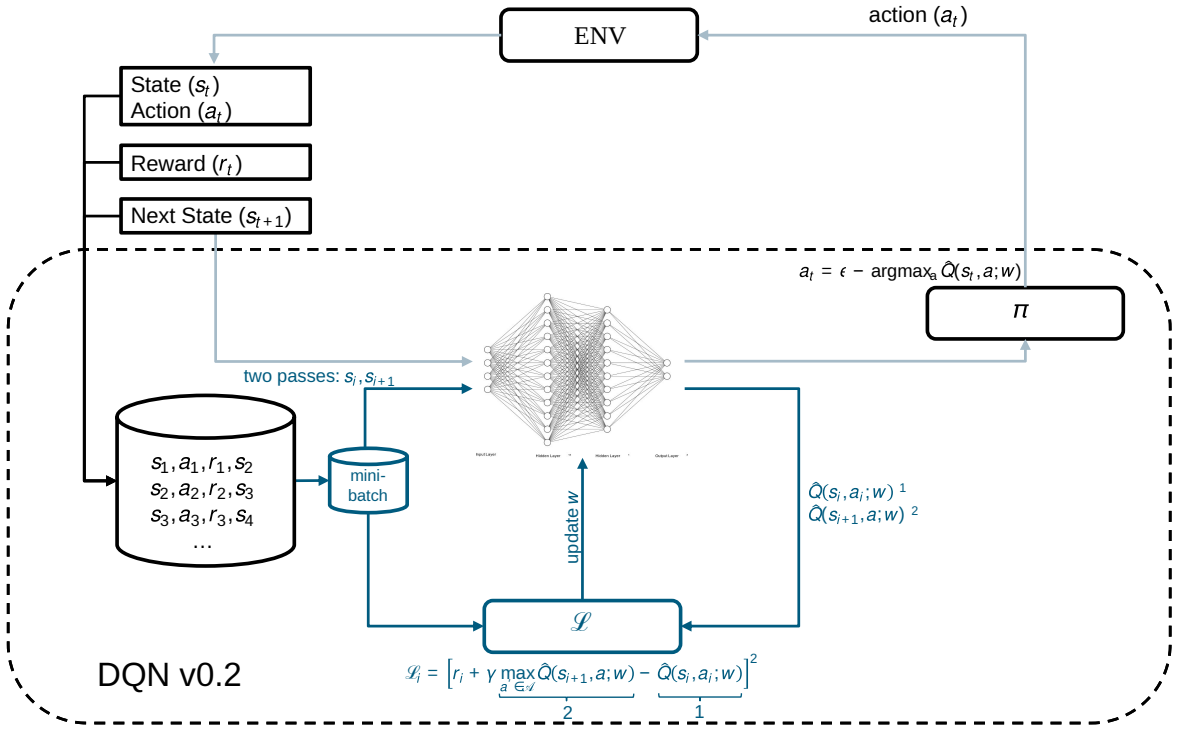
Stabilizer #2 · The target keeps moving

- How does it work?
- A **bag of tricks** for stabilizing learning:
  1. Experience Replay.
  2. Separate, frozen target Q-network:
     - **Problem:** Target Q-values $y_i = r + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s', a'; w_{i-1})$ change constantly.
     - → *Oscillations* and *divergence* during learning.

Stabilizer #2 · Use two Q-networks

- How does it work?
- A **bag of tricks** for stabilizing learning:
  1. Experience Replay.
  2. Separate, frozen target Q-network:
     - **Solution:** Two Q-networks:
     - → Frozen Target Q-network with parameters w^- predicts Q-learning targets $\hat{Q}(s', a'; w_i^-)$.
     - → Dynamic Main Q-network with parameters evaluates current Q-values $\hat{Q}(s', a'; w_{i+1})$.
     - → Perform a gradient descent step (w.r.t.) towards $_i - \hat{Q}(s, a; w_i)^2$
     - Added delay breaks correlations between Q-network and target.
     - *Avoids oscillations* by having fixed targets.
     - *(Note: We periodically update the target Q-network by copying the weights $w^- \leftarrow w_i$.)*
     - *Reduces* chance of *divergence*.

DQN v0.2 · Replay is not enough



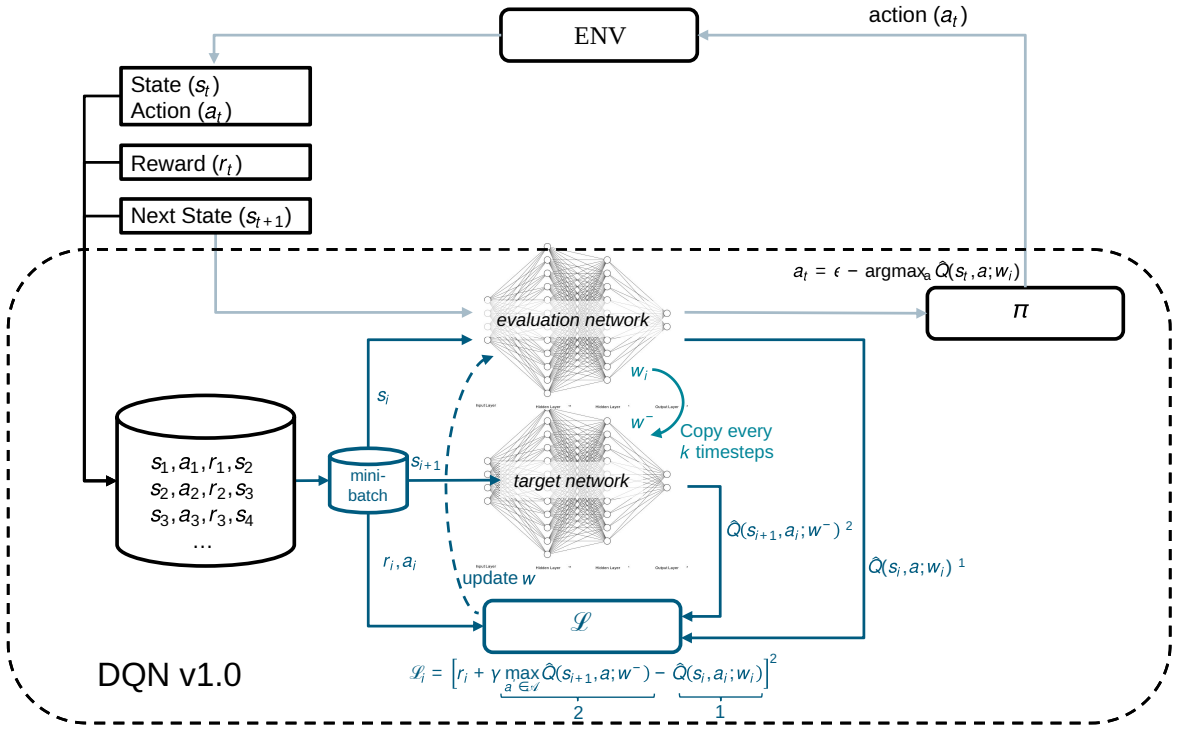
Replay fixes the correlated data stream — but not the moving target

Random mini-batches reduce correlations between training samples.

The online network still generates its own bootstrap target.

Next: freeze a separate target network for several updates.

DQN v1.0 · Add the frozen target network



Replay + target network = the core stabilization recipe

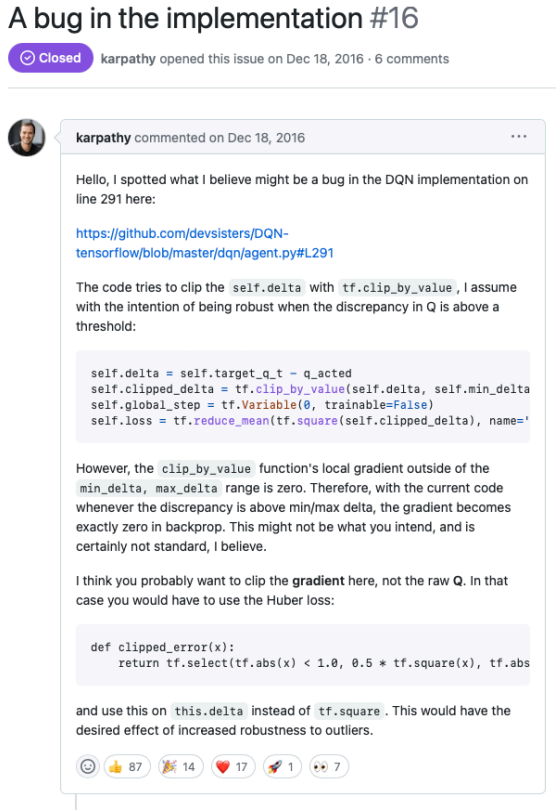
The evaluation network is updated by gradient descent.

The frozen target network supplies the bootstrap target.

Every k timesteps, copy the evaluation-network weights to the target network.

Stabilizer #3 · Reward clipping

- How does it work?
- A **bag of tricks** for stabilizing learning:
  1. Experience Replay.
  2. Separate, frozen target Q-network.
  3. Apply reward clipping:
     - **Problem:** Large rewards result in large variances in Q-values.
     - Different games have different reward values.
     - *Oscillations* and *divergence* during learing.
     - **Solution:** Clip the rewards (and loss terms) to a range [−1.0, 1.0].
     - Avoids *oscillations* by normalizing rewards when training for multiple games.
     - Prevents Q-values from becoming too large.



A bug in the implementation #16

Closed karpathy opened this issue on Dec 18, 2016 · 6 comments

karpathy commented on Dec 18, 2016

Hello, I spotted what I believe might be a bug in the DQN implementation on line 291 here:

https://github.com/devsisters/DQN-tensorflow/blob/master/dqn/agent.py#L291

The code tries to clip the `self.delta` with `tf.clip_by_value`, I assume with the intention of being robust when the discrepancy in Q is above a threshold:

```
self.delta = self.target_q_t - q_acted
self.clipped_delta = tf.clip_by_value(self.delta, self.min_delta
self.global_step = tf.Variable(0, trainable=False)
self.loss = tf.reduce_mean(tf.square(self.clipped_delta), name='
```

However, the `clip_by_value` function's local gradient outside of the `min_delta, max_delta` range is zero. Therefore, with the current code whenever the discrepancy is above min/max delta, the gradient becomes exactly zero in backprop. This might not be what you intend, and is certainly not standard, I believe.

I think you probably want to clip the **gradient** here, not the raw **Q**. In that case you would have to use the Huber loss:

```
def clipped_error(x):
    return tf.select(tf.abs(x) < 1.0, 0.5 * tf.square(x), tf.abs
```

and use this on `this.delta` instead of `tf.square`. This would have the desired effect of increased robustness to outliers.

87 14 17 1 7

EXTENSION #1

Double DQN reduces maximization bias

VANILLA TARGET

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

The max tends to select actions whose estimates are accidentally high.

DOUBLE DQN

$$a^\star = \arg\max_{a'} Q(s', a'; \mathbf{w})$$

$$= r + \hat{Q}(s', a^\star; \mathbf{w}^-)$$

online selects **target evaluates**

EXTENSION #2

Prioritized Replay spends updates where the Bellman error is large

$$= y_i - \hat{Q}(s_i, a_i; \mathbf{w})$$

SAMPLING PRIORITY

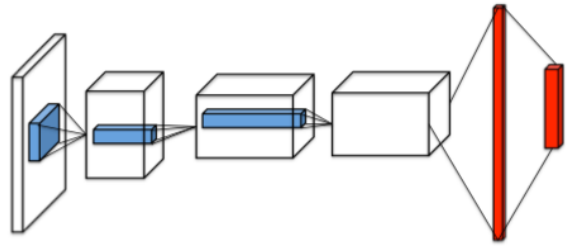
$$\propto (|\delta_i| + \varepsilon)^{\alpha}$$

Transitions with surprising targets are replayed more often.

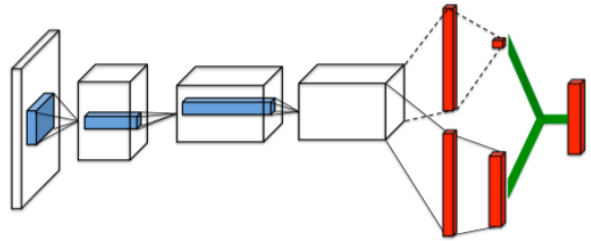
Importance-sampling weights can compensate for the bias introduced by non-uniform sampling.

EXTENSION #3

Dueling networks separate state value from action advantage



single stream



dueling architecture

$$= V(s) + A(s,a) - \text{mean}_{a'} A(s,a')$$

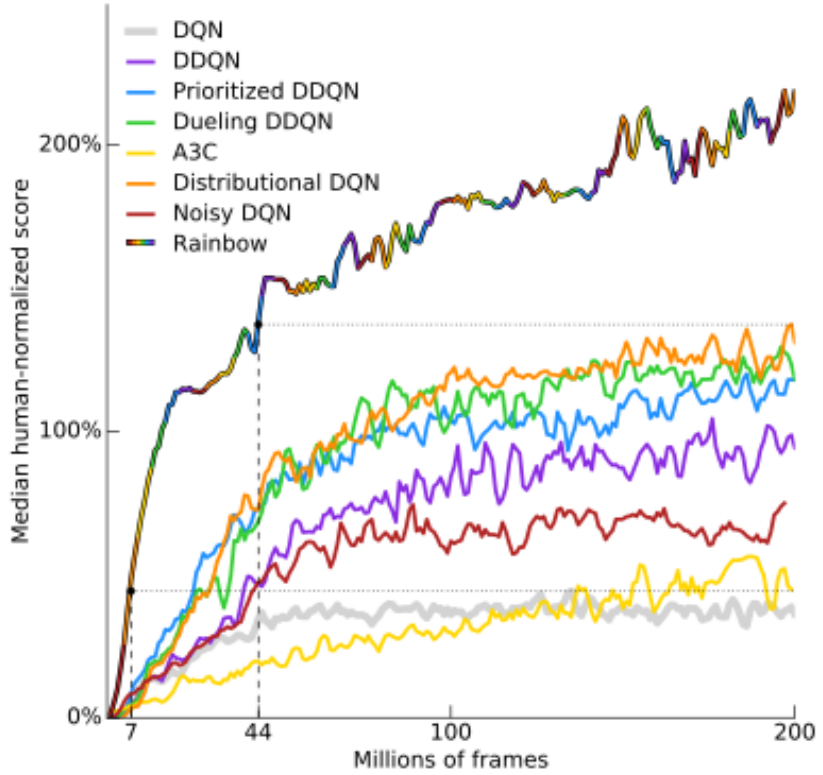
INTUITION

Learn whether a state is good even when the exact action matters little.

RAINBOW

Later work combined Double DQN, prioritized replay, dueling networks and other improvements into one agent.

Rainbow combines several DQN improvements in one agent



Hessel et al. (2018), Rainbow: Combining Improvements in Deep Reinforcement Learning. AAAI.

THE DEADLY TRIAD

Three ingredients make instability especially likely

1

Function approximation
shared parameters couple values

2

Bootstrapping
targets use current estimates

3

Off-policy learning
behavior and target policies differ

DQN uses all three — so its stabilizers are not cosmetic.

TAKEAWAYS

The conceptual ladder



1. **Function approximation** trades exact storage for generalization.
2. **Fitted Q-Iteration** makes Bellman backups look like supervised labels.
3. **DQN** uses a neural Q-function, but needs a better optimization schedule.
4. **Experience replay** decorrelates updates and reuses data.
5. **Target networks** slow down the movement of Bellman labels.

LESSON OF TODAY

“Be careful with (non-linear) function approximation.”

generalization ↔ interference ↔ stability tricks

Powerful representations are useful precisely because one update can change many predictions. That is also why they can destabilize learning.



REFERENCES

Key sources

- Sutton & Barto (2018). *Reinforcement Learning: An Introduction*, 2nd ed.
- Ernst, Geurts & Wehenkel (2005). Tree-Based Batch Mode Reinforcement Learning — Fitted Q-Iteration.
- Mnih et al. (2015). Human-level control through deep reinforcement learning — DQN.
- van Hasselt, Guez & Silver (2016). Deep Reinforcement Learning with Double Q-learning.
- Schaul et al. (2016). Prioritized Experience Replay.
- Wang et al. (2016). Dueling Network Architectures for Deep Reinforcement Learning.
- Hessel et al. (2018). Rainbow: Combining Improvements in Deep Reinforcement Learning.
- Thierry & Scherrer (2010). Improvements on Learning Tetris with Cross-Entropy.
- Tziortziotis, Tziortziotis & Blekas (2014). Play Ms. Pac-Man Using an Advanced Reinforcement Learning Agent.
- Lagoudakis & Parr (2003). Least-Squares Policy Iteration. JMLR.