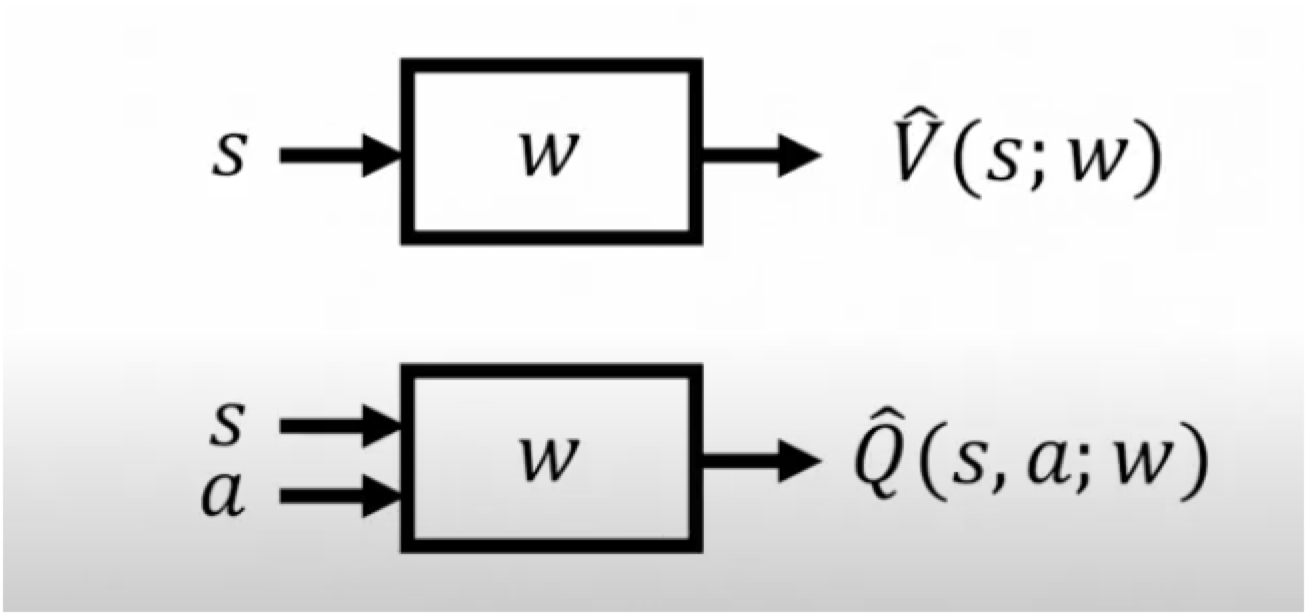
Linear VFA

Why do we need approximations?

1. The state space can be ridiculously large in real world.
2. Tabular methods have limited generalization.
   Even if we end up in a state-action pair that we have not seen exactly before, we are still able to make good decisions(estimate state or Q-value).

Value Function Approximation (VFA)

Our goal is to represent a (state-action/state) value function with a parametrized function instead of a table



Benefits of VFA include:

- Reduce memory needed to store $(P, R)/V, Q, \pi$
- Reduce computation needed to compute $(P, R)/V/Q/\pi$
- Reduce experience needed to find a good $(P, R)/V/Q/\pi$

Linear vs Non-linear

One has to be careful when talking about linearity in **feature space** and **model weight space**.
The function approximator can be linear (weighted combination of features) or non-linear(neural nets with activation functions).
In addition, the features themselves can be linear (polynomials) or non-linear(fourier basis, RBF kernel).

Function approximators

Many possible function approximators including: linear combination of features, neural nets, decision trees, NNs, fourier/wavelet bases
We mainly focus on **linear feature representations** and **neural networks**.

Feature Vectors

We need to build a feature vector to represent a state s:

$$x(s) = (x_1(s), x_2(s), \dots, x_n(s))^T$$

For building feature vectors (that can be linear or non-linear)

VFA with an Oracle

Represent a value function for a particular policy with a **weighted** linear combination of features:

$$\tilde{V}(s; w) = \sum_{j=1}^{n} x_j(s) w_j = x(s)^T w$$

Then, our objective function is:

$$J(w) = E_\pi[(V^\pi(s) - \tilde{V}(s; w))^2]$$
$$= E_\pi[(V^\pi(s) - x(s)^T w))^2]$$

Recall weight update is:

$$\Delta w = -\frac{1}{2} \alpha \Delta_w J(w)$$

Substituting the gradient of the objective function, one obtains:

$$\Delta w = -\frac{1}{2} \alpha (2 * x(s) w (V^\pi(s) - x(s)^T))$$

So, the update is then equal to :

$$update = stepsize * feature_{val} prediction_{err}$$

VFA without an Oracle

Of course, we don't have access to true value function. But we can use the same tricks we applied in tabular learning in this setting.
Monte Carlo with linear VFA
Episodic settings
The true value functions is the return G_t. Remember it is an unbiased but *noisy* sample of the true expected return $V^\pi(s_t)$. Therefore, we can reduce MC VFA to doing supervised learning on a set of (state, return) pairs: $(s_1, G_1), \dots (s_t, G_t)$
Concretely, when using linear VFA for policy evaluation:

$$\Delta w = \alpha (G_t - \tilde{V}(s_t; w)) \Delta_w \tilde{V}(s_t; w)$$

$$= \alpha(G_t - \tilde{V}(s_t; w))x(s_t)$$

$$= \alpha(G_t - x(s_t)^T w)x(s_t)$$

Note that G_t may be a very noisy estimate of the true return.
TD Learning with linear VFA
For TD(0):

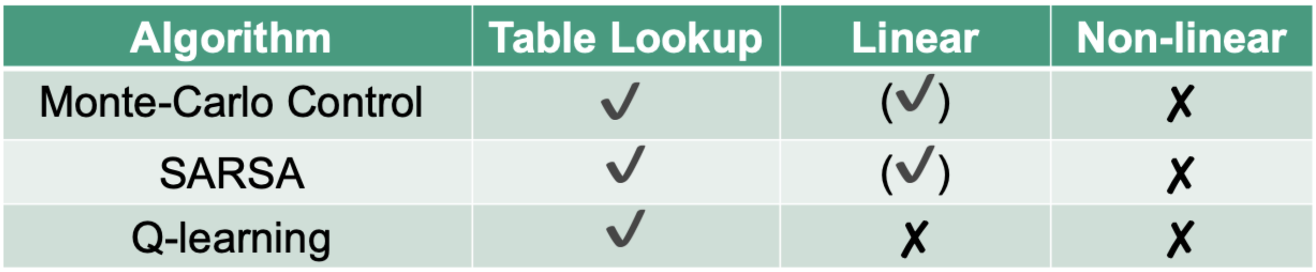
$$\Delta w = \alpha(R_{t+1} + \gamma\tilde{V}(s_{t+1}; w) - \tilde{V}(s_t; w))\Delta_w \tilde{V}(s_t; w)$$

Target is $R_{t+1} + \gamma\tilde{V}^\pi(s_{t+1})$ is a biased estimate of the true value for state s.

SARSA with linear VFA
The true value function in this case can be replaced with $R_t + \gamma * Q(s_{t+1}, a_{t+1}; w)$
And the update rule changes accordingly
Q-learning with linear VFA
Similarly, the true value function can be replaced with $R_t + \gamma * max_a Q(s_{t+1}, a_{t+1}; w)$

Convergence of Control Algorithms

Non-linear approximations do not guarantee convergence properties. In fact, **Q-learning** values could diverge over time. Generally, for **off-policy** control/prediction algorithms, there is not guarantee on convergence.
TD does not follow the gradient of any objective function, divergence possible with both linear and non-linear function approximation



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

1. Any V-/Q- value function can be approximated with a linear FA
2. Finding such linear FA is **not** easy. Requires a lot of feature engineering.

Why not use Neural Networks (non-linear approximators)? In theory, they don't converge, but, in some applications they do! For example, TD-Gammon paper: increasing the hidden units in NNs correlate with higher performance in the game
Hot Topic:
Off Policy function approximation convergence.
Exciting recent work on batch RL that can converge with nonlinear VFA: uses primal dual optimization. An important issue is not just whether the algorithm converges, but **what** solution it converges too.

Non-linear VFA

An alternative to hand designing feature set is to use much richer function approximation class that is able to directly go from states without requiring an explicit specification of features.
Kernel-based approaches have some appealing properties(including convergence under certain cases) but can't typically scale well to enormous spaces and datasets
Here comes to Neural Networks to rescue!

Deep Neural Networks (DNN)

They combine both linear and non-linear transformations:

- Linear: MLP weight multiplication
- Non-linear: activation functions in nodes(sigmoid, ReLU)
- Universal function approximator

DQN

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

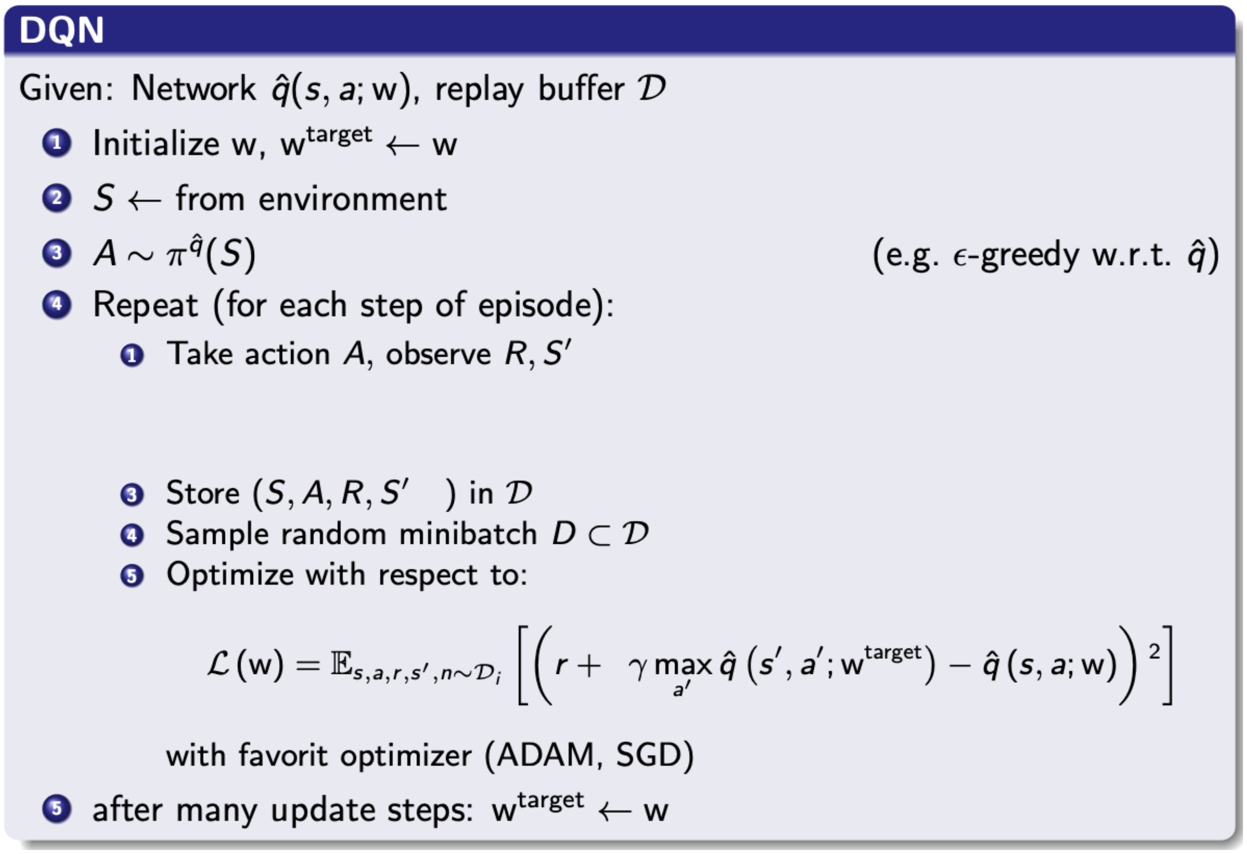
Two **problems** that come up in Q-learning with VFA that leads to divergence:
Problem 0: Coverage: convergence proof of Value Iteration, Q-learning and SARSA depend on covering the entire state space (at least for linear VFA). In high dimensional problems, this is hard to achieve. Therefore, use ϵ greedy exploration
Problem.1 Correlations between samples (non-iid)
Solution.1: to help remove correlations, store dataset in a replay buffer D from prior experience. To perform experience replay, repeat the following:

- $(s,a,r,s') \sim D$: sample a tuple from the replay buffer
- Computer the target value for the sampled $s: r + \gamma max_{a'}\tilde{Q}(s',a';w)$
- Use SGD to update network weights

$$\Delta w = \alpha(R_t + \gamma max_{a_{t+1}}\tilde{Q}(s_{t+1},a_{t+1};w) - \tilde{Q}(s,a;w))\Delta_w\tilde{Q}(s_t,a_t;w)$$

Problem.2 Non-stationary targets
Solution.2: to improve stability, fix the **target weights** used in the target calculation for multiple updates
Keep w^{target} fixed for some time(old value of w). Then, TD error is computed using

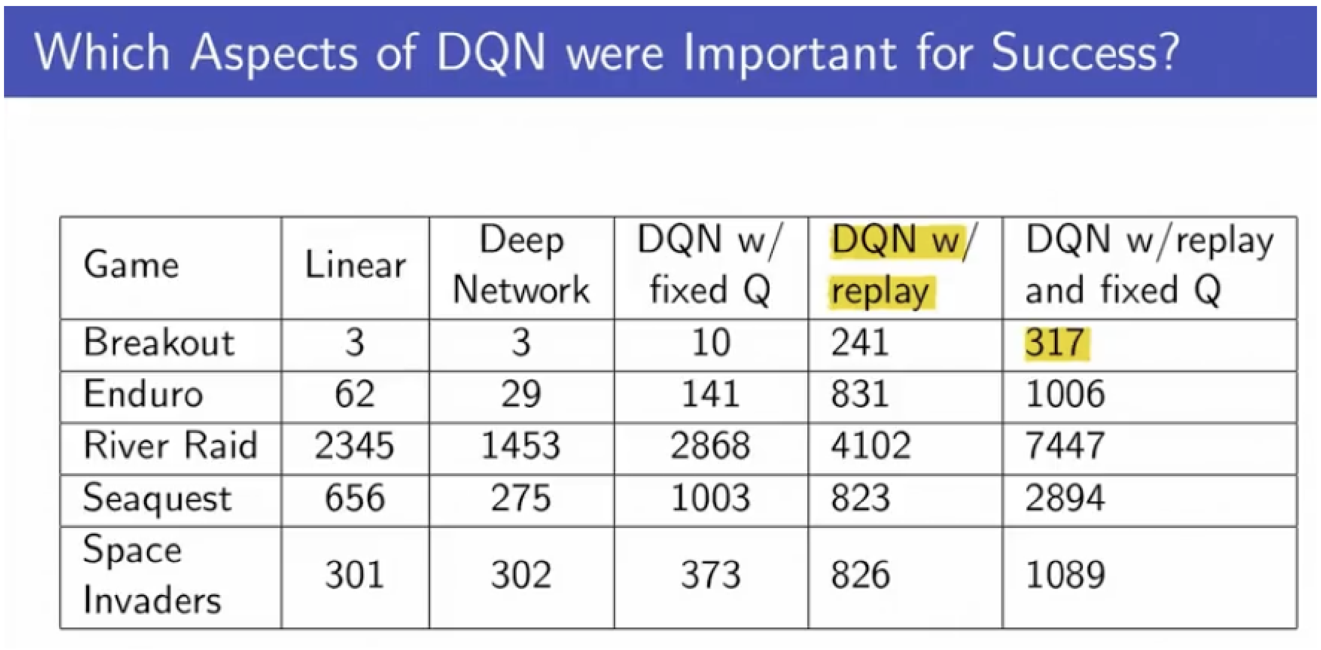
w^{target}



DQN

```
Given: Network $\hat{q}(s,a;\mathrm{w})$, replay buffer $\mathcal{D}$
  1. Initialize w, $\mathrm{w}^{\text{target}} \leftarrow \mathrm{w}$
  2. $S \leftarrow$ from environment
  3. $A \sim \pi^{\hat{q}}(S)$                                  (e.g. $\epsilon$-greedy w.r.t. $\hat{q}$)
  4. Repeat (for each step of episode):
      1. Take action $A$, observe $R, S'$
      3. Store $(S, A, R, S' \quad)$ in $\mathcal{D}$
      4. Sample random minibatch $D \subset \mathcal{D}$
      5. Optimize with respect to:
         $$\mathcal{L}(\mathrm{w}) = \mathbb{E}_{s,a,r,s',n\sim\mathcal{D}_i}\left[\left(r + \quad \gamma \max_{a'} \hat{q}\left(s', a'; \mathrm{w}^{\text{target}}\right) - \hat{q}\left(s, a; \mathrm{w}\right)\right)^2\right]$$
         with favorit optimizer (ADAM, SGD)
  5. after many update steps: $\mathrm{w}^{\text{target}} \leftarrow \mathrm{w}$
```

Another trick: reward clipping
Different games have different reward values and this may lead to oscillations and divergence during learning. Therefore, clip the rewards to $[-1, 1]$ range. This prevents Q-values become too large
Experience replay is hugely important.



Which Aspects of DQN were Important for Success?

Game	Linear	Deep Network	DQN w/ fixed Q	DQN w/ replay	DQN w/replay and fixed Q
Breakout	3	3	10	241	317
Enduro	62	29	141	831	1006
River Raid	2345	1453	2868	4102	7447
Seaquest	656	275	1003	823	2894
Space Invaders	301	302	373	826	1089

Extensions

Double DQN
Recall max operator bias in Q-learning. This happens in DQN as well.

Double DQN splits action selection and action evaluation:

$$y_i = r + \gamma\tilde{Q}(s', argmax\tilde{Q}(s', a'; \theta_i); v_i)$$

Use estimations from main Q-network to **select** actions
Use estimations from target Q-network to **evaluate** actions

```
1. Initialize $Q^A$ and $Q^B$, s
2. Repeat for each episode:
   2.1. Choose a, based on $Q^A(s,*)$ and $Q^B(s,*)$, observe $r, s'$
   2.2 Choose randomly either UPDATE(A) or UPDATE(B):
   if UPDATE(A):
   Define $a^* = argmax_a(Q^A(s,a))$
   $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \gamma Q^B(s', a^*) - Q^A(s,a))$
   else if UPDATE(B):
   Define $b^* = argmax_a Q^B(s,a)$
   $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s', b^*) - Q^B(s,a))$
   end if
   $s \leftarrow s'$
```

Prioritized Experience Replay
Some experiences retain more information for learning than others. Vanilla experience sampling is **uniform**. Instead, use PER which samples mini-batches based on their absolute Bellman error e. Leads to faster convergence

$$\delta = r + \gamma max_{a \in A} Q(s', a') - Q(s,a)$$

$$e = |\delta|$$

Using DDQN notation:

$$y_i = r + \gamma\tilde{Q}(s', argmax_{a' \in A}\tilde{Q}(s', a'; \theta_i); v_i)$$

$$\delta = y_i - Q(s, a; \theta_i)$$

$$e = |\delta|$$

Dueling DQN
Split Q-network into two channels:

1. Action-independent value function $V(s; v)$
2. Action-dependent advantage function $A(s, a : w)$
   Then, $Q(s,a) = V(s,v) + A(s,a;w)$

Deadly Triad

Instability and divergence in RL stem from:

1. Function approximation

2. Bootstrapping
3. Off-policy training