Function Approximation Methods

CS60077: Reinforcement Learning

Abir Das

IIT Kharagpur

Oct 21, 22, 2021

Agenda

- Get started with the function approximation methods. Revisit risk minimization, gradient descent *etc.* from Machine Learning class.
- Get familiar with different on and off policy evaluation and control methods with function approximation.

Resources

- Reinforcement Learning by Balaraman Ravindran [Link]
- Reinforcement Learning by David Silver [Link]
- SB: Chapter 9

Tabular Methods

- In the last few lectures we have seen '*tabular methods*' for solving RL problems.
- The state or action values are all represented by look-up tables. Either every state s or every state-action pair (s, a) has an entry in the form of $V(s)$ or $Q(s, a)$.
- The approach, in general, updates *old* values in these tables towards a *target* value in each iteration.
- Let the notation $s \mapsto u$ denote an individual update where s is the state updated and u is the target.

Tabular Methods

- In the last few lectures we have seen '*tabular methods*' for solving RL problems.
- The state or action values are all represented by look-up tables. Either every state s or every state-action pair (s, a) has an entry in the form of $V(s)$ or $Q(s, a)$.
- The approach, in general, updates *old* values in these tables towards a *target* value in each iteration.
- Let the notation $s \mapsto u$ denote an individual update where s is the state updated and u is the target.
  - Monte Carlo update is $s_t \mapsto G_t$
  - TD(0) update is $s_t \mapsto R_{t+1} + \gamma V(s_{t+1})$
  - DP update is $s_t \mapsto \sum_{a \in \mathcal{A}} \pi(a|s_t) \left\{ R(s_t, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s_t, a) v(s') \right\}$

Tabular Methods - Limitations

§ What are some problems of tabular methods?

Tabular Methods - Limitations

- What are some problems of tabular methods?
- Large state spaces and that means huge memory need and also huge amount of time to update each state enough number of times.

Tabular Methods - Limitations

- What are some problems of tabular methods?
- Large state spaces and that means huge memory need and also huge amount of time to update each state enough number of times.
- This also means, continuous states and actions can not be handled.

Tabular Methods - Limitations

- What are some problems of tabular methods?
- Large state spaces and that means huge memory need and also huge amount of time to update each state enough number of times.
- This also means, continuous states and actions can not be handled.
- States not encountered previously will not have a sensible policy *i.e.*, *generalisation* is an issue.

Tabular Methods - Limitations

- What are some problems of tabular methods?
- Large state spaces and that means huge memory need and also huge amount of time to update each state enough number of times.
- This also means, continuous states and actions can not be handled.
- States not encountered previously will not have a sensible policy *i.e.*, *generalisation* is an issue.
- Fortunately, a change of representation can address all these issues to some extent. Instead of representing the value functions as look-up tables, they are represented as a parameterized functions.
- We denote the approximate value function in parameterized form as $\hat{v}(s; \mathbf{w}) \approx v_\pi(s)$ where $\mathbf{w} \in \mathbb{R}^d$ is the learnable parameter vector.
- Any form of function approximator *e.g.*, linear function approximator, multi-layer neural networks, decision trees, nearest neighbours *etc.* can be used. However, in practice, some fit more easily into this role than others.

Advantages and Specialities of Function Approximators

- Handle large state spaces.
- In tabular methods, learned values at each state are decoupled - an update at one state affected no other state. Parameterized value function representation means an update at one state affects many others helping *generalization*.

Advantages and Specialities of Function Approximators

- Handle large state spaces.
- In tabular methods, learned values at each state are decoupled - an update at one state affected no other state. Parameterized value function representation means an update at one state affects many others helping *generalization*.
- In supervised machine learning teminology, an update of the form $s \mapsto u$ means training with input-output pairs where the output is u.

Advantages and Specialities of Function Approximators

- Handle large state spaces.
- In tabular methods, learned values at each state are decoupled - an update at one state affected no other state. Parameterized value function representation means an update at one state affects many others helping *generalization*.
- In supervised machine learning teminology, an update of the form $s \mapsto u$ means training with input-output pairs where the output is u.
- Viewing each update as conventional training example has some downsides too.
- The training set is not static unlike most traditional supervised learning setting. RL generally requires function approximation methods able to handle nonstationary target functions (target functions that change over time)
- Control methods changes the policy and thus the generated data. For evaluation even, the target values of training examples are non-stationary when bootstrapping is applied.

Types of Value Function Approximators

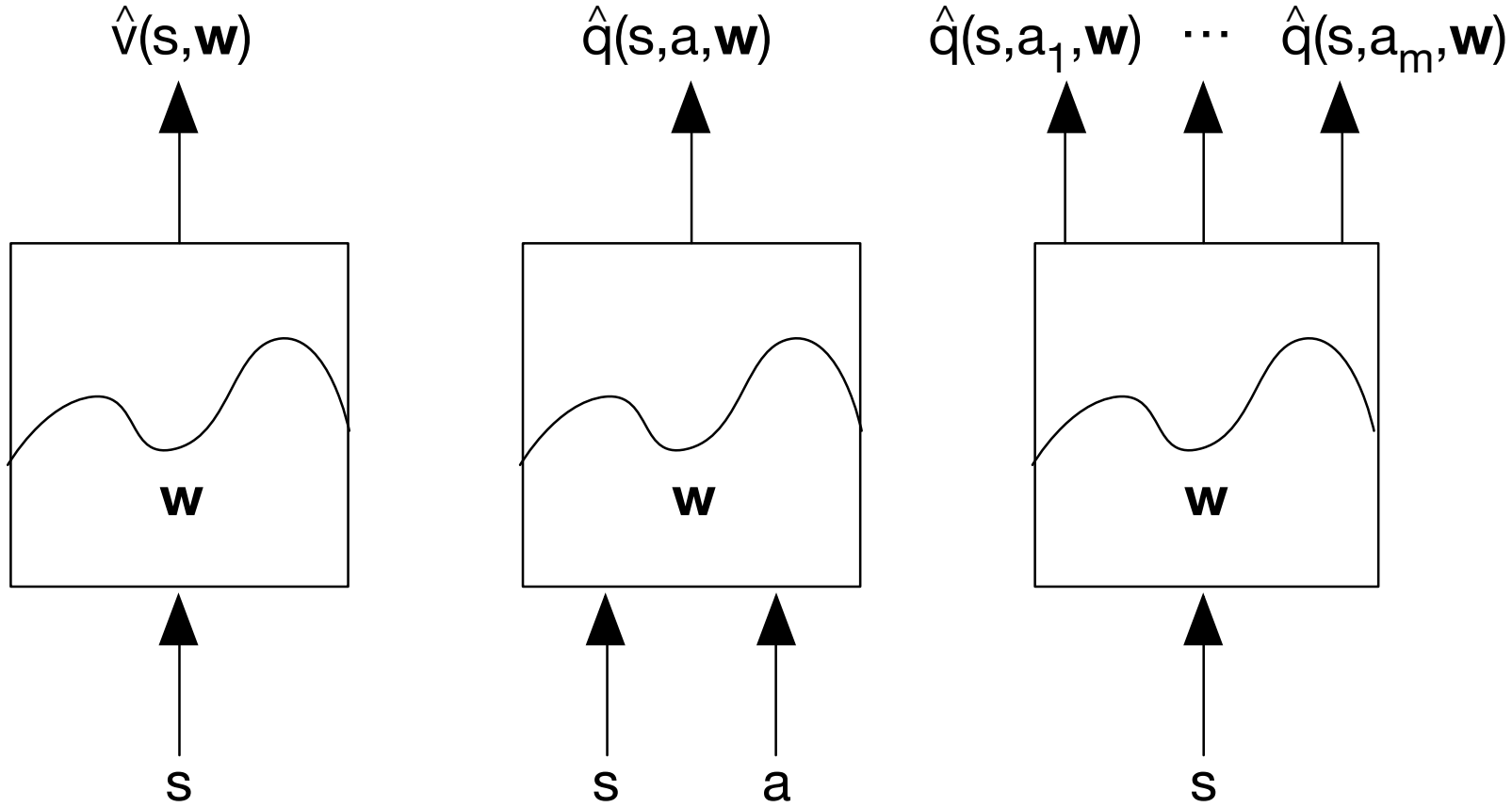


Figure credit: *David Silver, DeepMind*

Error Measures in Function Approximators

- We need to figure out a measure for the error that the function approximator is making.
- One candidate can be the sum of error squares.

$$\sum_{s} [v_{\pi}(s) - \hat{v}(s; \mathbf{w})]^2$$

Error Measures in Function Approximators

- We need to figure out a measure for the error that the function approximator is making.
- One candidate can be the sum of error squares.

$$\sum_{s} [v_{\pi}(s) - \hat{v}(s; \mathbf{w})]^2$$

- Do we average?

Error Measures in Function Approximators

- We need to figure out a measure for the error that the function approximator is making.
- One candidate can be the sum of error squares.

$$\sum_s [v_\pi(s) - \hat{v}(s; \mathbf{w})]^2$$

- Do we average?
- In classical supervised setting, the averaging by $\frac{1}{N}$ is to approximate the *expected* error by *empirical* error.
- With training data $\{s^{(i)}, v_\pi(s^{(i)})\}$ coming from probability distribution $\mathcal{D}(s, v_\pi(s))$,
  - Expected error: $\int l(\hat{v}(s; \mathbf{w}), v_\pi(s)) d\mathcal{D}(s, v_\pi(s))$
  - Empirical error: $\frac{1}{N} \sum_{i=1}^{N} l(\hat{v}(s^{(i)}; \mathbf{w}), v_\pi(s^{(i)}))$

Error Measures in Function Approximators

- This was assuming the training data that we get, follows the true data distribution. So, if a data point is more likely in the true distribution, it is more likely that it comes more often as the training example.
- So, we need to average but with a *state distribution*.

$$\sum_{s} \mu(s) \left[v_{\pi}(s) - \hat{v}(s; \mathbf{w}) \right]^2$$

Error Measures in Function Approximators

- This was assuming the training data that we get, follows the true data distribution. So, if a data point is more likely in the true distribution, it is more likely that it comes more often as the training example.
- So, we need to average but with a *state distribution*.

$$\sum_s \mu(s) \left[v_\pi(s) - \hat{v}(s; \mathbf{w})\right]^2$$

- $\mu(s) > 0, \sum_s \mu(s) = 1$ provides the probability of obtaining a state any time we sample a MRP (why MRP?)

Error Measures in Function Approximators

- This was assuming the training data that we get, follows the true data distribution. So, if a data point is more likely in the true distribution, it is more likely that it comes more often as the training example.
- So, we need to average but with a *state distribution*.

$$\sum_s \mu(s) \left[v_\pi(s) - \hat{v}(s; \mathbf{w})\right]^2$$

- $\mu(s) > 0, \sum_s \mu(s) = 1$ provides the probability of obtaining a state any time we sample a MRP (why MRP?)
- In a non-episodic task, the distribution is the stationary distribution under π
- In an episodic task, $\mu(s)$ is the fraction of time spent in s. A simple proof is given in [SB] - section 9.2. Under on-policy training, this is called the *on-policy distribution*

Error Measures in Function Approximators

- This essentially tells that for the states that are visited more often under the policy π, the approximation will be good.
- For on-policy sampling, the following will approximate the error measure described in the previous slide.

$$\frac{1}{N}\sum_{t=1}^{N}\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w})\right]^2$$

Error Measures in Function Approximators

- This essentially tells that for the states that are visited more often under the policy π, the approximation will be good.
- For on-policy sampling, the following will approximate the error measure described in the previous slide.

$$\frac{1}{N}\sum_{t=1}^{N}\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w})\right]^2$$

- But what is the practical problem in it? Hint: A big assumption was made.

Error Measures in Function Approximators

- This essentially tells that for the states that are visited more often under the policy π, the approximation will be good.
- For on-policy sampling, the following will approximate the error measure described in the previous slide.

$$\frac{1}{N}\sum_{t=1}^{N}\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w})\right]^2$$

- But what is the practical problem in it? Hint: A big assumption was made. - Where will I get the target $v_\pi(s_t)$?

Error Measures in Function Approximators

- This essentially tells that for the states that are visited more often under the policy π, the approximation will be good.
- For on-policy sampling, the following will approximate the error measure described in the previous slide.

$$\frac{1}{N}\sum_{t=1}^{N}\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w})\right]^2$$

- But what is the practical problem in it? Hint: A big assumption was made. - Where will I get the target $v_\pi(s_t)$?
- We will come back to this.

Gradient Descent Primer

- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$.
- The gradient of $J(\mathbf{w})$ is defined as,

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_d} \end{pmatrix}$$

- To find local minimum of $J(\mathbf{w})$, $\mathbf{w}$ is adjusted in the direction of negative gradient.

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}),$$

  where α is the step size parameter.
- Stochastic Gradient Descent (SGD) adjusts the weight vector after each example by computing gradient of J only for that example.

Gradient Monte Carlo Algorithm

§ SGD update rule for mean square error.

$$\begin{aligned}\mathbf{w}_{k+1} &= \mathbf{w}_k - \frac{1}{2}\alpha\nabla_{\mathbf{w}_k}\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w}_k)\right]^2 \\ &= \mathbf{w}_k + \alpha\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w}_k)\right]\nabla_{\mathbf{w}_k}\hat{v}(s_t;\mathbf{w}_k) \qquad (1)\end{aligned}$$

§ Lets look at different possibilities to approximate the target $v_\pi(s_t)$.

- Monte Carlo target: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$
- TD(0) target: $R_{t+1} + \gamma\hat{v}(s_{t+1};\mathbf{w}_k)$

Gradient Monte Carlo Algorithm

§ SGD update rule for mean square error.

$$\begin{aligned}\mathbf{w}_{k+1} &= \mathbf{w}_k - \frac{1}{2}\alpha\nabla_{\mathbf{w}_k}\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w}_k)\right]^2 \\ &= \mathbf{w}_k + \alpha\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w}_k)\right]\nabla_{\mathbf{w}_k}\hat{v}(s_t;\mathbf{w}_k) \qquad (1)\end{aligned}$$

§ Lets look at different possibilities to approximate the target $v_\pi(s_t)$.

- Monte Carlo target: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$
- TD(0) target: $R_{t+1} + \gamma\hat{v}(s_{t+1};\mathbf{w}_k)$

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left[G_t - \hat{v}(S_t,\mathbf{w})\right]\nabla\hat{v}(S_t,\mathbf{w})$

Figure credit: *[SB: Chapter 9]*

Data Preparation

- Traditionally supervised methods gets data in the form $< \{x^{(i)}, f(x^{(i)})\} >$, from where a function $\hat{f}(x)$ to predict for an unknown x is learned.
- In RL, data comes in the form of $< s_t, a_t, R_{t+1}, s_{t+1}, a_{t+1}, R_{t+2} + \cdots >$ from where the approximate targets have to be generated.
- For MC targets, this is done as $< s_t, R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots >$, $< s_{t+1}, R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots >$ *etc.*
- For TD(0) targets, this is done as $< s_t, R_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_k) >$, $< s_{t+1}, R_{t+2} + \gamma \hat{v}(s_{t+2}; \mathbf{w}_{k+1}) >$ *etc.*
- The last setting gives non-stationary training data as data is generated using the function approximator itself.
- The key step in eqn. 1, relies on the target being independent of $\mathbf{w}_t$.
- An approximate way is to take into account the effect of changing $\mathbf{w}_t$ on the estimate but ignore its effect on the target. Thus it is termed as a *semi-gradient descent* method.

Semi-gradient TD(0)

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S' is terminal

Figure credit: *[SB: Chapter 9]*

Feature Vectors

- Feature vectors are representations of states. Corresponding to every state s, there is a real-valued vector $\mathbf{x}(s) \in \mathbb{R}^d$ *i.e.*, with same number of components as $\mathbf{w}$

$$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_d(s) \end{pmatrix}$$

- For example, features can be,
  - Distance of robot from landmarks.
  - Co-ordinates of cells in gridworlds.
  - Piece and pawn configurations in chess.
- Linear methods approximate state-value function by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$.

$$\hat{v}(s; \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i$$

Linear Function Approximator

- The gradient w.r.t. $\mathbf{w}$ is, $\nabla_{\mathbf{w}} \hat{v}(s; \mathbf{w}) = \mathbf{x}(s)$.
- So, the SGD update rule in linear function approximator case is

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \left[v_\pi(s_t) - \hat{v}(s_t; \mathbf{w}_k) \right] \mathbf{x}(s_t)$$

- The gradient Monte Carlo algorithm converges to the global optimum under linear function approximation if α is reduced over time according to the usual conditions.
- The semi-gradient TD(0) algorithm also converges under linear function approximation, but the convergence proof is not trivial (see section 9.4 in [SB]).
- The update rule, in this case, is

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \left[R_{t+1} + \gamma \mathbf{w}_k^T \mathbf{x}(s_{t+1}) - \mathbf{w}_k^T \mathbf{x}(s_t) \right] \mathbf{x}(s_t)$$

Table Look-up Features

- Table look-up is a special case of linear value function approximation
- Using table lookup features

$$\mathbf{x}^{table}(s) = \begin{pmatrix} \mathbf{1}(s = s_1) \\ \vdots \\ \mathbf{1}(s = s_d) \end{pmatrix}$$

- Parameter vector $\mathbf{w}$ gives value of each individual state,

$$\hat{v}(s; \mathbf{w}) = \begin{pmatrix} \mathbf{1}(s = s_1) \\ \vdots \\ \mathbf{1}(s = s_d) \end{pmatrix}^T \begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix}$$

- State aggregation methods are used to save memory and ease computation.

Looking back at Eligibility Traces

§ Lets say we get a reward at the end of some step. What eligibility trace says is that the credit for the reward should trickle down in proportion to all the way to the first state. The credit should be more for the state-action pairs which were close to the rewarding step and also for those state-action pairs which were visited frequently along the way.

§

$t \leftarrow 1$;
repeat
  After $s_{t-1} \xrightarrow{R_t} s_t$
    $e(s_{t-1}) = e(s_{t-1}) + 1$;
  foreach $s \in \mathcal{S}$ **do**
    $V_T(s) \leftarrow V_T(s) + \alpha_T(R_t + \gamma V_{T-1}(s_t) - V_{T-1}(s_{t-1}))e(s)$;
    $e(s) = \lambda\gamma e(s)$
  $t \leftarrow t + 1$
until *this episode terminates*;

$E(S, A) \leftarrow E(S, A) + 1$
For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
  $Q(s, a) \leftarrow Q(s, a) + \alpha\delta E(s, a)$
  $E(s, a) \leftarrow \gamma\lambda E(s, a)$
$S \leftarrow S'$; $A \leftarrow A'$

Eligibility Trace for Function Approximation

- Earlier we had eligibility values for all states. Now, we have two options for eligibility traces in function approximation based methods.
- Should the eligibility trace be on the features $\mathbf{x}(s)$ or the parameters $\mathbf{w}$?
- Hint:
  - What was eligibility doing in TD(λ) or SARSA(λ) algorithms? Was it associated with something getting updated or not?

Eligibility Trace for Function Approximation

- Earlier we had eligibility values for all states. Now, we have two options for eligibility traces in function approximation based methods.
- Should the eligibility trace be on the features $\mathbf{x}(s)$ or the parameters $\mathbf{w}$?
- Hint:
  - What was eligibility doing in TD(λ) or SARSA(λ) algorithms? Was it associated with something getting updated or not?
- The update step updates the parameter values. So, eligibilities are associated with the parameters.

Eligibility Trace for Function Approximation

- From slide 14, the update rule w/o eligibility was

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\delta_t \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_k), \text{ where, } \delta_t = [R_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_k) - \hat{v}(s_t; \mathbf{w}_k)$$

- Now it will be,

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\delta_t \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_k)\mathbf{e}(\mathbf{w}_k) \quad \texttt{//elementwise}$$

Eligibility Trace for Function Approximation

- From slide 14, the update rule w/o eligibility was

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\delta_t \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_k), \text{ where, } \delta_t = [R_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_k) - \hat{v}(s_t; \mathbf{w}_k)$$

- Now it will be,

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\delta_t \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_k)\mathbf{e}(\mathbf{w}_k) \quad \texttt{//elementwise}$$

- From slide 16, the update rule w/o eligibility for linear models was

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\delta_t \mathbf{x}(s_t)$$

- Now it will be,

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha\delta_t \mathbf{x}(s_t)\mathbf{e}(\mathbf{w}_k) \quad \texttt{//elementwise}$$

Eligibility Trace for Function Approximation

- What eligibility of a state (in earlier cases of TD(0), say) signifies is that - how much a state is responsible for the the final reward that is obtained.
- Keeping this in mind - what is a quantity that is performing a similar thing in function approximation?

Eligibility Trace for Function Approximation

- What eligibility of a state (in earlier cases of TD(0), say) signifies is that - how much a state is responsible for the the final reward that is obtained.
- Keeping this in mind - what is a quantity that is performing a similar thing in function approximation?
- Gradients - *i.e.*, the partial derivatives of the predicted value w.r.t. the parameters.
- It tells how much a particular parameter is responsible for the predicted output.
- This similarity is used by replacing the usual way of incrementing eligibility by 1, by an accumulation of gradients.

Eligibility Trace for Function Approximation

§

$$\delta_t = [R_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_k) - \hat{v}(s_t; \mathbf{w}_k)]$$
$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t-1} + \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_k)$$
$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \delta_t \mathbf{e}_t$$

Eligibility Trace for Function Approximation

§

$$\delta_t = [R_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_k) - \hat{v}(s_t; \mathbf{w}_k)]$$
$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t-1} + \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_k)$$
$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \delta_t \mathbf{e}_t$$

§ For linear models, this second line will be changed to,

$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t+1} + \mathbf{x}(s_t)$$

Eligibility Trace for Function Approximation

§

$$\delta_t = [R_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_k) - \hat{v}(s_t; \mathbf{w}_k)]$$
$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t-1} + \nabla_{\mathbf{w}} \hat{v}(s_t; \mathbf{w}_k)$$
$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha \delta_t \mathbf{e}_t$$

§ For linear models, this second line will be changed to,

$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t+1} + \mathbf{x}(s_t)$$

§ Using lookup table representation, this becomes,

$$\mathbf{e}_t = \gamma \lambda \mathbf{e}_{t+1} + \begin{pmatrix} \mathbf{1}(s = s_1) \\ \vdots \\ \mathbf{1}(s = s_d) \end{pmatrix}$$

Control with Function Approximation

- Like the MC or TD approaches, we first need to resort to action value function $Q(s, a)$.
- For this, the very first thing that is needed is to encode the actions as well, *i.e.*, we need to find $\mathbf{x}(s, a)$ instead of $\mathbf{x}(s)$.
- Encoding the actions as one-hot vectors is often a possible choice.
- Another option is to maintain different set of parameters for different action, *i.e.*, $Q(s, a) = \mathbf{x}^T(s)\mathbf{w}_a$

Control with Function Approximation

- Like the MC or TD approaches, we first need to resort to action value function $Q(s, a)$.
- For this, the very first thing that is needed is to encode the actions as well, *i.e.*, we need to find $\mathbf{x}(s, a)$ instead of $\mathbf{x}(s)$.
- Encoding the actions as one-hot vectors is often a possible choice.
- Another option is to maintain different set of parameters for different action, *i.e.*, $Q(s, a) = \mathbf{x}^T(s)\mathbf{w}_a$
- (From previous topic): The SARSA update rule is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)$$

- So, for $Q(s_{t+1}, a_{t+1}), \mathbf{x}^T(s_{t+1})\mathbf{w}_{a_{t+1}}$ will be used and for $Q(s_t, a_t), \mathbf{x}^T(s_t)\mathbf{w}_{a_t}$ will be used

Afterstates

- These options are managable for small number of actions.
- What about large number of actions or continuous actions?
- What about generalization?

Afterstates

- These options are managable for small number of actions.
- What about large number of actions or continuous actions?
- What about generalization?
- The concept of '*afterstates*' help in this regard.
- Afterstate concept is based on separating stochasticity of states and deterministic nature of agent's moves.

Afterstates

- These options are managable for small number of actions.
- What about large number of actions or continuous actions?
- What about generalization?
- The concept of '*afterstates*' help in this regard.
- Afterstate concept is based on separating stochasticity of states and deterministic nature of agent's moves.

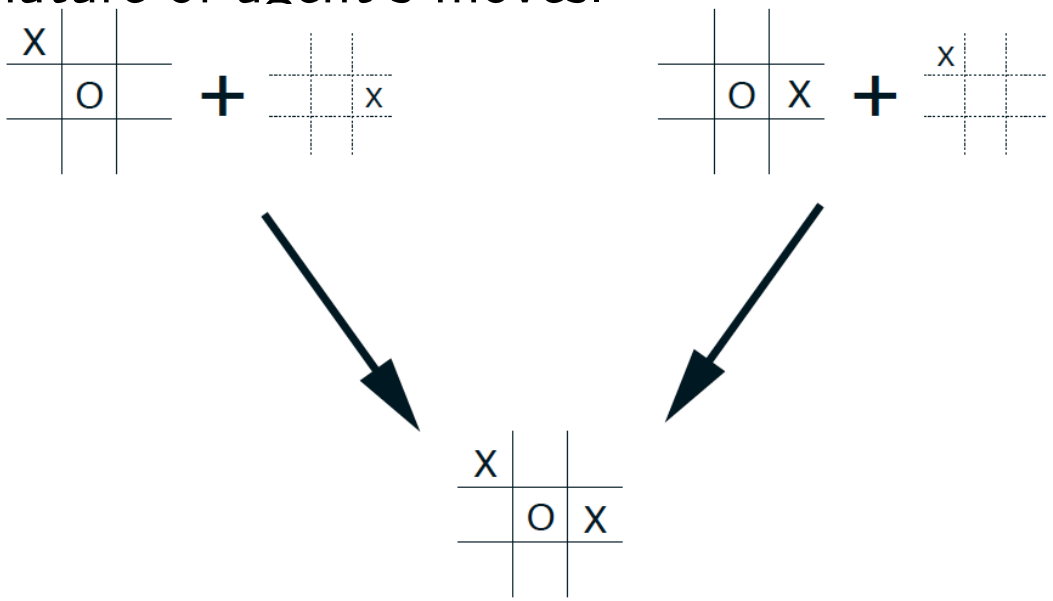


Figure credit: *[SB: Chapter 6]*

Afterstates

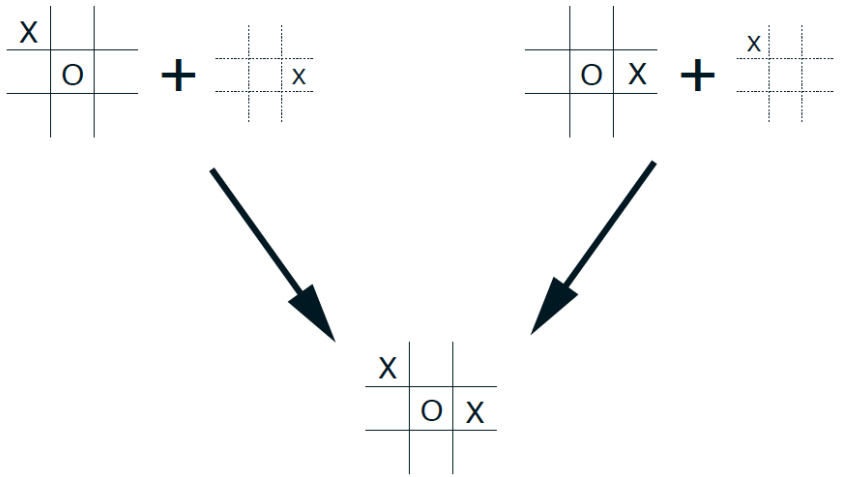
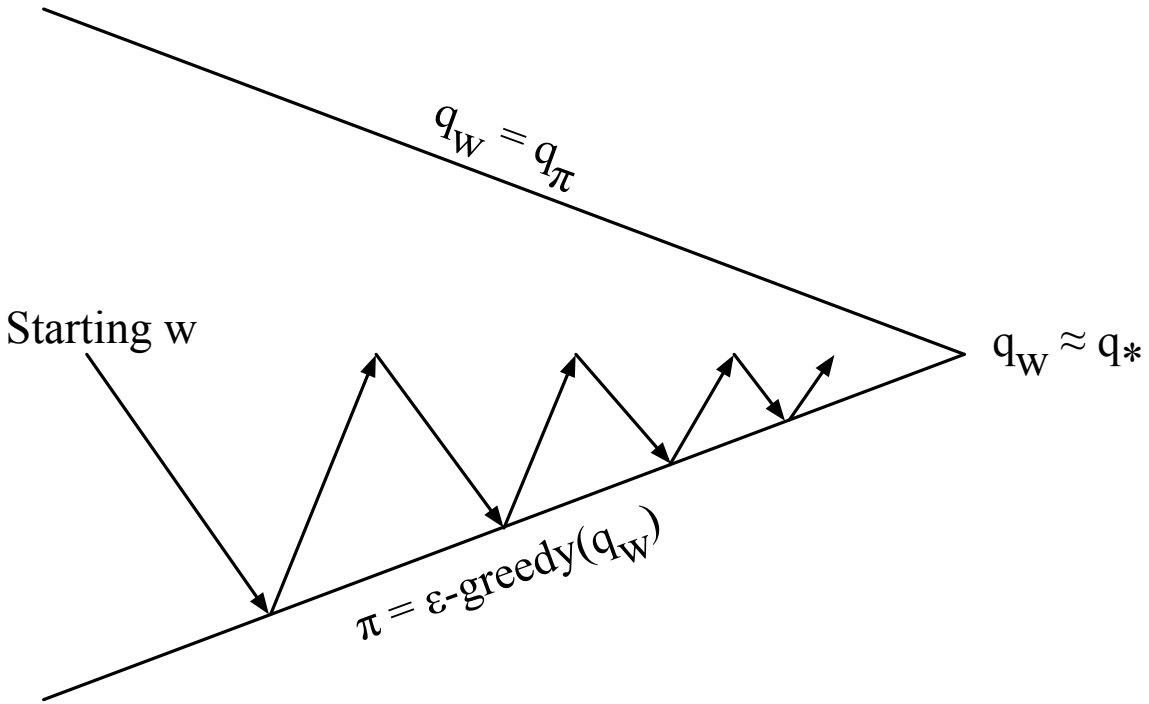


Figure credit: *[SB: Chapter 6]*

- In such cases, the position move pairs are different but produce the same "afterposition".
- A conventional action value function would have to separately asses both pairs whereas an afterstate value function would immediately asses both equally.
- $\mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}' \rightarrow \mathcal{S}$. Instead of learning action value over (s, a), state value function over afterstates are learned.
- What are the advantages?

Control with Value Function Approximation



Policy evaluation Approximate policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

Figure credit: *[David Silver: Deepmind]*

Control with Value Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

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

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

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

Slide credit: *[David Silver: Deepmind]*

Control with Value Function Approximation

- Like prediction, we must substitute a *target* for $q_\pi(S, A)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For forward-view TD(λ), target is the action-value λ-return

$$\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For backward-view TD(λ), equivalent update is

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$
$$E_t = \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$
$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Slide credit: *[David Silver: Deepmind]*

Least Squares Solution

- We have seen that the error that is minimized in function approximation based methods is,

$$\frac{1}{N}\sum_{t=1}^{N}\left[v_\pi(s_t) - \hat{v}(s_t;\mathbf{w})\right]^2$$

- We have also seen how this error function is minimized using gradient descent updates.
- But, if we look carefully, solving this is nothing but finding a least squares solution, *i.e.*, finding parameter vector $\mathbf{w}$ minimising sum-squared error between the target values and the predicted values.
- We know that, for linear function approximator (*i.e.* when $\hat{v}(s_t;\mathbf{w})$ is a linear function of $\mathbf{w}$), the solution to this is exact and is obtained in closed form.

Least Squares Solution

§ From slide 16, we have seen that the update rule for linear function approximator is

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\Delta\mathbf{w}_t = \mathbf{w}_t + \alpha\left[R_{t+1} + \gamma\mathbf{w}_t^T\mathbf{x}(s_{t+1}) - \mathbf{w}_t^T\mathbf{x}(s_t)\right]\mathbf{x}(s_t) \\
&= \mathbf{w}_t + \alpha\left[R_{t+1} + \gamma\mathbf{w}_t^T\mathbf{x}_{t+1} - \mathbf{w}_t^T\mathbf{x}_t\right]\mathbf{x}_t \text{ [using } \mathbf{x}_t \text{ for } \mathbf{x}(s_t)\text{]} \\
&= \mathbf{w}_t + \alpha[R_{t+1}\mathbf{x}_t + \gamma\underbrace{\mathbf{w}_t^T\mathbf{x}_{t+1}}_{\text{scalar}}\mathbf{x}_t - \underbrace{\mathbf{w}_t^T\mathbf{x}_t}_{\text{scalar}}\mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha[R_{t+1}\mathbf{x}_t + \gamma\mathbf{x}_t\mathbf{x}_{t+1}^T\mathbf{w}_t - \mathbf{x}_t\mathbf{x}_t^T\mathbf{w}_t] \\
&= \mathbf{w}_t + \alpha[R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\mathbf{w}_t]
\end{aligned}$$

§ For a given $\mathbf{w}_t$, the expected value of the new weight ($\mathbf{w}_{t+1}$) can be written as, (The expectation is for different values of R and $\mathbf{x}$'s)

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t)$$

where, $\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d$ and $\mathbf{A} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T] \in \mathbb{R}^{d\times d}$

Least Squares Solution

- At minimum, the update will mean no change, *i.e.*,

$$\mathbf{b} - \mathbf{A}\mathbf{w}_{TD} = 0$$
$$\implies \mathbf{b} = \mathbf{A}\mathbf{w}_{TD}$$
$$\implies \mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b}$$

- This quantity is called the *TD fixed point*.
- For non-linear function approximators, the TD fixed point is not obtained in closed form. However, gradient descent update can be applied on a batch of training data for an iterative solution in such cases.
- Traditional supervised least square solution requires training data in the form

$$\mathcal{D} = \{\langle x^{(1)}, f(x^{(1)})\rangle, \langle x^{(2)}, f(x^{(2)})\rangle, \cdots, \langle x^{(T)}, f(x^{(T)})\rangle\}$$

where the following loss is minimized,

$$LS(\mathbf{w}) = \frac{1}{2}\mathbb{E}_{\mathcal{D}}\left[f(x^{(i)}) - \hat{f}(x^{(i)}; \mathbf{w})\right]^2 = \frac{1}{2T}\sum_{i=1}^{T}\left[f(x^{(i)}) - \hat{f}(x^{(i)}; \mathbf{w})\right]^2$$

Least Squares Solution

- The gradient of the loss is computed as,

$$\frac{1}{T}\sum_{i=1}^{T}\Big[f(x^{(i)}) - \hat{f}(x^{(i)};\mathbf{w})\Big]\nabla_{\mathbf{w}}\hat{f}(x^{(i)};\mathbf{w})$$

- This is pretty similar to the gradients used in the function approximation methods till now, but with an important difference.
- They are not batch methods. The gradients are computed per step. Thus it is not sample efficient.
- The idea is to form batches of training data from a few trajectories. This is called *experience*.

$$\mathcal{D} = \{\langle s^{(1)}, v_\pi(s^{(1)})\rangle, \langle s^{(2)}, v_\pi(s^{(2)})\rangle, \cdots, \langle s^{(T)}, v_\pi(s^{(T)})\rangle\}$$

- And the gradient of the loss is,

$$\frac{1}{T}\sum_{i=1}^{T}\Big[v_\pi(s^{(i)}) - \hat{v}(s^{(i)};\mathbf{w})\Big]\nabla_{\mathbf{w}}\hat{v}(s^{(i)};\mathbf{w})$$

Mini-batch Gradient Descent with Experience Replay

- Given experience of the form,

$$\mathcal{D} = \{\langle s^{(1)}, v_\pi(s^{(1)})\rangle, \langle s^{(2)}, v_\pi(s^{(2)})\rangle, \cdots, \langle s^{(T)}, v_\pi(s^{(T)})\rangle\}$$

- Repeat
  - Sample state, value from experience $\langle s^{(i)}, v_\pi(s^{(i)})\rangle \sim \mathcal{D}$
  - Apply minibatch gradient descent update

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_i \left[v_\pi(s^{(i)}) - \hat{v}(s^{(i)}; \mathbf{w})\right] \nabla_{\mathbf{w}} \hat{v}(s^{(i)}; \mathbf{w})$$

- Converges to least square solution.

Experience Replay in Deep Q-Networks (DQN)

DQN uses experience replay and fixed Q-targets

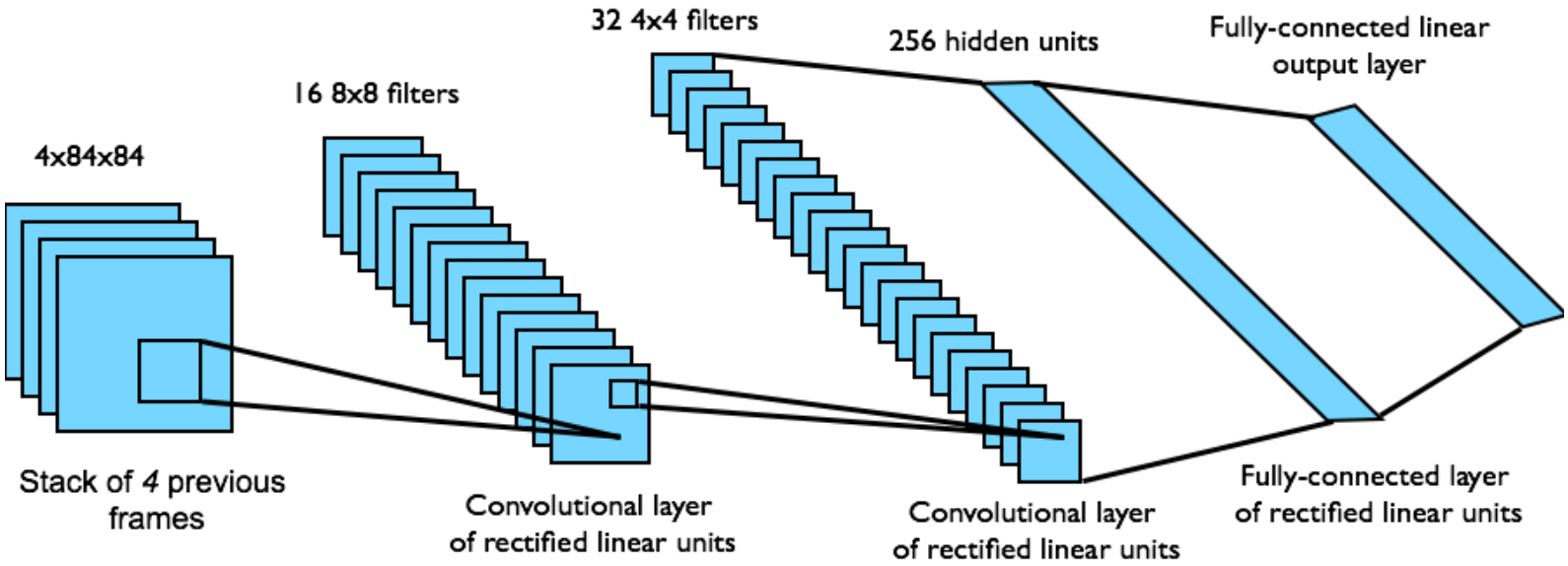
- Take action a_t at according to ϵ-greedy policy.
- Store transition $(s_t, a_t, R_{t+1}, s_{t+1}, a_{t+1}, \cdots)$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
- Compute Q-learning targets w.r.t. old, fixed parameters $\mathbf{w}^-$
- Optimise MSE between Q-network and Q-learning targets

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

- Using minibatch gradient descent

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



Network architecture and hyperparameters fixed across all games

Slide credit: *[David Silver: Deepmind]*