increase, yielding another U-shape relationship between generalization error versus model capacity. We try to find the optimal capacity point, of which under-fitting occurs on the left and over-fitting occurs on the right. Regularization add a penalty term to the cost function, to reduce the generalization error, but not training error. No free lunch theorem states that there is no universally best model, or best regularizor. An implication is that deep learning may not be the best model for some problems. There are model parameters, and hyperparameters for model capacity and regularization. Cross-validation is used to tune hyperparameters, to strike a balance between bias and variance, and to select the optimal model.

Maximum likelihood estimation (MLE) is a common approach to derive good estimation of parameters. For issues like numerical underflow, the product in MLE is converted to summation to obtain negative log-likelihood (NLL). MLE is equivalent to minimizing KL divergence, the dissimilarity between the empirical distribution defined by the training data and the model distribution. Minimizing KL divergence between two distributions corresponds to minimizing the cross-entropy between the distributions. In short, maximization of likelihood becomes minimization of the negative log-likelihood (NLL), or equivalently, minimization of cross entropy.

Gradient descent is a common approach to solve optimization problems. Stochastic gradient descent extends gradient descent by working with a single sample each time, and usually with minibatches.

Importance sampling is a technique to estimate properties of a particular distribution, by samples from a different distribution, to lower the variance of the estimation, or when sampling from the distribution of interest is difficult.

Frequentist statistics estimates a single value, and characterizes variance by confidence interval; Bayesian statistics considers the distribution of an estimate when making predictions and decisions.

generative vs discriminative

2.2 DEEP LEARNING

Deep learning is in contrast to "shallow" learning. For many machine learning algorithms, e.g., linear regression, logistic regression, support vector machines (SVMs), decision trees, and boosting, we have input layer and output layer, and the inputs may be transformed with manual feature engineering before training. In deep learning, between input and output layers, we have one or more hidden layers. At each layer except input layer, we compute the input to each unit, as the weighted sum of units from the previous layer; then we usually use nonlinear transformation, or activation function, such as logistic, tanh, or more popular recently, rectified linear unit (ReLU), to apply to the input of a unit, to obtain a new representation of the input from previous layer. We have weights on links between units from layer to layer. After computations flow forward from input to output, at output layer and each hidden layer, we can compute error derivatives backward, and backpropagate gradients towards the input layer, so that weights can be updated to optimize some loss function.

A feedforward deep neural network or multilayer perceptron (MLP) is to map a set of input values to output values with a mathematical function formed by composing many simpler functions at each layer. A convolutional neural network (CNN) is a feedforward deep neural network, with convolutional layers, pooling layers and fully connected layers. CNNs are designed to process data with multiple arrays, e.g., colour image, language, audio spectrogram, and video, benefit from the properties of such signals: local connections, shared weights, pooling and the use of many layers, and are inspired by simple cells and complex cells in visual neuroscience (LeCun et al., 2015). ResNets (He et al., 2016d) are designed to ease the training of very deep neural networks by adding shortcut connections to learn residual functions with reference to the layer inputs. A recurrent neural network (RNN) is often used to process sequential inputs like speech and language, element by element, with hidden units to store history of past elements. A RNN can be seen as a multilayer neural network with all layers sharing the same weights, when being unfolded in time of forward computation. It is hard for RNN to store information for very long time and the gradient may vanish. Long short term memory networks (LSTM) (Hochreiter and Schmidhuber, 1997) and gated recurrent unit (GRU) (Chung et al., 2014) were proposed to address such issues, with gating mechanisms to manipulate information through recurrent cells. Gradient backpropagation or its variants can be used for training all deep neural networks mentioned above.

Dropout (Srivastava et al., 2014) is a regularization strategy to train an ensemble of sub-networks by removing non-output units randomly from the original network. Batch normalization (Ioffe and Szegedy, 2015) performs the normalization for each training mini-batch, to accelerate training by reducing internal covariate shift, i.e., the change of parameters of previous layers will change each layer's inputs distribution.

Deep neural networks learn representations automatically from raw inputs to recover the compositional hierarchies in many natural signals, i.e., higher-level features are composed of lower-level ones, e.g., in images, the hierarch of objects, parts, motifs, and local combinations of edges. Distributed representation is a central idea in deep learning, which implies that many features may represent each input, and each feature may represent many inputs. The exponential advantages of deep, distributed representations combat the exponential challenges of the curse of dimensionality. The notion of end-to-end training refers to that a learning model uses raw inputs without manual feature engineering to generate outputs, e.g., AlexNet (Krizhevsky et al., 2012) with raw pixels for image classification, Seq2Seq (Sutskever et al., 2014) with raw sentences for machine translation, and DQN (Mnih et al., 2015) with raw pixels and score to play games.

2.3 Reinforcement Learning

We provide background of reinforcement learning briefly in this section. After setting up the RL problem, we discuss value function, temporal difference learning, function approximation, policy optimization, deep RL, RL parlance, and close this section with a brief summary. To have a good understanding of deep reinforcement learning, it is essential to have a good understanding of reinforcement learning first.

2.3.1 Problem Setup

A RL agent interacts with an environment over time. At each time step t, the agent receives a state s_t in a state space $\mathcal{S}$ and selects an action a_t from an action space $\mathcal{A}$, following a policy $\pi(a_t|s_t)$, which is the agent's behavior, i.e., a mapping from state s_t to actions a_t, receives a scalar reward r_t, and transitions to the next state s_{t+1}, according to the environment dynamics, or model, for reward function $\mathcal{R}(s,a)$ and state transition probability $\mathcal{P}(s_{t+1}|s_t,a_t)$ respectively. In an episodic problem, this process continues until the agent reaches a terminal state and then it restarts. The return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the discounted, accumulated reward with the discount factor $\gamma \in (0,1]$. The agent aims to maximize the expectation of such long term return from each state. The problem is set up in discrete state and action spaces. It is not hard to extend it to continuous spaces.

2.3.2 Exploration vs Exploitation

multi-arm bandit

various exploration techniques

2.3.3 Value Function

A value function is a prediction of the expected, accumulative, discounted, future reward, measuring how good each state, or state-action pair, is. The state value $v_\pi(s) = E[R_t|s_t = s]$ is the expected return for following policy π from state s. $v_\pi(s)$ decomposes into the Bellman equation: $v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$. An optimal state value $v_*(s) = \max_\pi v_\pi(s) = \max_a q_{\pi^*}(s,a)$ is the maximum state value achievable by any policy for state s. $v_*(s)$ decomposes into the Bellman equation: $v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$. The action value $q_\pi(s,a) = E[R_t|s_t = s, a_t = a]$ is the expected return for selecting action a in state s and then following policy π. $q_\pi(s,a)$ decomposes into the Bellman equation: $q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')]$. An optimal action value function $q_*(s,a) = \max_\pi q_\pi(s,a)$ is the maximum action value achievable by any policy for state s and action a. $q_*(s,a)$ decomposes into the Bellman equation: $q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]$. We denote an optimal policy by π^*.

2.3.4 DYNAMIC PROGRAMMING

2.3.5 TEMPORAL DIFFERENCE LEARNING

When a RL problem satisfies the Markov property, i.e., the future depends only on the current state and action, but not on the past, it is formulated as a Markov Decision Process (MDP), defined by the 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. When the system model is available, we use dynamic programming methods: policy evaluation to calculate value/action value function for a policy, value iteration and policy iteration for finding an optimal policy. When there is no model, we resort to RL methods. RL methods also work when the model is available. Additionally, a RL environment can be a multi-armed bandit, an MDP, a POMDP, a game, etc.

Temporal difference (TD) learning is central in RL. TD learning is usually refer to the learning methods for value function evaluation in Sutton (1988). SARSA (Sutton and Barto, 2018) and Q-learning (Watkins and Dayan, 1992) are also regarded as temporal difference learning.

TD learning (Sutton, 1988) learns value function $V(s)$ directly from experience with TD error, with bootstrapping, in a model-free, online, and fully incremental way. TD learning is a prediction problem. The update rule is $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$, where α is a learning rate, and $r + \gamma V(s') - V(s)$ is called TD error. Algorithm 1 presents the pseudo code for tabular TD learning. Precisely, it is tabular TD(0) learning, where "0" indicates it is based on one-step return.

Bootstrapping, like the TD update rule, estimates state or action value based on subsequent estimates, is common in RL, like TD learning, Q learning, and actor-critic. Bootstrapping methods are usually faster to learn, and enable learning to be online and continual. Bootstrapping methods are not instances of true gradient decent, since the target depends on the weights to be estimated. The concept of semi-gradient descent is then introduced (Sutton and Barto, 2018).

Input: the policy π to be evaluated
Output: value function V
initialize V arbitrarily, e.g., to 0 for all states
for *each episode* **do**
    initialize state s
    for *each step of episode, state s is not terminal* **do**
        $a \leftarrow$ action given by π for s
        take action a, observe r, s'
        $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
        $s \leftarrow s'$
    end
end

Algorithm 1: TD learning, adapted from Sutton and Barto (2018)

Output: action value function Q
initialize Q arbitrarily, e.g., to 0 for all states, set action value for terminal states as 0
for *each episode* **do**
    initialize state s
    for *each step of episode, state s is not terminal* **do**
        $a \leftarrow$ action for s derived by Q, e.g., ϵ-greedy
        take action a, observe r, s'
        $a' \leftarrow$ action for s' derived by Q, e.g., ϵ-greedy
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$, $a \leftarrow a'$
    end
end

Algorithm 2: SARSA, adapted from Sutton and Barto (2018)

Output: action value function Q
initialize Q arbitrarily, e.g., to 0 for all states, set action value for terminal states as 0
for *each episode* **do**
    initialize state s
    for *each step of episode, state s is not terminal* **do**
        $a \leftarrow$ action for s derived by Q, e.g., ϵ-greedy
        take action a, observe r, s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
    end
end

Algorithm 3: Q learning, adapted from Sutton and Barto (2018)

SARSA, representing state, action, reward, (next) state, (next) action, is an on-policy control method to find the optimal policy, with the update rule, $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$. Algorithm 2 presents the pseudo code for tabular SARSA, precisely tabular SARSA(0).

Q-learning is an off-policy control method to find the optimal policy. Q-learning learns action value function, with the update rule, $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$. Q learning refines the policy greedily with respect to action values by the max operator. Algorithm 3 presents the pseudo code for Q learning, precisely, tabular Q(0) learning.

TD-learning, Q-learning and SARSA converge under certain conditions. From an optimal action value function, we can derive an optimal policy.

2.3.6 Multi-step Bootstrapping

The above algorithms are referred to as TD(0) and Q(0), learning with one-step return. We have TD learning and Q learning variants and Monte-Carlo approach with multi-step return in the forward view. The eligibility trace from the backward view provides an online, incremental implementation, resulting in TD(λ) and Q(λ) algorithms, where $\lambda \in [0,1]$. TD(1) is the same as the Monte Carlo approach.

Eligibility trace is a short-term memory, usually lasting within an episode, assists the learning process, by affecting the weight vector. The weight vector is a long-term memory, lasting the whole duration of the system, determines the estimated value. Eligibility trace helps with the issues of long-delayed rewards and non-Markov tasks (Sutton and Barto, 2018).

TD(λ) unifies one-step TD prediction, TD(0), with Monte Carlo methods, TD(1), using eligibility traces and the decay parameter λ, for prediction algorithms. De Asis et al. (2018) made unification for multi-step TD control algorithms.

2.3.7 Function Approximation

We discuss the tabular cases above, where a value function or a policy is stored in a tabular form. Function approximation is a way for generalization when the state and/or action spaces are large or continuous. Function approximation aims to generalize from examples of a function to construct an approximate of the entire function; it is usually a concept in supervised learning, studied in the fields of machine learning, patten recognition, and statistical curve fitting; function approximation in reinforcement learning usually treats each backup as a training example, and encounters new issues like nonstationarity, bootstrapping, and delayed targets (Sutton and Barto, 2018). Linear function approximation is a popular choice, partially due to its desirable theoretical properties, esp. before the work of Deep Q-Network (Mnih et al., 2015). However, the integration of reinforcement learning and neural networks dated back a long time ago (Sutton and Barto, 2018; Bertsekas and Tsitsiklis, 1996; Schmidhuber, 2015).

Algorithm 4 presents the pseudo code for TD(0) with function approximation. $\hat{v}(s, \boldsymbol{w})$ is the approximate value function, $\boldsymbol{w}$ is the value function weight vector, $\nabla \hat{v}(s, \boldsymbol{w})$ is the gradient of the approximate value function with respect to the weight vector, and the weight vector is updated following the update rule, $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma \hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})] \nabla \hat{v}(s, \boldsymbol{w})$.

Input: the policy π to be evaluated
Input: a differentiable value function $\hat{v}(s, \boldsymbol{w})$, $\hat{v}(terminal, \cdot) = 0$
Output: value function $\hat{v}(s, \boldsymbol{w})$
initialize value function weight $\boldsymbol{w}$ arbitrarily, e.g., $\boldsymbol{w} = 0$
for *each episode* **do**
    initialize state s
    for *each step of episode, state s is not terminal* **do**
        $a \leftarrow \pi(\cdot|s)$
        take action a, observe r, s'
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma\hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})]\nabla\hat{v}(s, \boldsymbol{w})$
        $s \leftarrow s'$
    end
end

Algorithm 4: TD(0) with function approximation, adapted from Sutton and Barto (2018)

When combining off-policy, function approximation, and bootstrapping, instability and divergence may occur (Tsitsiklis and Van Roy, 1997), which is called the deadly triad issue (Sutton and Barto, 2018). All these three elements are necessary: function approximation for scalability and generalization, bootstrapping for computational and data efficiency, and off-policy learning for freeing behaviour policy from target policy. What is the root cause for the instability? Learning or sampling are not, since dynamic programming suffers from divergence with function approximation; exploration, greedification, or control are not, since prediction alone can diverge; local minima or complex non-linear function approximation are not, since linear function approximation can produce instability (Sutton, 2016). It is unclear what is the root cause for instability – each single factor mentioned above is not – there are still many open problems in off-policy learning (Sutton and Barto, 2018).

Table 1 presents various algorithms that tackle various issues (Sutton, 2016). Deep RL algorithms like Deep Q-Network (Mnih et al., 2015) and A3C (Mnih et al., 2016) are not presented here, since they do not have theoretical guarantee, although they achieve stunning performance empirically.

Before explaining Table 1, we introduce some background definitions. Recall that Bellman equation for value function is $v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$. Bellman operator is defined as $(B_\pi v)(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$. TD fix point is then $v_\pi = B_\pi v_\pi$. Bellman error for the function approximation case is then $\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma\hat{v}^\pi(s', \boldsymbol{w})] - \hat{v}^\pi(s, \boldsymbol{w})$, the right side of Bellman equation with function approximation minus the left side. It can be written as $B_\pi v_{\boldsymbol{w}} - v_{\boldsymbol{w}}$. Bellman error is the expectation of the TD error.

ADP algorithms refer to dynamic programming algorithms like policy evaluation, policy iteration, and value iteration, with function approximation. Least square temporal difference (LSTD) (Bradtke and Barto, 1996) computes TD fix-point directly in batch mode. LSTD is data efficient, yet with squared time complexity. LSPE (Nedić and Bertsekas, 2003) extended LSTD. Fitted-Q algorithms (Ernst et al., 2005; Riedmiller, 2005) learn action values in batch mode. Residual gradient algorithms (Baird, 1995) minimize Bellman error. Gradient-TD (Sutton et al., 2009a;b; Mahmood et al., 2014) methods are true gradient algorithms, perform SGD in the projected Bellman error (PBE), converge robustly under off-policy training and non-linear function approximation. Emphatic-TD (Sutton et al., 2016) emphasizes some updates and de-emphasizes others by reweighting, improving computational efficiency, yet being a semi-gradient method. See Sutton and Barto (2018) for more details. Du et al. (2017) proposed variance reduction techniques for policy evaluation to achieve fast convergence. White and White (2016) performed empirical comparisons of linear TD methods, and made suggestions about their practical use.

2.3.8 Policy Optimization

In contrast to value-based methods like TD learning and Q-learning, policy-based methods optimize the policy $\pi(a|s; \boldsymbol{\theta})$ (with function approximation) directly, and update the parameters $\boldsymbol{\theta}$ by gradient ascent on $E[R_t]$. REINFORCE (Williams, 1992) is a policy gradient method, updating $\boldsymbol{\theta}$ in the direction of $\nabla_{\boldsymbol{\theta}} \log \pi(a_t|s_t; \boldsymbol{\theta}) R_t$. Usually a baseline $b_t(s_t)$ is subtracted from the return to reduce the variance of gradient estimate, yet keeping its unbiasedness, to yield the gradient direction $\nabla_{\boldsymbol{\theta}} \log \pi(a_t|s_t; \boldsymbol{\theta})(R_t - b_t(s_t))$. Using $V(s_t)$ as the baseline $b_t(s_t)$, we have the advantage func-

		algorithm					
		TD(λ) SARSA(λ)	ADP	LSTD(λ) LSPE(λ)	Fitted-Q	Residual Gradient	GTD(λ) GQ(λ)
issue	linear computation	✓	✓			✓	✓
	nonlinear convergent				✓	✓	✓
	off-policy convergent			✓		✓	✓
	model-free, online	✓		✓		✓	✓
	converges to PBE = 0	✓	✓	✓	✓		✓

Table 1: RL Issues vs. Algorithms

tion $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$, since R_t is an estimate of $Q(a_t, s_t)$. Algorithm 5 presents the pseudo code for REINFORCE algorithm in the episodic case.

Input: policy $\pi(a|s, \boldsymbol{\theta})$, $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes, $\alpha > 0$, $\beta > 0$
Output: policy $\pi(a|s, \boldsymbol{\theta})$
initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for *true* **do**
  generate an episode s_0, a_0, r_1, $\cdots$, s_{T-1}, a_{T-1}, r_T, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  for *each step* t *of episode 0,* $\cdots$, $T-1$ **do**
    $G_t \leftarrow$ return from step t
    $\delta \leftarrow G_t - \hat{v}(s_t, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta\delta\nabla_{\boldsymbol{w}}\hat{v}(s_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t\boldsymbol{\delta}\nabla_{\boldsymbol{\theta}} log\pi(a_t|s_t, \boldsymbol{\theta})$
  end
end

Algorithm 5: REINFORCE with baseline (episodic), adapted from Sutton and Barto (2018)

In actor-critic algorithms, the critic updates action-value function parameters, and the actor updates policy parameters, in the direction suggested by the critic. Algorithm 6 presents the pseudo code for one-step actor-critic algorithm in the episodic case.

Input: policy $\pi(a|s, \boldsymbol{\theta})$, $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes, $\alpha > 0$, $\beta > 0$
Output: policy $\pi(a|s, \boldsymbol{\theta})$
initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for *true* **do**
  initialize s, the first state of the episode
  $I \leftarrow 1$
  for s *is not terminal* **do**
    $a \sim \pi(\cdot|s, \boldsymbol{\theta})$
    take action a, observe s', r
    $\delta \leftarrow r + \gamma\hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})$ (if s' is terminal, $\hat{v}(s', \boldsymbol{w}) \doteq 0$)
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta\delta\nabla_{\boldsymbol{w}}\hat{v}(s_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha I\delta\nabla_{\boldsymbol{\theta}} log\pi(a_t|s_t, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $s \leftarrow s'$
  end
end

Algorithm 6: Actor-Critic (episodic), adapted from Sutton and Barto (2018)

Policy iteration alternates between policy evaluation and policy improvement, to generate a sequence of improving policies. In policy evaluation, the value function of the current policy is estimated from the outcomes of sampled trajectories. In policy improvement, the current value function is used to generate a better policy, e.g., by selecting actions greedily with respect to the value function.

2.3.9 Deep Reinforcement Learning

We obtain deep reinforcement learning (deep RL) methods when we use deep neural networks to approximate any of the following components of reinforcement learning: value function, $\hat{v}(s;\boldsymbol{\theta})$ or $\hat{q}(s,a;\boldsymbol{\theta})$, policy $\pi(a|s;\boldsymbol{\theta})$, and model (state transition function and reward function). Here, the parameters $\boldsymbol{\theta}$ are the weights in deep neural networks. When we use "shallow" models, like linear function, decision trees, tile coding and so on as the function approximator, we obtain "shallow" RL, and the parameters $\boldsymbol{\theta}$ are the weight parameters in these models. Note, a shallow model, e.g., decision trees, may be non-linear. The distinct difference between deep RL and "shallow" RL is what function approximator is used. This is similar to the difference between deep learning and "shallow" machine learning. We usually utilize stochastic gradient descent to update weight parameters in deep RL. When off-policy, function approximation, in particular, non-linear function approximation, and bootstrapping are combined together, instability and divergence may occur (Tsitsiklis and Van Roy, 1997). However, recent work like Deep Q-Network (Mnih et al., 2015) and AlphaGo (Silver et al., 2016a) stabilized the learning and achieved outstanding results.

2.3.10 RL Parlance

We explain some terms in RL parlance.

The prediction problem, or policy evaluation, is to compute the state or action value function for a policy. The control problem is to find the optimal policy. Planning constructs a value function or a policy with a model.

On-policy methods evaluate or improve the behavioural policy, e.g., SARSA fits the action-value function to the current policy, i.e., SARSA evaluates the policy based on samples from the same policy, then refines the policy greedily with respect to action values. In off-policy methods, an agent learns an optimal value function/policy, maybe following an unrelated behavioural policy, e.g., Q-learning attempts to find action values for the optimal policy directly, not necessarily fitting to the policy generating the data, i.e., the policy Q-learning obtains is usually different from the policy that generates the samples. The notion of on-policy and off-policy can be understood as same-policy and different-policy.

The exploration-exploitation dilemma is about the agent needs to exploit the currently best action to maximize rewards greedily, yet it has to explore the environment to find better actions, when the policy is not optimal yet, or the system is non-stationary.

In model-free methods, the agent learns with trail-and-error from experience explicitly; the model (state transition function) is not known or learned from experience. RL methods that use models are model-based methods.

In online mode, training algorithms are executed on data acquired in sequence. In offline mode, or batch mode, models are trained on the entire data set.

With bootstrapping, an estimate of state or action value is updated from subsequent estimates.

2.3.11 Brief Summary

A RL problem is formulated as an MDP when the observation about the environment satisfies the Markov property. An MDP is defined by the 5-tuple $(\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R},\gamma)$. A central concept in RL is value function. Bellman equations are cornerstone for developing RL algorithms. Temporal difference learning algorithms are fundamental for evaluating/predicting value functions. Control algorithms find optimal policies. Reinforcement learning algorithms may be based on value function and/or policy, model-free or model-based, on-policy or off-policy, with function approximation or not, with sample backups (TD and Monte Carlo) or full backups (dynamic programming and exhaustive search), and about the depth of backups, either one-step return (TD(0) and dynamic programming) or multi-step return (TD(λ), Monte Carlo, and exhaustive search). When combining

off-policy, function approximation, and bootstrapping, we face instability and divergence (Tsitsiklis and Van Roy, 1997), the deadly triad issue (Sutton and Barto, 2018). Theoretical guarantee has been established for linear function approximation, e.g., Gradient-TD (Sutton et al., 2009a;b; Mahmood et al., 2014), Emphatic-TD (Sutton et al., 2016) and Du et al. (2017). With non-linear function approximation, in particular deep learning, algorithms like Deep Q-Network (Mnih et al., 2015) and AlphaGo (Silver et al., 2016a; 2017) stabilized the learning and achieved stunning results, which is the focus of this overview.

3 CORE ELEMENTS

A RL agent executes a sequence of actions and observe states and rewards, with major components of value function, policy and model. A RL problem may be formulated as a prediction, control or planning problem, and solution methods may be model-free or model-based, with value function and/or policy. Exploration-exploitation is a fundamental tradeoff in RL. Knowledge would be critical for RL. In this section, we discuss core RL elements: value function in Section 3.1, policy in Section 3.2, reward in Section 3.3, model and planning in Section 3.4, exploration in Section 3.5, and knowledge in Section 3.6.

3.1 VALUE FUNCTION

Value function is a fundamental concept in reinforcement learning, and temporal difference (TD) learning (Sutton, 1988) and its extension, Q-learning (Watkins and Dayan, 1992), are classical algorithms for learning state and action value functions respectively. In the following, we focus on Deep Q-Network (Mnih et al., 2015), a recent breakthrough, and its extensions.

3.1.1 DEEP Q-NETWORK (DQN) AND EXTENSIONS

Mnih et al. (2015) introduced Deep Q-Network (DQN) and ignited the field of deep RL. We present DQN pseudo code in Algorithm 7.

Input: the pixels and the game score
Output: Q action value function (from which we obtain policy and select action)
Initialize replay memory D
Initialize action-value function Q with random weight $\boldsymbol{\theta}$
Initialize target action-value function $\hat{Q}$ with weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for *episode = 1 to* M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for t *= 1 to* T **do**
    Following ϵ-greedy policy, select $a_t = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \arg\max_a Q(\phi(s_t), a; \boldsymbol{\theta}) & \text{otherwise} \end{cases}$
    Execute action a_i 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
    `// experience replay`
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ w.r.t. the network parameter $\boldsymbol{\theta}$
    `// periodic update of target network`
    Every C steps reset $\hat{Q} = Q$, i.e., set $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
  end
end

Algorithm 7: Deep Q-Nework (DQN), adapted from Mnih et al. (2015)

Before DQN, it is well known that RL is unstable or even divergent when action value function is approximated with a nonlinear function like neural networks. DQN made several important contributions: 1) stabilize the training of action value function approximation with deep neural networks