Machine Learning for Time Series – WS 2018/19

12. 02. 2019, Thilo Kratzer (yq25iziz)

Bayesian Inference

Prior of θ:	$p(\theta)$
Likelihood of θ:	$p(\mathcal{D} \mid \theta)$
Posterior of θ given $\mathcal{D}$:	$p(\theta \mid \mathcal{D})$
Marginal Likelihood:	$p(\mathcal{D})$
Bayes rule:	$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta) p(\theta)}{p(\mathcal{D})}$
Model comparison:	$p(\mathcal{D}) = \int p(\mathcal{D} \mid \theta) p(\theta) d\theta$
Prediction of new point:	$p(y \mid x, \mathcal{D}) = \int p(y \mid \theta, x, \mathcal{D}) p(\theta \mid \mathcal{D}) d\theta$

Gaussian Process Regression

Gaussian Process:

- Gaussian prior: $p(f \mid \mathcal{M}_i) \sim \mathcal{GP}(m \equiv 0, k)$
- Gaussian likelihood: $p(y \mid x, f, \mathcal{M}_i) \sim \mathcal{N}(f, \sigma^2_{\text{noise}} I)$
- Gaussian posterior: $p(f \mid x, y, \mathcal{M}_i) \sim \mathcal{GP}(m_{\text{post}}, k_{\text{post}})$
- Gaussian predictive: $p(y_* \mid x_*, x, y, \mathcal{M}_i) \sim \mathcal{N}(.,.)$

Gaussian Process Classification

Gaussian Process Classification:

- Place a GP prior on $f(x)$: $f_i = (x_i) = x_i^T w$
- Sigmoidal likelihood: $p(y = +1 \mid f) = \sigma(f)$
- Posterior on $f(x)$: $p(f \mid X, y) = \frac{p(y \mid f) p(f \mid X)}{p(y \mid X)}$
- Inference: $p(f_* \mid X, y, x_*) = \int p(f_* \mid X, x_*, f) p(f \mid X, y) df$
  $\overline{\pi}_* \mathrel{\hat{=}} p(y_* = +1 \mid X, y, x_*) = \int \sigma(f_*) p(f_* \mid X, y, x_*) df_*$

Laplace Approximation: Second order Taylor approximation about mode (value that appears most often) of non-Gaussian posterior $\to$ obtain Gaussian approximation around maximum of posterior

Expectation Propagation Approximation: Approximate the non-Gaussian likelihood by a local likelihood approximation in the form of an un-normalized Gaussian function. Then minimize the Kullback-Leibler divergence between the posterior and its approximation:

$$KL(P \mid\mid Q) = \int_{\mathbb{R}} p(x) \log \frac{p(x)}{q(x)} dx \geq 0$$

Kalmann Filtering

Bayesian Kalman Filtering:

- State space model: $z_n = F_n z_{n-1} + r_n$
  $y_n = H_n z_n + q_n$
- Prediction: $p(z_{n+1} \mid y_{1:n}) = \int p(z_{n+1} \mid z_n) \underbrace{p(z_n \mid y_{1:n})}_{\text{filtering density}} dz_n$
- Filtering: $p(z_n \mid y_{1:n}) = \alpha \underbrace{p(y_n \mid z_n)}_{\text{likelihood}} \underbrace{p(z_n \mid y_{1:n-1})}_{\text{prediction}} = \alpha\, p(y_n \mid z_n) \int p(z_n \mid z_{n-1}) \underbrace{p(z_{n-1} \mid y_{1:n-1})}_{\text{recursive call}} dz_{n-1}$

Linearized Dynamical System: When f and/or h are nonlinear, the transition probability is non-Gaussian and the predictive distribution is, in general, intractable. A possible approach is to consider a linearized dynamical system (using Taylor series) around the estimated mean of the current state. Then, the approximate transition density and likelihood are again Gaussian.

Extended Kalman Filtering: see Linearized Dynamical System

Unscented Kalman Filtering: Let us call sigma points a set of weighted points chosen deterministically, which capture the mean and covariance of the random variable z. The sigma points capture the mean and covariance of z. When propagated through any nonlinear system, the transformed sigma points capture the predictive and filtering mean and covariance.

$$z_0 = \mu \text{ and } z_l = \mu \pm \left[\sqrt{(D+\kappa)\Sigma}\right]_l \text{ where } l = 1, ..., D$$

First step is passing the old state through transition function f in order to approximate the predictive density. The second step is to approximate the likelihood by passing the predictive density through the observation function h.

Monte Carlo Methods

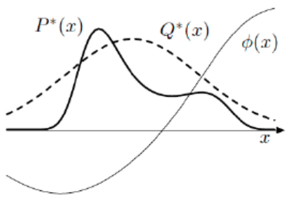
Importance sampling: The goal is to find the expectation of $\phi(x)$ wrt. the distribution $P(x)$.

$$\mathbb{E}[\phi(x)] = \int \phi(x) P(x) dx$$

Sampling from $P(x)$ is difficult, instead sample from a simpler distribution (Proposal distribution) $Q(x)$.

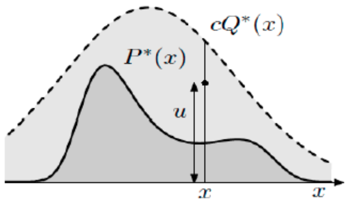
Each sample has a weight: $\omega_r \equiv \frac{P^*(x^{(r)})}{Q^*(x^{(r)})}$

Expectation wrt. $\phi(x)$ becomes: $\mathbb{E}[\phi(x)] = \frac{\sum_r \omega_r \phi(x^{(r)})}{\sum_r \omega_r}$



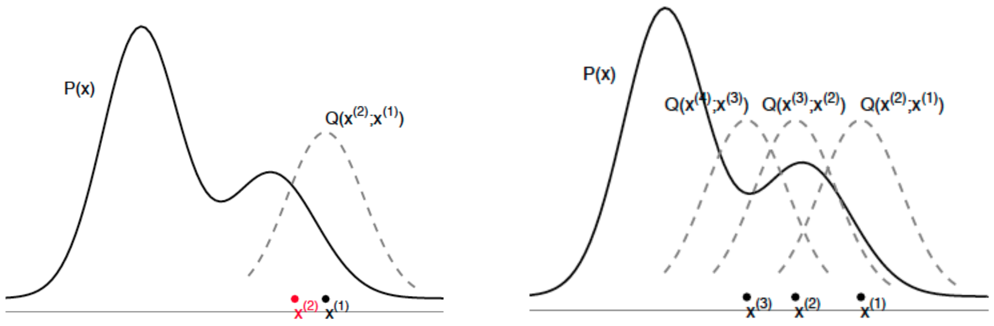
Rejection Sampling: We have a complicated distribution $P(x)$ and a sampler density $Q(x)$. We assume there is a constant c such that $cQ(x) > P(x)$ for all x.

1. sample x from $Q(x)$
2. sample u uniformly from interval $[0, cQ(x)]$
3. if $u < P(x)$ then accept it else reject it



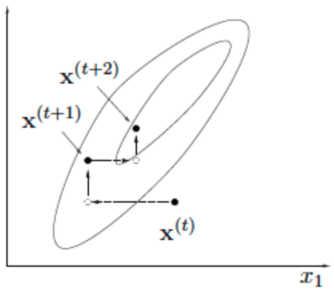
Metropolis–Hastings method:

1. draw sample x' from $Q(x'; x^{(t)})$ where $x^{(t)}$ is the current sample
2. The new sample is accepted or rejected with some quantity $a = \dfrac{P(x')}{P(x^{(t)})} \cdot \dfrac{Q(x^{(t)}; x')}{Q(x'; x^{(t)})}$
3. If $a > 1$ then the new state is accepted, Otherwise, the new state is accepted with probability a
4. If the sample is accepted: $x^{(t+1)} = x'$, else $x^{(t+1)} = x^{(t)}$



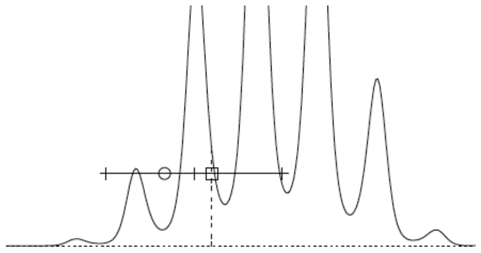
Gibbs sampling: A method for sampling from distributions over at least two random variables. Sampling from conditional distribution of each random variable once at a time.

1. initialize starting values for $x_1^{(1)}, ..., x_k^{(1)}$
2. Do until convergence: Sample each dimension condition on the other dimensions



Slice sampling: Slice sampling is a Markov chain Monte Carlo method. It can be applied wherever the Metropolis method can be applied, that is, to any system for which the target density $P(x)$ can be evaluated at any point x.

1. Choose starting point x_0, evaluate $P(x_0)$
2. Draw a vertical coordinate $u' \sim Uniform(0, P(x))$
3. Create a horizontal interval (x_l, x_r) enclosing x
4. Modify the interval until both ends of the interval be placed above $P(x)$
5. Draw $x \sim Uniform(x_l, x_r)$
6. If $P(x) > u'$ accept x as a sample, else modify interval

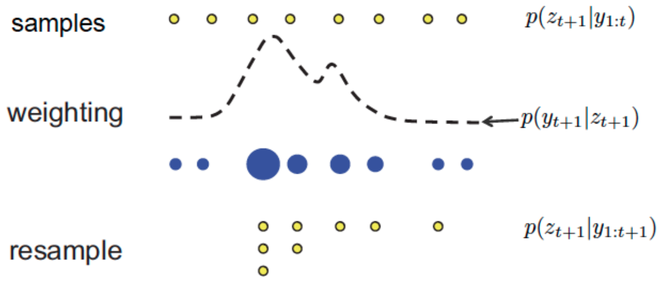


Particle Filtering: A generic solution that involves importance sampling sequentially through time.

1. We have available a collection of samples, or particles drawn randomly from the filtering density at time t
2. We do prediction using these samples
3. Then we perform the correction step using Bayes theorem to get the filtering density at time $t+1$

The basic sequential Monte Carlo sampling algorithm fails after a few steps because most of the particles will have negligible weight. That is called the **degeneracy problem**. Key idea, resampling:

1. Eliminate particles with low importance weights
2. Multiply particles with high importance weights



Sequential Importance Sampling:

1. We start off with many particles drawn from the filtering distribution. Taking each particle in turn and generating a new state from the state transition density according to $z_{t+1}^{(s)} \sim p(z_{t+1} \mid z_t^{(s)})$. Each pair is now a joint random sample from $p(z_{t+1}, z_t^{(s)} \mid y_{1:t})$.
2. By construction, $z_{t+1}^{(s)}$ taken on its own is a random sample from the required marginal distribution $p(z_{t+1} \mid y_{1:t})$.
3. We now have samples from predictive density.

Rao-Blackwellised Filtering: tbd.

Recurrent Neural Networks

Deep Neuronal Networks: A neuron's activation can be defined by passing the weighted inputs through an activation function f taking the neuron's bias b into account

$$y = f\left(\sum w_i x_i + b\right)$$

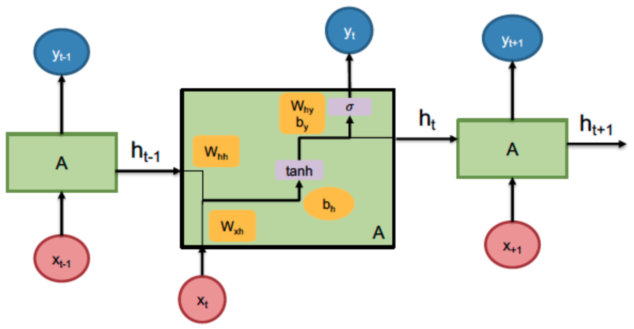
For our set of training samples x we define the cost C as

$$C(w,b) = \frac{1}{2n}\sum ||y(x) - a(x,W,b)||^2$$

where $a(x)$ is the actual obtained output in the last layer L of the network.

Recurrent Neuronal Networks

output and hidden state:	$y_t = \sigma(h_t) = \sigma(W_{hy}h_t + b_y)$
activation function:	$h_t = tanh(h_{t-1}, x_t)$
hidden state:	$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$



Compute the gradient of the loss: $\nabla C = [\nabla W_{xh}, \nabla W_{hh}, \nabla W_{hy}, \nabla b_h, \nabla b_y, \nabla h]$

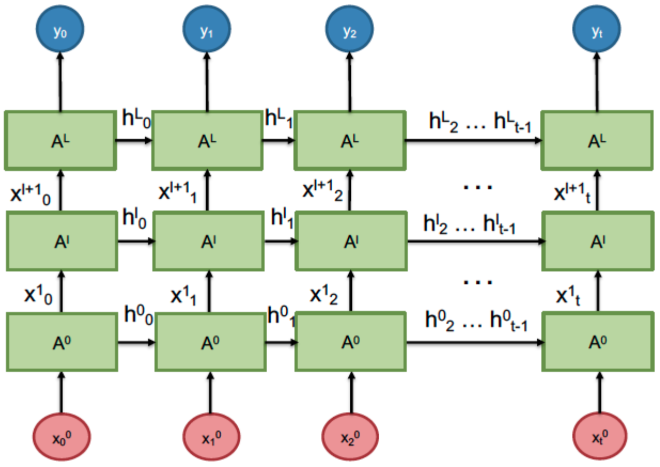
Problem: backpropagating through the whole sequence the parameter update is computational expensive

$\rightarrow$ truncated backpropagation through time

But the backward pass is based on the chain rule:

- if weights are large, the gradients grow exponentially
- if weights are small, the gradients shrink exponentially

Stacked / Deep Recurrent Neuronal Networks

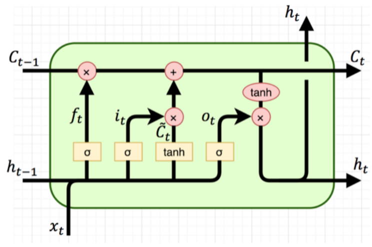


hidden state and output update are now:

$$h_t^l = tanh(W_{hh}^l h_{t-1}^l + W_{xh}^l x_t^l + b_h^l)$$
$$y_t^l = x_t^{l+1} = \sigma(W_{hy}^l h_t^l + b_y^l)$$

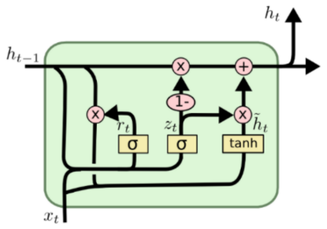
Long Short-Term Memory Networks

- Forget gate: Decide how much of the previous cell state will be forgotten
- Input gate: Decide what information we are going to store in the cell state
- Combining values: Update the old cell state c_{t-1} into c_t
- Output gate: Define output based on the cell state



Gated Recurrent Units

- Reset gate: Defines the relevance of the previous hidden state and the input
- Update gate: Defines the influence of the previous hidden state and the input on the cell state update
- Update candidate: Compute possible update based on r_t and the input h_{t-1} and x_t
- Update the hidden state: Combine old hidden state with the new hidden state candidate



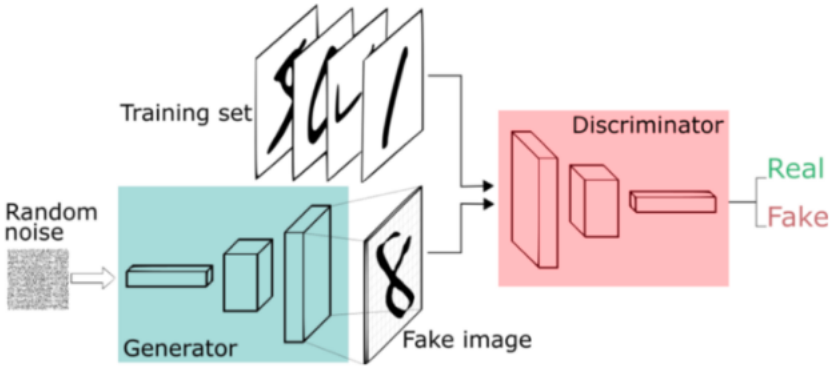
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Overfitting and Regularization:

- L^2 Parameter Regularization: add penalty term to cost function $C = -\frac{1}{2n} \sum ||y - a^L||^2 + \frac{\lambda}{2n} \sum w^2$
- Dropout: Temporarily delete half the hidden neurons, randomly selected

Generative Adversarial Networks

Generative Adversarial Networks:



Domain Adaptation

Given two domain with realted tasks:

- $D^s = \{X^s, p(X^s)\}$ with $T^s = \{Y^s, p(Y^s \mid X^s)\}$
- $D^t = \{X^t, p(X^t)\}$ with $T^t = \{Y^t, p(Y^t \mid X^t)\}$
- homogeneous: same feature space $X^t = X^s$
- heterogeneous: different feature space $X^t \neq X^s$

$D^S = D^T$	$T^S = T^T$	
Yes	Yes	classical ML
Yes	No	Inductive TL
No	Yes	Transductive TL (= DA)
No	No	Unsupervised TL

Instance re-weighting methods (for homogeneous DA): $X^t = X^s, p(X^t) \neq p(X^s), Y^t = Y^s$;
the conditional distributions are assumed shared between the two domains $\rightarrow p(Y \mid X^s) = p(Y \mid X^t)$

i.e. Transfer Adaptive Boosting (TrAdaBoost): Iteratively re-weights both source and target examples during the learning of a target classifier. Done by increasing the weights of miss-classified target instances and decreasing the weights of miss-classified source samples.

Parameter adaptation methods: Not necessary to assume $p(Y \mid X^s) = p(Y \mid X^t)$ $\rightarrow$ Investigates different options to adapt the classifier trained on the source domain (e.g. an SVM), in order to perform better on the target domain.

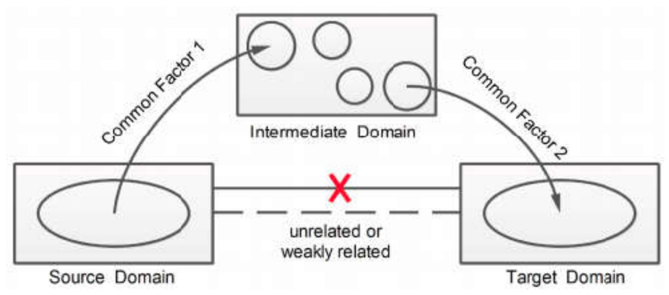
i.e. Adaptive SVM: a set of so called perturbation functions ∇_f are added to the source classifier f^* to progressively adjusts the decision boundaries of f^s for the target domain.

Feature augmentation: The original representation is augmented with itself and a vector of the same size filled with zeros as follows:

$$\text{source: } \begin{pmatrix} x^s \\ x^s \\ 0 \end{pmatrix} \quad \text{target: } \begin{pmatrix} x^t \\ 0 \\ x^t \end{pmatrix}$$

Then an SVM is trained on these augmented features to figure out which parts of the representation is shared between the domains and which are the domain specific ones.

Heterogeneous DA:



Reeinforcement Learning

Markov Decision Process (MDP) is a tool to formulate RL problems: MDP = $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$

- State $\mathcal{S}$ is the information used to determine what happens next: $\mathbb{P}(S_{t+1} \mid S_1, ..., S_t) = \mathbb{P}(S_{t+1} \mid S_t)$
- Actions $\mathcal{A} : \mathcal{S} \to \mathcal{S}$
- State transition model $\mathcal{P}$ helps to model the true (unknown) state transition function
- Reward function $\mathcal{R}_S = \mathbb{E}[R_{t+1} \mid S_t]$
- Discount factor γ: Decay value of rewards over time, total return $G = \sum_t \gamma^t R_t$

Expected long-term value of state s: $v(s) = \mathbb{E}(G)$

$\to$ we need policy π that helps us select the actions to maximize $\mathbb{E}(G)$

- deterministic policy: $a = \pi(s)$
- stochastic policy: $\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$

Dynamic Programming:

state-action-value function: $Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_t \gamma^t R_t \mid S_0 = s, A_0 = a\right]$

state-value function: $V^\pi(s) = Q^\pi(s, \pi(s))$

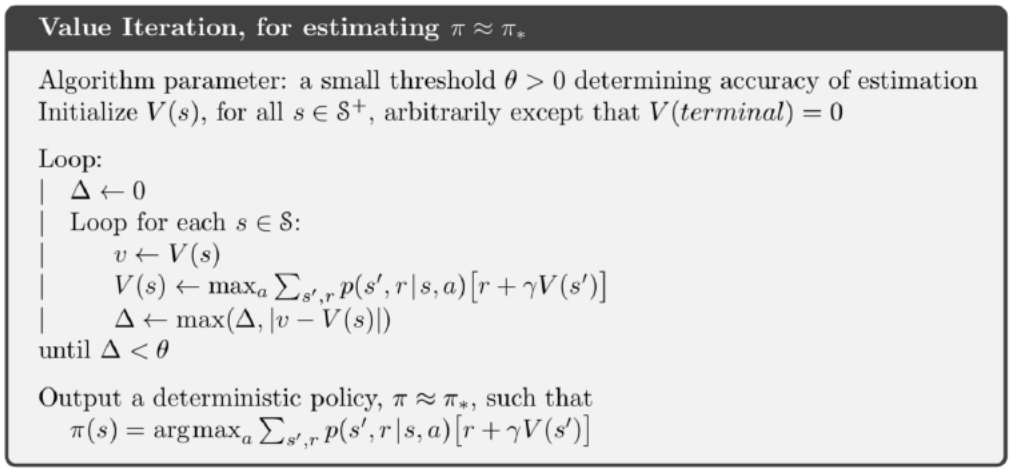
Bellman Equation:

$$Q^\pi(s,a) = \underbrace{\mathcal{R}(s,a)}_{\text{first step}} + \gamma \sum_{s' \in \mathcal{S}} \underbrace{\mathcal{P}(s' \mid s,a)}_{\text{transition}} \underbrace{Q^\pi(s', \pi(s'))}_{\text{expected return}}$$

$$V^\pi(s) = \underbrace{\mathcal{R}(s,\pi(s))}_{\text{first step}} + \gamma \sum_{s' \in \mathcal{S}} \underbrace{\mathcal{P}(s' \mid s,\pi(s))}_{\text{transition}} \underbrace{V^\pi(s')}_{\text{expected return}}$$

Optimal policy:

$$V^{\pi^*}(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a) V^{\pi^*}(s') \right\}$$

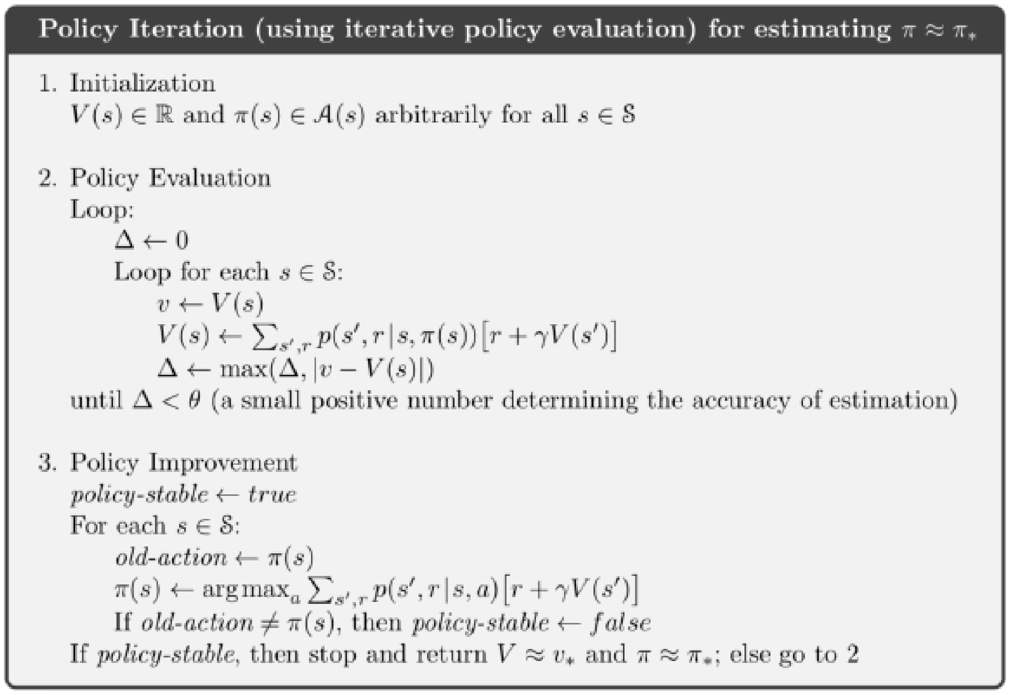


Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|  $\Delta \leftarrow 0$
|  Loop for each $s \in \mathcal{S}$:
|      $v \leftarrow V(s)$
|      $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
|      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
   $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
```



Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

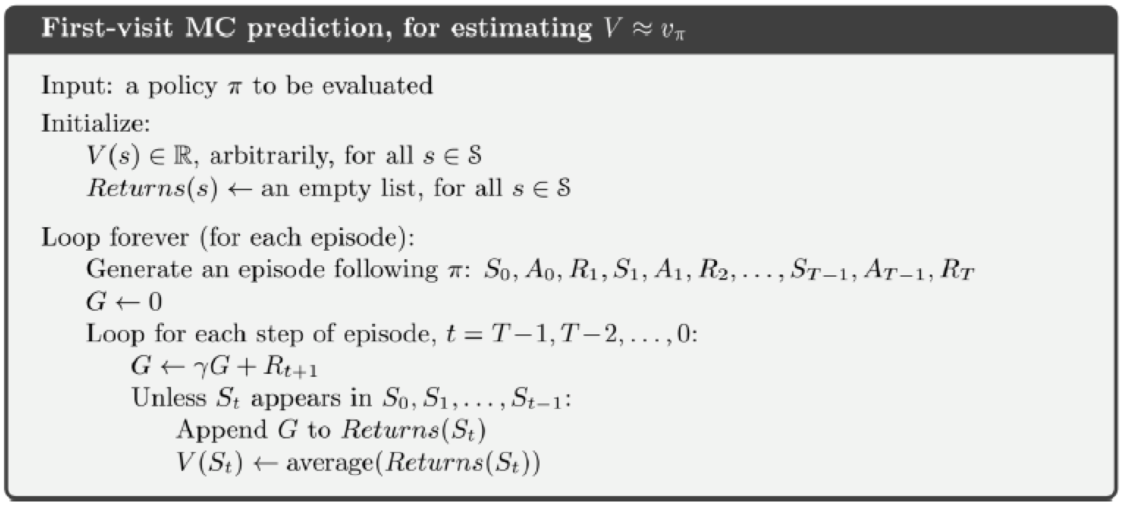
```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r \,|\, s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
       *old-action* $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
       If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

DP methods are guaranteed to find optimal solutions for Q and V in polynomial time. Policy Iteration computes the value function under a given policy to improve the policy while value iteration directly works on the states.

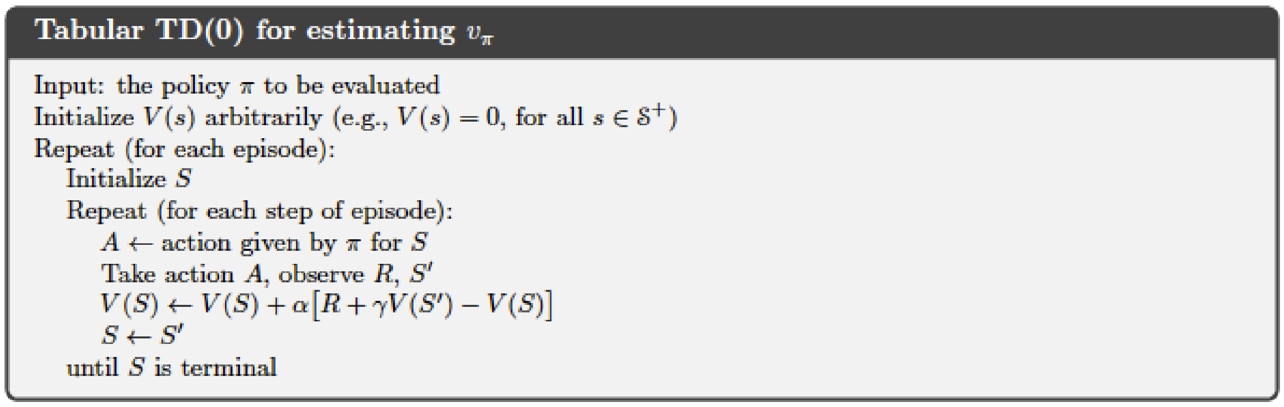
Monte Carlo Methods: Use experience samples to estimate the true V- and Q-value functions for policy π



First-visit MC prediction, for estimating $V \approx v_\pi$

```
Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$)
```



Tabular TD(0) for estimating v_π

```
Input: the policy $\pi$ to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$)
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

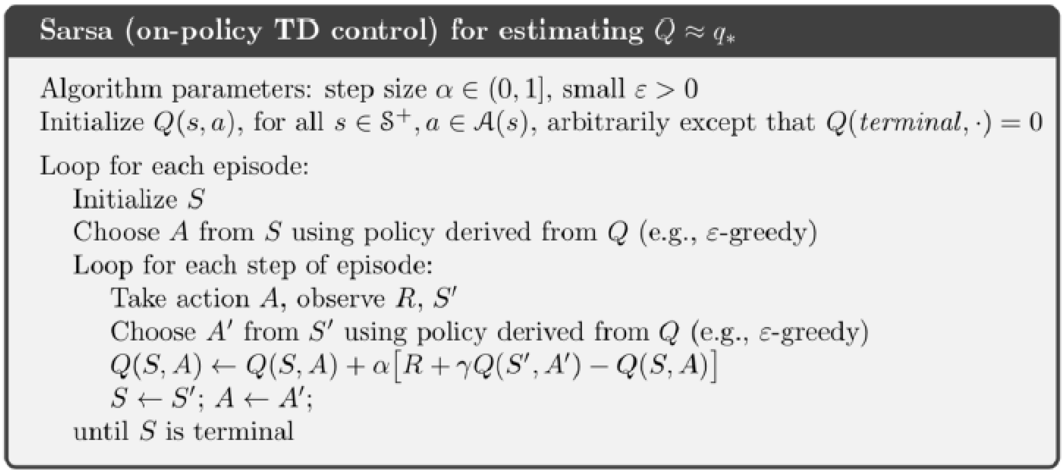
Temporal Difference

- can learn before/without knowing the final outcome
- has low variance, but some bias
- more efficient in Markov environments (exploits Markov property)
- usually converges faster than MC

Monte Carlo

- only works for episodic problems
- has high variance, but zero bias
- is more efficient in non-Markov environments

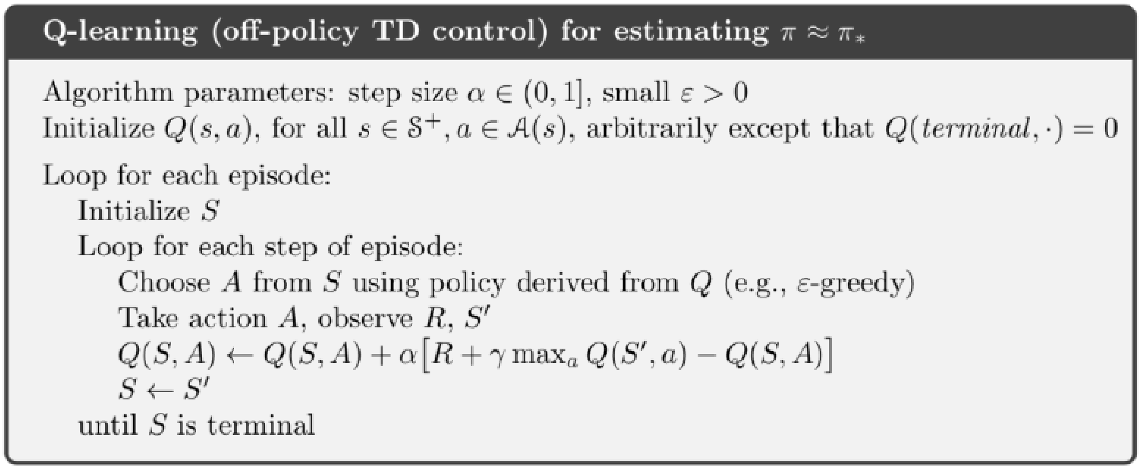
Q-Learning and SARSA: If we have calculated the value function for a given policy π, we can use it for deriving a better policy π' through greedy policy improvement over $V(s)$. Problem: Leaves areas of the state space unexplored. $\rightarrow$ Either take the best action or explore the action space with $\varepsilon =$ „probability of exploration“



```
Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  Loop for each step of episode:
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until $S$ is terminal
```



```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until $S$ is terminal
```

SARSA applies TD to $Q(s,a)$ and uses ε-greedy policy improvement at every time-step. Q-Learning evaluates one policy while following another and can re-use experience gathered from old policies.

Value Function Approximation: Describe a state using a vector of features. Features are functions from states to real numbers that capture important properties of the state. Our goal is to learn good parameters w that approximate the true value function well:

$$\begin{aligned}\hat{Q}^{\pi}(s,a;w) &= \phi(s,a)^T w \\ C &= \left(Q^+(s,a) - \hat{Q}^{\pi}(s,a;w)\right)^2 \\ \frac{\partial C}{\partial w} &= -2\phi(s,a)\left(Q^+(s,a) - \phi(s,a)^T w\right)\end{aligned}$$

$$\begin{aligned} Q^+(s,a) \quad = \quad & r + \gamma \max_{a'} Q(s',a') && \rightarrow \text{Q-Learning with linear VFA} \\ & r + \gamma Q(s',a') && \rightarrow \text{SARSA with linear VFA} \\ & G_t && \rightarrow \text{MC with linear VFA} \end{aligned}$$

In general, every value function can be approximated with linear FA but it's really hard to find some!

Deep Q-Networks: A Convolutional NN reads the image from the game, the CNN is a value function approximator for the Q-function, the reward is the game score.