term memory c_t is updated to include information from x_t, h_{t-1} while maintaining old information in c_{t-1}, which is given by

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ \tilde{c}_t &= \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right) \end{aligned}$$

where the ratio of storing new information and maintaining old information is controlled by the input gate value i_t and the forget gate value f_t. The use of forget gate enables information stored in c_t change slowly and last longer, therefore c_t is called the long term memory.

Information stored in long terms c_t will output to short-term memory h_t via $h_t = o_t \odot \tanh(c_t)$ where $o_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right)$. Note that the output gate o_t is affected by input x_t and short-term memory h_{t-1}.

The updated short-term memory h_t, which contains both short-term and long-term information, will be subsequently connected to additional feed-forward layers specific to tasks.

Note 35.2.1 (forget gate initialization to enable gradient flow). In practice, to make most of forget gate to cope with the vanishing gradient issue, we usually initialize b_f close to 1 (so f_t will be close 1)to enable gradients carry over multiple time steps.

In subsection 35.1.3, we have quantitatively discussed that BPTT in simple RNN can cause gradients to vanish, which ultimately prevents learning long-term relationship from sequences. Now we look at how LSTM quantitatively alleviates the gradient vanishing issues.

Because the cell state maintains long term memory, we want to see if gradient flow could suffer exponential decay as in a vanilla RNN. We look at the derivative for $\frac{\partial c_t}{\partial c_{t-1}}$. Note that

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ \tilde{c}_t &= \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right) \\ i_t &= \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) \\ f_t &= \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \end{aligned}$$

so we have

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t + \frac{\partial f_t}{\partial c_{t-1}} \odot c_{t-1} + \frac{\partial i_t}{\partial c_{t-1}} \odot \tilde{c}_t + i_t \odot \frac{\partial \tilde{c}_t}{\partial c_{t-1}}.$$

We are not going to pursue further details; instead, the key idea in LSTM is that there are multiple learnable components controls the damping or growing effects on each step via gates. The network can learn to set the gate values, conditioning on the current input and hidden state, to decide when to damp gradient to eliminate long term influence, and when to grow the gradient to preserve the long term dependence. On the contrary, the vanilla RNN lacks such flexibility (RNN only has $W_h h$ to control the gradient change) and the gradient either explodes or vanishes for long sequences.

35.2.2 Gated Recurrent Unit (GRU)

Gated recurrent unit [5] is a simpler RNN variant than LSTM (e.g., no cell states) but is still endowed with key features of LSTM for long-term dependence modeling. GRU employs an **update gate** $z_t \in [0, 1]$ and a **reset gate** $r_t \in [0, 1]$ to control information flow. The output of the hidden layer is then given by

$$
\begin{aligned}
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \\
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
\tilde{h}_t &= \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r)
\end{aligned}
$$

where $\tilde{h}_t$ is a candidate state. The reset gate controls the dependence of $\tilde{h}_t$ on the previous h_{t-1}, and the update gate controls the output h_t as the mixture of h_{t-1} and $\hat{h}_t$. Notably,

- When $z = 0, r = 1$, GRU becomes simple RNN;
- When z is large, most historical information is preserved.

The overall information flow scheme scheme can be summarized in Figure 35.2.2.

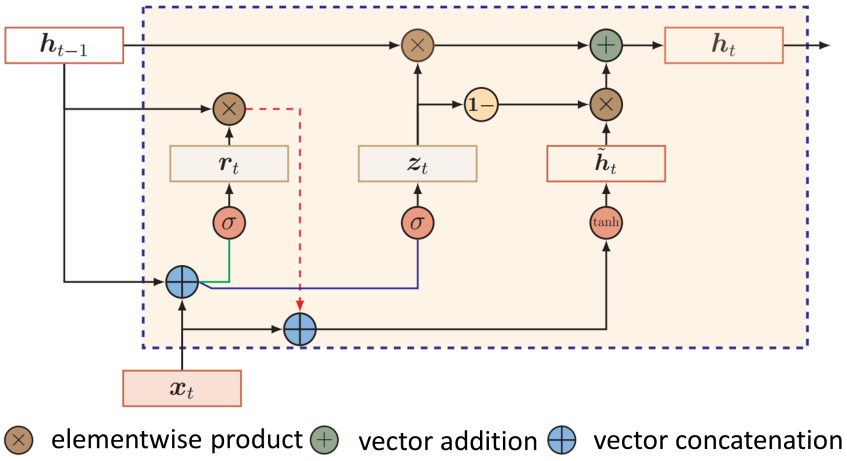


Figure 35.2.2: Scheme of a GRU cell. Modified from [2, p. 152].

And the mathematical relation can be summarized by

Definition 35.2.2 (GRU cell mathematical relations). *A GRU cell takes input x_t and previous output h_{t-1} and outputs h_t via the following relations:*

$$
\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
\tilde{h}_t &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
\end{aligned}
$$

where parameter and inputs dimensions include $h_t \in \mathbb{R}^H$, $x_t \in \mathbb{R}^D$, $r_t, z_t \in \mathbb{R}^H$, $W_z, W_r \in \mathbb{R}^{H \times D}$, $U_z, U_h, U_r \in \mathbb{R}^{H \times H}$, and $b_z, b_r, b_h \in \mathbb{R}$.

35.3 Common RNN architectures

35.3.1 Stacked RNN and bidirectional RNN

We have covered several most commonly used recurrent units Figure 35.3.1. These recurrent units can be used with other neural network units, which are collectively called recurrent neural networks, to perform general regression and classification tasks.

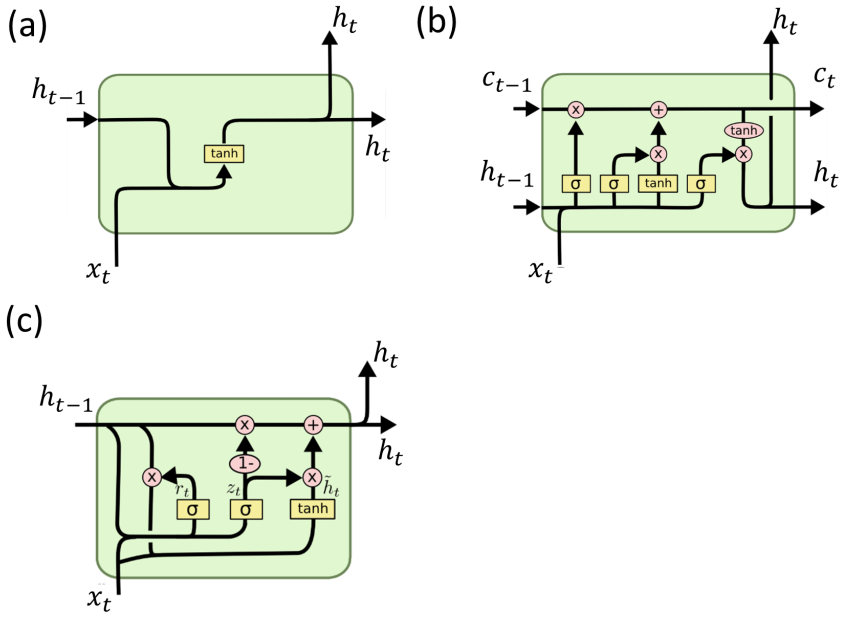
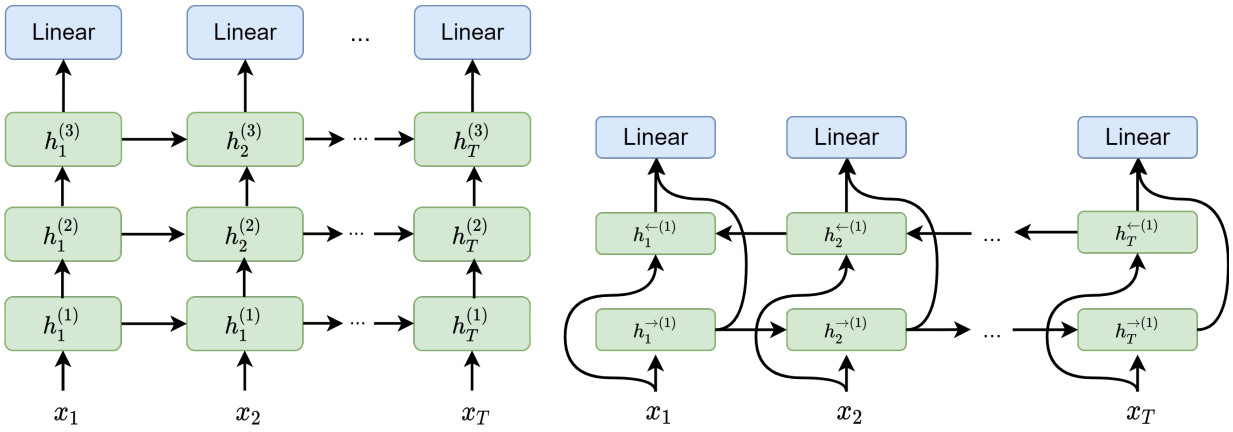


Figure 35.3.1: Different types of units in RNNs: (a) Vanilla RNN cell. (b) LSTM cell. (c) GRU cell. Credit

First,to enable recurrent layer to learn more high-level representative features in the sequence (e.g., in complex time series prediction and language modeling), we can stack multiple recurrent layers together and form stacked RNN [Figure 35.3.2a].



(a) Scheme for stacked RNN. **(b)** Scheme for bidirectional RNN.

Figure 35.3.2: Stacked RNN and bidirectional RNN.

In the stacked RNN with input dimension D and hidden unit dimension H, the weight matrices associated with first layer RNN cells are $W_{ih}^{(1)} \in \mathbb{R}^{D \times H}$ and $W_{hh}^{(1)} \in \mathbb{R}^{H \times H}$; the weight matrices associated with second layer RNN cells are $W_{ih}^{(1)} \in \mathbb{R}^{H \times H}$ and $W_{hh}^{(1)} \in \mathbb{R}^{H \times H}$ since first layer RNN cell output are the input to the second layer RNN cells.

Another important architecture of RNN is bidirectional RNN [Figure 35.3.2b], which can be used for context modeling with context from both past and future. Bidirectional RNN has one layer reverse recurrent layer flowing subsequent input information to preceding hidden states (e.g., from x_T to x_1).

Finally, we can combine stacking and bidirectionality together and get stacked bidirectional RNN [Figure 35.3.3]. Example application includes tagging a word's entity and machine translation in where the meaning and tag of a word usually depends on its preceding and subsequent surrounding words.

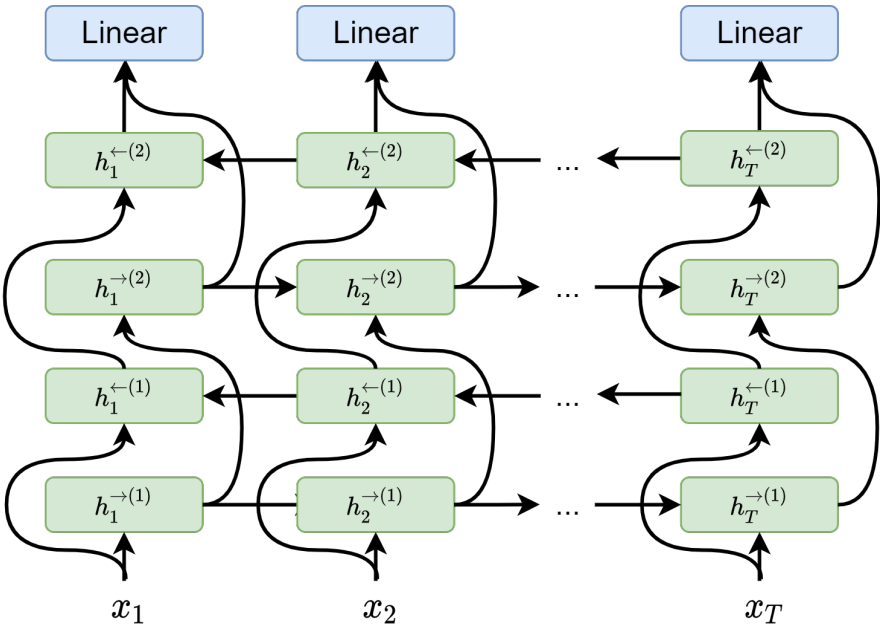


Figure 35.3.3: Stacked bidirectional RNN

35.3.2 Task dependent output layers

Without RNN extracts features from sequential inputs, we use different output layers to solve different tasks [Figure 35.3.4]. The architecture in which recurrent output of every time step t connects to the output layer [Figure 35.3.4(a)] is suitable for modeling dynamic system represented by $y_t = f(x_{1:t})$ (e.g., time series prediction), where y_t has dependency on $x_1, ..., x_t$. On the other hand, for sequence data classification problem, we only need one output vector given the whole sequence data, then we can use architecture like Figure 35.3.4(b)(c).

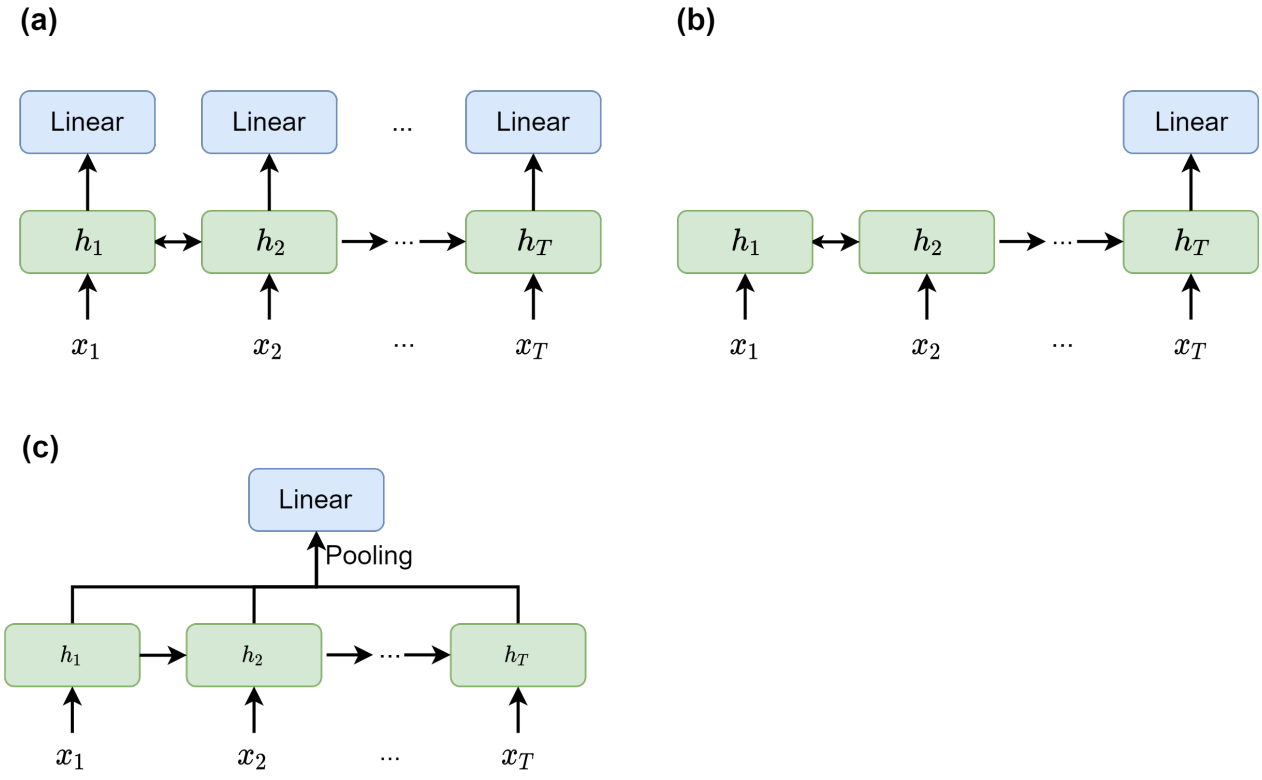


Figure 35.3.4: Typical RNN connection to the output layer.

35.3.3 Dropout

Like feed-forward multi-layer neural network, dropout can be applied to recurrent neural network to improve robustness and generalization[6]. Dropout can be applied at the input, output, and recurrent connections [Figure 35.3.5]. Given a sequence of inputs, it is unclear whether dropout mask applied should be different or same for every time step. [7] points out that the theoretically sound approach is to apply the dropout mask for every time step.

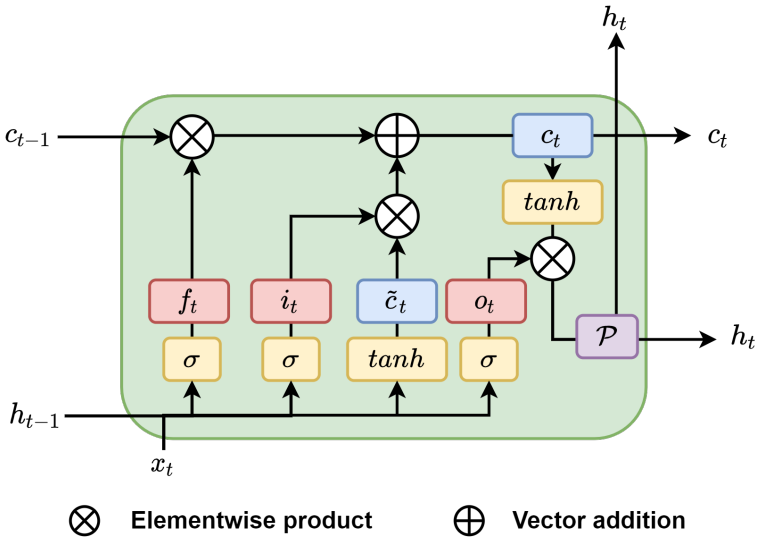


Figure 35.3.6: LSTM cell with a projection layer (purple box) added after the hidden output h_t.

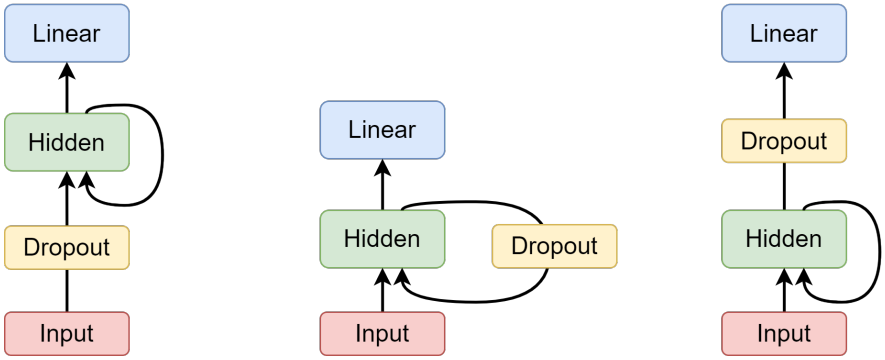


Figure 35.3.5: Three places to add Dropout to a RNN: at the input, at the recurrent connection, and at the output.

35.3.4 LSTM with recurrent projection layer

A recent improvement on LSTM [8] in terms of effective use of model parameters and reducing computational cost is to introduce a projection layer after the hidden output h_t, as shown in Figure 35.3.6.

This additional projection layer changes the LSTM cell in the following way. Let the $\tilde{h}_t$ be the original hidden output given by Definition 35.2.1. The project layer is associated with a weight matrix $W_{ph} \in \mathbb{R}^{P \times H}$ such that the new hidden output $h_t \in \mathbb{R}^P$ is given by

$$h_t = W_{ph}\tilde{h}_t.$$

Here P is the projection size.

Additional changes in the LSTM cell include

- While the hidden h_t has dimension P, cell states c_t still has dimension H.
- Since all hidden h_t has dimension P, all matrices associated with gates will have size $H \times P$.

35.4 RNN application examples

35.4.1 Time series prediction

35.4.1.1 *Simple RNN prediction*

The most straight forward application of RNN is time series modeling. RNN can model complex, nonlinear time series beyond linear time series we consider in chapter 22. A basic architecture for nonlinear time series with possible long memory dependence is given by Figure 35.4.1.

Time series prediction problems are generally formulated as a regression problem. Let $x_1, x_2, ..., x_N, x_i \in \mathbb{R}^D$ be an example time series observation. Let the context window by K, we can transform the time series observation into training samples with input and outcome pairs like

$$(x_1, ..., x_k), x_k; (x_2, ..., x_{k+1}), x_{k+1}; (x_1, ..., x_k), x_{k+1}; \cdots$$

The training examples are then fed into RNN and yield predicted values [Figure 35.4.1(b)]. The loss function can simply be the mean squared error between the outcome and the predicted value.

In the prediction stage, RNN can predict more than the immediate next step by constantly feeding preceding predicted values to the network [Figure 35.4.1(b)]. As a demonstration, we train a RNN to predict time series

$$y(t) = f(x(t), x(t-1), ...),$$

where $x(t) = \cos(t), y(t) = \sin(t)$.

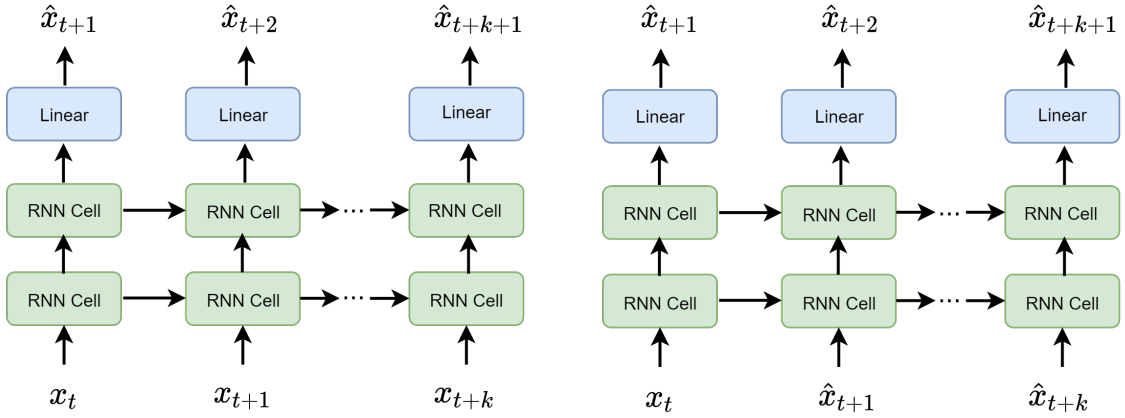


Figure 35.4.1: RNN architecture for time series prediction . (left) In the training phase, RNN are updated by minimizing the next step prediction error. (right) In the prediction phase, trained RNN is used to sequentially predict next step state value based on preceding predicted state value.

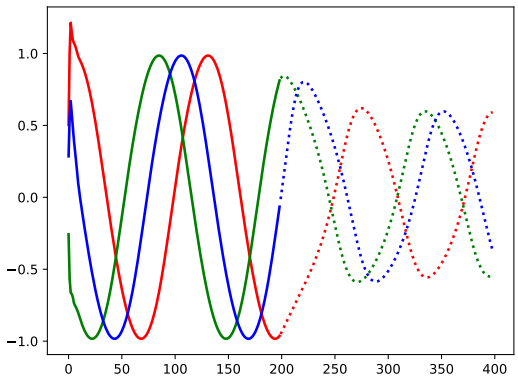
Consider a toy model. The training data are 1000 trajectories of sine wave data with random initial phases ϕ

$$
\begin{aligned}
&(\sin(\frac{0}{T}+\phi^0), \sin(\frac{1}{T}+\phi^0, ..., \sin(\frac{N}{T}+\phi^0) \\
&(\sin(\frac{0}{T}+\phi^1), \sin(\frac{1}{T}+\phi^1, ..., \sin(\frac{N}{T}+\phi^1) \\
&\dots \\
&(\sin(\frac{0}{T}+\phi^{1000}), \sin(\frac{1}{T}+\phi^{1000}, ..., \sin(\frac{N}{T}+\phi^{1000})
\end{aligned}
$$

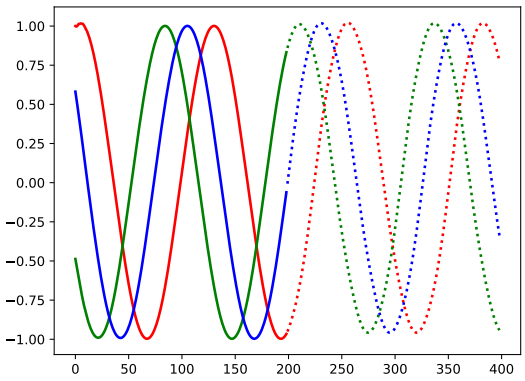
where $N = 10T, T = 20$.

We use a two-layer stack RNN and train for 50 episodes [Figure 35.4.2]. After training, we use 3 out of sample time series to examine the one-step forward and multiple forward prediction performance. We have following observations:

- After 2 episodes of training, we can see that one step forward prediction straggles only when input sequence is short (i.e., less than 10). Multi-step forward prediction has poor performance.
- After 17 episodes of training, we can see good performance at both one-step forward and multi-step forward prediction.



(a) RNN (after 2 episode's training) predicted values with 3 different observed time series of 200 steps as the input. The solid lines are one-step forward predictions; the dashed lines are multi-step forward predictions.



(b) RNN (after 17 episode's training) predicted values with 3 different observed time series of 200 steps as the input. The solid lines are one-step forward predictions; the dashed lines are multi-step forward predictions..

Figure 35.4.2: RNN one-step forward and multiple forward prediction performance for Sine time series.

In the previous architecture [Figure 35.4.1], the input are the historical observations of the time series. Now we consider a more general architecture that allows the input to include additional covariate time series z_t [Figure 35.4.3]. Covariate time series are sequential observations of variables that variables that have some relationship with the time series. We usually assume covariate time series are available during future horizon (they are from either external model prediction or from user input). The use of covariate time series are quite common in practice as a means to enhance prediction performance. For example, let the stock price of APPLE be the time series of our interest, and SP500 index be the covariate time series. Predicting APPLE stock price given SP500 observation will be easier than directly predicting APPLE stock price from its historical prices, as there is strong correlation between SP500 and APPLE stock.

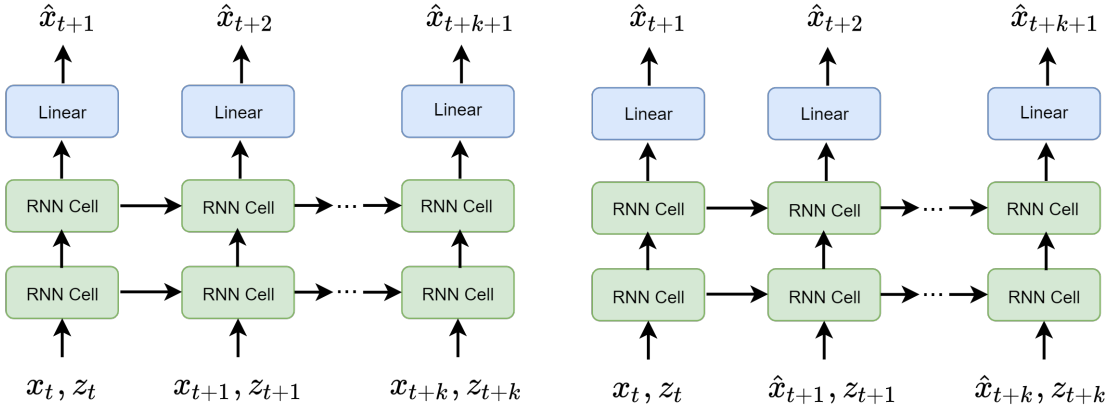


Figure 35.4.3: RNN architecture for time series prediction with covariates. (left) In the training phase, RNN are updated by minimizing the next step prediction error. (right) In the prediction phase, trained RNN is used to sequentially predict next step state value based on preceding predicted state value. Note that covariate time series are assumed available for all time steps.

35.4.1.2 *Deep autoregressive (DeepAR) model*

Previous simple RNN architectures [Figure 35.4.1, Figure 35.4.3] can be used to predict deterministic time series or to predict mean levels for stochastic time series. Deep autoregressive (DeepAR) model [9] is an extension that combines statistical modeling methods and RNN. DeepAR models allows we make additional distribution assumption on the time series.

Suppose We are given a number of time series $x_t^{(i)}$ where i is the time series index and t is the time index; and a number of covariate time series $z_t^{(i)}$. The goal is to learn the conditional distribution of the future distribution of x_t of the time series, given its past and the covariates.

More formally, denote $[x_t, x_{t+1}, \ldots, x_T] := z_{t:T}$, the goal is to model the conditional distribution $P\left(x_{T+1}|z_{t:T+1}, x_{1:T}\right)$. In DeepAR model architecture [Figure 35.4.4], we assume the conditional distribution follows some parametric forms and use RNN to learn the distribution parameters. For example, if assumed conditional distribution is Gaussian, then the outputs are parameters including the mean μ_t and standard deviation σ_t for prediction $\hat{x}_{T+1}$.

The network parameters for the feed-forward module and the recurrent modules are trained by minimizing the negative log likelihood function. If we assume z_t follows Gaussian, then the loss function is given by

$$L\left(x_t|\hat{\mu}_t,\hat{\sigma}_t\right) = -\sum_{i=1}^{N}\log\left(2\pi\sigma_i^2\right)^{-\frac{1}{2}} + \sum_{i=1}^{N}\left(\frac{\left(z_t^{(i)}-\mu_t\right)^2}{2\sigma_t^2}\right),$$

where $\{x_t\}$ are the outcomes in the training examples, $\{\mu_t\}, \{\sigma_t\}$ are predicted parameters for the conditional distribution.

During prediction, the time series values in the prediction horizon are not available (because that's what we are trying to predict). Therefore, samples from the conditional distribution (whose parameters are predicted during the previous step) are used as the input values for the current time step.

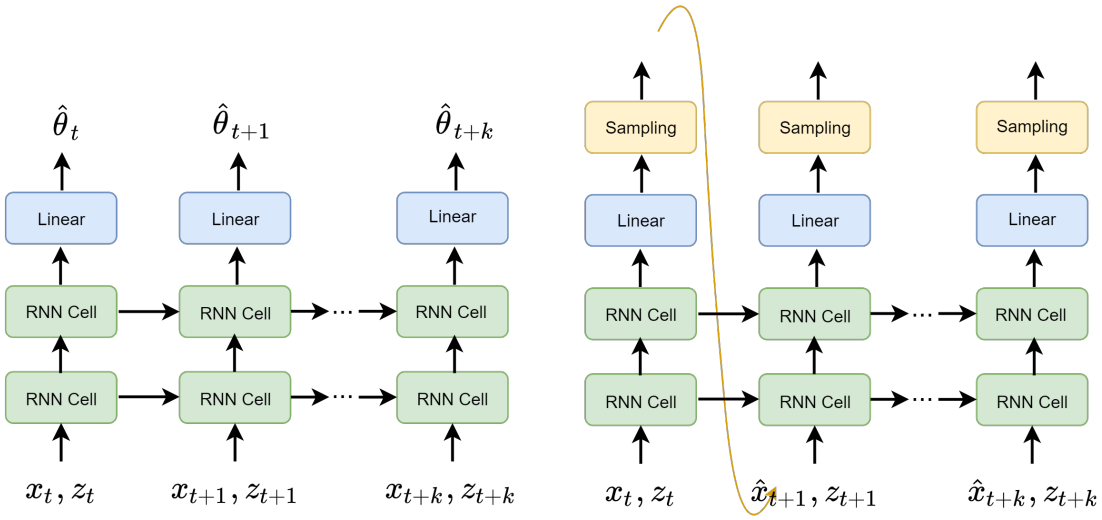


Figure 35.4.4: DeepAR model architecture for time series prediction. Outputs are the parameters characterizing the condition distribution of x_{t+1} conditioned on histories of x_t and z_t. (a) In the training phase, RNN are updated by minimizing the negative log likelihood function. (b) In the prediction phase, a predicted $\hat{x}_{t+1}$ are sampled from predicted conditional distribution and then used to predict next step conditional distribution.

35.4.1.3 *Deep factor model*

In modeling multivariate stochastic time series, a linear factor model represents individual time series to a few common global factors via

$$x_{i,t} = \sum_{k=1}^{K}\beta_{ik}g_{k,t} + r_{i,t},$$

where $x_{i,t}, i = 1, ..., N$ are the multivariate time series to model, $g_k, k = 1, .., K, K \ll N$ are the global factor time series, and r_i are the individual random effects time series following common zero mean stochastic process (e.g., Gaussian processes) .

Clearly, global factor time series are playing critical roles in the following aspects:

- Capturing the dominant collective dynamics coexist among time series.
- Revealing latent structures among time series and offering model interpretation (i.e., explaining the dynamics via global factors).

Usually, the global factor time series are constructed as a linear combination of $\{x_{i,t}\}, \{z_i\}$ via PCA or reduced rank multivariate regression. However, these linear constructed global factors often fall short in facing complex time series modeling challenges.

One approach to construct more competent global factor time series is to harness RNN's capability of capturing complex pattern and structure among time series. The resulting model is known as **deep factor model** [10].

In the deep Factor model, each time series $x_i(t)$ is governed by a non-random global component and a random component, which can be written by

$$x_{i,t} = \sum_{k=1}^{K} \beta_{i,k} g_{k,t} + r_{i,t}$$

The global factors are given by

$$g_{k,t} = RNN_k(\{x_{i,t}, z_{i,t}\}),$$

where RNN_k denote a RNN taking input sequences of $\{x_i, z_i\}$ and producing one output. In the basic case, we can assume $r_{i,t} \sim N(0, \sigma_{i,t})$, where $\sigma_{i,t}$ is specified by neural networks or other deterministic functions. Beside being simple Gaussian processes, the random effect $r_{i,t}$ can further be modeled by a linear state space model to introduce additional linear structures among the noises, given by [10]

$$\begin{aligned} h_{i,t} &= F_{i,t} h_{i,t-1} + q_{i,t} \epsilon_{i,t}, \quad \epsilon_{i,t} \sim N(0, 1) \\ r_{i,t} &= a_{i,t}^{\top} h_{i,t} \end{aligned}$$

where h_i are latent states, F is the transition matrix for h_i, and a is emission coefficient matrix, and $q_{i,t}$ are strength for the random shocks.

35.4.2 MNIST classification with sequential observation

One interesting application of RNN is image classification based on sequential input of small image patches. Contrasting a CNN that directly takes the whole image to the

neural network, RNN can only take one stripe of a image at a time as input. For a 28 by 29 image, one image is then decomposed into 28 such stripes, reminiscent of a multi-dimensional time series of length 28. Different from time series modeling, the RNN architecture only employs the last hidden output for classification. The recurrent nature of the RNN 'stores' all previously observed image stripes in the last hidden output for classification.

The final neural network architecture is showed in Figure 35.4.5. There are 128 hidden units in the LSTM unit. After moderate training, the classification accuracy can achieve 97%, comparable to a simple CNN.

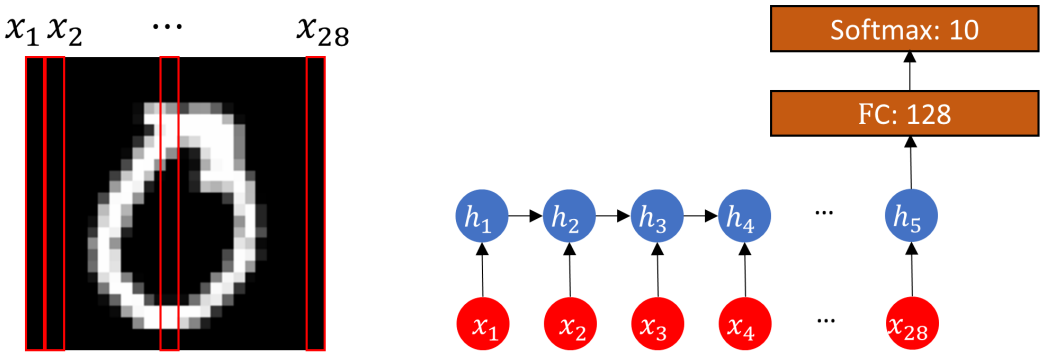


Figure 35.4.5: A RNN architecture for MNIST image recognition.

35.4.3 Character-level language modeling

35.4.3.1 *Word classification*

In sentiment classification, we take a sequence of words as the input and output a sentiment label. In word classification, we take a sequence of characters and output a label characterizing the word.

An example application is to classify names origin via RNN in Figure 35.4.6. The overall architecture contains the following components:

- Characters are fed into the LSTM layer one by one.
- The last output from the LSTM will go to a Softmax output layer to produce classification probabilities on the language origin (Chinese, American, Japanese, etc.).

Note that we usually do not need embedding layer since the one-hot encoding on character level is already of low dimensionality.

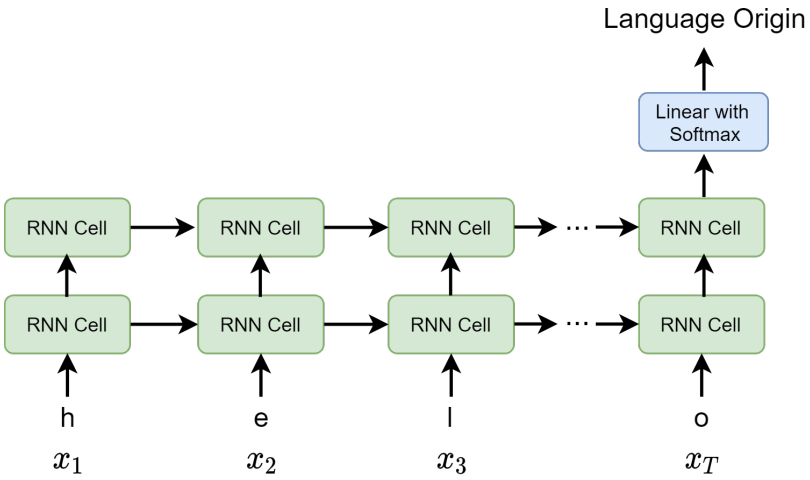


Figure 35.4.6: A RNN for character-level word classification.

35.4.3.2 *Text generation*

RNN has been used for advanced language modeling tasks, like machine translation, caption generation, and chatting robot. Before diving into these advanced applications, we cover one very basic language modeling task in this section, known as character-level language modeling. The basic goal of character-level language modeling is to predict or generate a sequence of sensible characters based on preceding characters.

One concrete application example of character-level language model is when we write an email in Gmail and type messages on a phone, we observe Gmail and our phones will suggest next character/word. Alternatively, a simple word completion program can also be made by n-gram statistical model, which basically counts n-gram frequencies. However, with RNN designed to efficiently capture the long term relation among characters, we expect RNN language modeling can do far more than a empirical n-gram statistical model.

The first step towards the language modeling task is to train a RNN that is able to predict the next character based on an input character sequence [Figure 35.4.7 (left)]. A naive n-gram model would required K^n memory, where K is the vocabulary size (say K = 26 for alphabetic letters). From this perspective, RNN can be viewed as an efficient parametric n-gram model.

The overall RNN architecture contains the following components:

- Characters from a training text corpus (e.g., a English novel) are fed into the LSTM layer one by one.

- Each output from LSTM will go to a Softmax output layer to produce prediction probabilities on the next letter (26 classes).
- Network parameters are updated by taking gradient descent based on a cross-entropy loss function applied to each prediction.

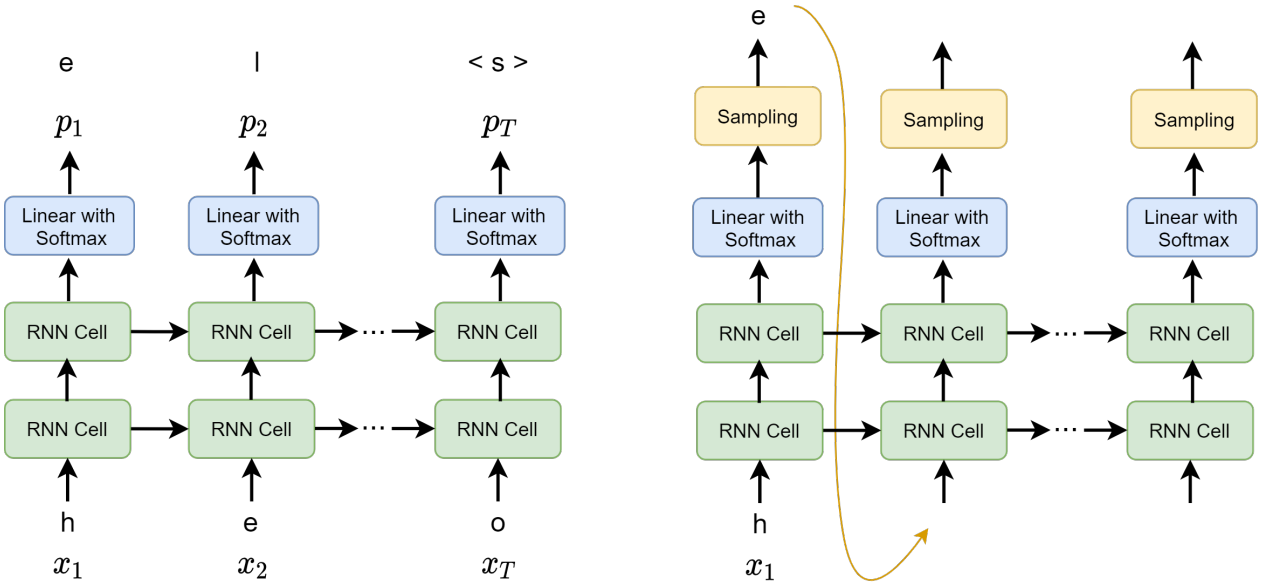


Figure 35.4.7: A RNN for character-level language model. (left) During the training session, one-hot coded characters are directly fed into RNN and predict the next character in the word. (right) A RNN for character-level language model. During the word generation session, the network starts with a user input character and continues the generation process with predicted character from the previous step.

The stage of text generation is to use the trained RNN to sequentially predict new character based on previous predicted character.

The text generation scheme and architecture are showed in Figure 35.4.7 (right). The actual predicted character requires sampling from the prediction probability output by the softmax layer.

There are different sampling strategies from predicted probabilities to promote the total probability of a full generated word. On one extreme, a greedy sampling strategy will take character with the largest probability. But such an approach usually results in repetitive, deterministic patterns that don't look like coherent language. A more general

and flexible strategy is to introduce a temperature controlled sampling strategy, where the sampling probability for character i is given by

$$p_i^T \propto \exp(\frac{p_i^S}{T}),$$

where p_i^S is the sampling probability from softmax output. As $T \to \infty$, we get uniform sampling, and as $T \to 0$, we have greedy sampling. Compared to greedy sampling, probabilistic sampling can generate interesting language patterns that resemble human language.

BIBLIOGRAPHY

1. Haykin, S. *Neural Networks and Learning Machines, 3/E* (Pearson Education India, 2010).
2. Qiu, X. *Neural Network and Deep Learning* (Fudan University, 2019).
3. Pascanu, R., Mikolov, T. & Bengio, Y. *On the difficulty of training recurrent neural networks* in *International conference on machine learning* (2013), 1310–1318.
4. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural computation* **9,** 1735–1780 (1997).
5. Chung, J., Gulcehre, C., Cho, K. & Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).
6. Zaremba, W., Sutskever, I. & Vinyals, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).
7. Gal, Y. & Ghahramani, Z. *A theoretically grounded application of dropout in recurrent neural networks* in *Advances in neural information processing systems* (2016), 1019–1027.
8. Sak, H., Senior, A. W. & Beaufays, F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling (2014).
9. Salinas, D., Flunkert, V., Gasthaus, J. & Januschowski, T. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting* (2019).
10. Wang, Y. *et al.* Deep factors for forecasting. *arXiv preprint arXiv:1905.12417* (2019).

Part VII

OPTIMAL CONTROL AND REINFORCEMENT LEARNING

36

CLASSICAL OPTIMAL CONTROL THEORY

36.1 Basic problem

We start with the basic formulation of a basic optimal control problem, which aims to seek an optimal control policy that maximizes accumulated gain during a dynamical process.

Definition 36.1.1 (basic optimal control problem). *Given a dynamic system*

$$\dot{x}(t) = a(x(t), u(t), t), x(0) = x_0,$$

the basic optimal control problem is to maximize the performance measure

$$\max_{u(x,t)} J(x_0, u(x,t)) = h(x_{t_f}, t_f) + \int_{t_0}^{t_f} g(x(t), u(x,t), t)dt$$

*The functional relationship $u^*(x,t) = f(x(t),t)$ that maximize J is called **optimal control policy** .*

For different concrete types of performance measure function g, see [1, p. 30]. Another common optimal control problem is with respect to an infinite horizon process.

Definition 36.1.2 (optimal control problem infinite horizon under discount). *Given a dynamic system $\dot{x}(t) = a(x(t), u(t), t), x(0) = x_0$, the optimal control problem for infinite horizon is to maximize the performance measure*

$$\max_{u(x)} J(x_0, u(x_0)) = \int_{t_0}^{\infty} e^{-\gamma(t-t_0)} g(x(t), u(x), t)dt,$$

*where $\gamma \in (0,1)$ is the discount factor, and the functional relationship $u^*(x) = f(x)$ that maximize J is called **optimal control policy**.*

In the basic optimal control problem, the control policy is time dependent. For infinite horizon problem, the optimal control policy does not have time dependence. If the optimal control is a function of initial state x_0 and t, that is $u^*(t) = f(x_0, t)$, then the optimal control is **open-loop control.**

36.2 Controllability & observability

A fundamental property of dynamical systems in the context of optimal control is its **controllability**. Intuitively, a dynamical system is controllable we can use finite steps of control to reach any states.

Definition 36.2.1 (controllability for discrete-time linear system). *A n dimensional discrete-time system*

$$x(k+1) = Ax(k) + Bu(k)$$

*is said to be **completely controllable** if for $x(0) = 0$ and given x_1, there exists a finite index N and sequence of control inputs $u(0), u(1), ..., u(N-1)$ such that this input sequence will yield $x(N) = x_1$.*

Note that the choice of the initial condition $x(0) = 0$ will not lose generality, because for other initial condition we can always arrive at that state using finite steps. Given a linear system, we have linear algebra tool to examine its controllability, as we show below.

Theorem 36.2.1 (controllability criterion). *[2, p. 278]A discrete-time linear system is completed controllable if and ony if the $n \times nm$ controllability matrix*

$$M = [B, AB, ..., A^{n-1}B]$$

has rank n.

Proof. Suppose a sequence of inputs $u(0), u(1), ..., u(N-1)$ is applied to the system, with $x(0) = 0$. It follows

$$x(N) = A^{N-1}Bu(0) + A^{N-2}Bu(1) + \cdots + Bu(N-1).$$

From here, we can see points in the state space can be reached if and only if they can be expressed as linear combinations of columns of M. It can be showed that $N = n$ will suffice (see reference). □

Remark 36.2.1 (caution when u is constrained)**.** The above theorem assumes that admissible u is unconstrained. If u is constrained, the theorem will not apply.

36.3 Dynamic programming principle

36.3.1 Principle of optimality

Theorem 36.3.1 (principle of optimality for trajecotries). *[1, p. 54]Let $a-b-e$ be an optimal trajectory in the state space from a to e with associated cost $J_{abc}*$, then $b-e$ is the optimal path from b to e.*

Proof: Suppose there is another path $b-f-e$ with less cost than the cost of $b-e$, then the total cost for $a-b-e$ can be reduced, which is a contradiction.

36.3.2 The Hamilton-Jacobi-Bellman equation (finite horizon)

Although the goal of optimal control problem is to seek optimal control policy u that maximize process gain

$$\max_{u(x,t)} h(x_{t_f}, t_f) + \int_{t_0}^{t_f} g(x(t), u(x,t), t)dt,$$

it is usually intractable to directly solve for u. In the Hamilton-Jacobi-Bellman equation framework, we first derive the governing equation for value function $V(x(t),t)$, which is the maximum process gain if the system starts from $x(t)$ at time t. Then u can be solved via V, as we show in the following.

Theorem 36.3.2 (HJB for finite horizon process). *[1, p. 88] Let value function $V(x(t),t)$ be*

$$V(x(t),t) = \min_{u(\tau), t\le\tau\le t_f} [\int_t^{t_f} g(x(\tau), u(\tau), \tau)d\tau + h(x(t_f), t_f)].$$

Then the HJB equation is given as

$$0 = V_t + \min_{u(x,t)} [g(x(t), u(x,t), t) + V_x \dot{x}]$$

with boundary condition

$$V(x(t_f), t_f) = h(x(t_f), t_f).$$

With solved V, $u(x,t)$ is given by

$$u(x,t) = \arg\min_{u(x,t)} g(x(t), u(x,t), t) + V_x \dot{x}.$$

Proof. Let

$$V(x(t),t) = \min_{u(\tau), t\le\tau\le t_f} [\int_t^{t_f} g(x(\tau), u(\tau), \tau) d\tau + h(x(t_f), t_f)]$$

By subdividing the interval, we have

$$\begin{aligned} V(x(t),t) &= \min_{u(\tau), t\le\tau\le t_f} [\int_t^{t_f} g(x(\tau), u(\tau), \tau) d\tau + h(x(t_f), t_f)] \\ &= \min_{u(\tau), t\le\tau\le t_f} [\int_t^{t+dt} g(x(\tau), u(\tau), \tau) d\tau \\ &+ \int_{t+dt}^{t_f} g(x(\tau), u(\tau), \tau) d\tau + h(x(t_f), t_f)] \\ &= \min_{u(t)} [g(x(t), u(t), t) dt + V(x(t+dt), t+dt)] \\ &= \min_{u(t} [g(x(t), u(t), t) dt + V(x(t), t) + V_t dt + V_x \dot{x} dt] \end{aligned}$$

Then, we have the HJB equation

$$0 = V_t + \min_{u(t)} [g(x(t), u(t), t) + V_x \dot{x}]$$

with boundary condition

$$V(x(t_f), t_f) = h(x(t_f), t_f)$$

□

Remark 36.3.1. The function $V(x(t),t)$ is not a function of u since it is the already the minimum value.

36.3.3 The Hamilton-Jacobi-Bellman equation (infinite horizon)

In the infinite horizon setting, the value function $V(x(t),t)$ can be showed to be independent of t. Therefore, it can be written as $V(x(t))$.

Lemma 36.3.1 (time independence of value function). *Define the value function*

$$V(x(t),t) = \min_{u(\tau),t\leq\tau}[\int_t^\infty \exp(-\gamma(\tau-t))g(x(\tau),u(\tau),\tau)d\tau]$$

then V only depends on $x(t_0)$.

Proof.

$$\begin{aligned} V(x(t),t) &= \min_{u(\tau),t\leq\tau}[\int_t^\infty \exp(-\gamma(\tau-t))g(x(\tau),u(\tau),\tau)d\tau] \\ &= \min_{u(s),0\leq s}[\int_0^\infty \exp(-\gamma s)g(x(s+t),u(s+t),s+t)ds] \\ &= \min_{u(s),0\leq 0}[\int_0^\infty \exp(-\gamma s)g(x(s),u(s),s)ds = V(x(0),0) \end{aligned}$$

where we use variable substitution and the time invariance of g. □

Theorem 36.3.3 (HJB for infinite horizon process). *Let*

$$V(x(t),t) = \min_{u(\tau),t\leq\tau}[\int_t^\infty \exp(-\gamma(\tau-t))g(x(\tau),u(\tau),\tau)d\tau],$$

then HJB equation

$$0 = \min_{u(t)}[g(x(t),u(t),t) - \gamma V + V_x^T\dot{x}]$$

with boundary condition $V(x(t),t) = C, \forall x \in X$

Proof. Let

$$V(x(t),t) = \min_{u(\tau),t\leq\tau}[\int_t^\infty \exp(-\gamma(\tau-t))g(x(\tau),u(\tau),\tau)d\tau]$$

By subdividing the interval, we have

$$\begin{aligned}
V(x(t),t) &= \min_{u(\tau),t\leq\tau}[\int_t^{t_f} \exp(-\gamma(\tau-t)g(x(\tau),u(\tau),\tau)d\tau] \\
&= \min_{u(\tau),t\leq\tau}[\int_t^{t+dt} \exp(-\gamma dt)g(x(\tau),u(\tau),\tau)d\tau \\
&+ \exp(-\gamma dt)\int_{t+dt}^{\infty} \exp(-\gamma(\tau-t-dt)g(x(\tau),u(\tau),\tau)d\tau] \\
&= \min_{u(t)}[g(x(t),u(t),t)dt + \exp(-\gamma dt)V(x(t+dt),t+dt)] \\
&= \min_{u(t)}[g(x(t),u(t),t)dt + \exp(-\gamma dt)V(x(t),t) + V_x\dot{x}dt]
\end{aligned}$$

Then, we have the HJB equation

$$0 = \min_{u(t)}[g(x(t),u(t),t) - \gamma V + V_x\dot{x}]$$

with boundary condition $V(x(t),t) = C, \forall x \in X$ where we have used the time independence property of V, and $\exp(-\gamma dt) = 1 - \gamma dt$ □

Remark 36.3.2. If $\gamma = 0$, then there is no discount.

36.4 Deterministic linear quadratic control

36.4.1 Linear quadratic control (finite horizon)

Definition 36.4.1 (finite horizon linear quadratic control). *Consider the system state equation given as*

$$\dot{x}(t) = A(t)x(t) + B(t)u(t)$$

and we want to minimize

$$J = \frac{1}{2}x^T(t_f)Hx(t_f) + \frac{1}{2}\int_{t_0}^{t_f} x^T(t)Qx(t) + u^T(t)R(t)u(t)dt$$

where H and Q are real symmetric positive semi-definite matrices, R is a real symmetric positive definite matrix.

Remark 36.4.1. Note that matrix R has to be positive definite to eliminate the situation that $u(t)$ blows up in order to minimize J.

Theorem 36.4.1 (HJB equation for finite horizon linear quadratic control). *Define the value function*

$$V(x(t),t) = \min_{u(\tau),t\leq\tau\leq t_f}[\frac{1}{2}x^T(t_f)Hx(t_f) + \frac{1}{2}\int_{t_0}^{t_f} x^T(t)Qx(t) + u^T(t)R(t)u(t)dt]$$

The HJB equation is given as

$$0 = V_t + \frac{1}{2}x^TQx - \frac{1}{2}V_x^TBR^{-1}B^TV_x + V_x^TAx$$

with boundary condition $V(x(t_f),t_f) = \frac{1}{2}x^T(t_f)Hx(t_f)$.

Proof. Use Theorem 36.3.2, we have

$$0 = V_t + \min_{u(t)}[g(x(t),u(t),t) + V_x^T\dot{x}].$$

Note that $\dot{x} = Ax + Bu$, the minimize

$$\frac{1}{2}x^T(t)Qx(t) + \frac{1}{2}u^T(t)R(t)u(t) + V_x^T(Ax + Bu)$$

over u. The minimizer is given by $u^* = -R^{-1}B^TV_x$. Plug in u^* and we will get the result.

□

Remark 36.4.2 (solution to HJB). We can propose a solution with quadratic form $V(x(t),t) = \frac{1}{2}x^T H(t)x$ and solve the form of $H(t)$. Also see [1, p. 93] for details.

36.4.2 Linear quadratic control(infinite horizon)

Definition 36.4.2 (infinite horizon linear quadratic control). *Consider the system state equation given as*

$$\dot{x}(t) = Ax(t) + Bu(t)$$

and we want to minimize

$$J = \frac{1}{2}\int_{t_0}^{\infty} \exp(-\gamma t)[x^T(t)Qx(t) + u^T(t)R(t)u(t)]dt$$

where H and Q are real symmetric positive semi-definite matrices, R is a real symmetric positive definite matrix, and γ is the discount factor($\gamma = 0$ means no discount).

Remark 36.4.3. Note that R has to be positive definite to eliminate the situation that $u(t)$ blows up in order to minimize J.

Theorem 36.4.2. *The HJB equation for the infinite horizon linear quadratic control problem is given as*

$$\gamma V = \frac{1}{2}x^T Qx - \frac{1}{2}V_x^T BR^{-1}B^T V_x + V_x^T Ax$$

with boundary condition $V(x(t_0) = 0, t_0) = 0$.

Proof. Use Theorem 36.3.3, we have

$$\gamma V = \min_{u(t)}[g(x(t),u(t),t) + V_x^T \dot{x}].$$

Note that $\dot{x} = Ax + Bu$, the minimize

$$\frac{1}{2}x^T(t)Qx(t) + \frac{1}{2}u^T(t)R(t)u(t) + V_x^T(Ax + Bu)$$

over u. The minimizer is given by $u^* = -R^{-1}B^T V_x$. Plug in u^* and we will get the result. □

Remark 36.4.4 (solution methods).

- See [1, p. 213][3] for details on how to solve this nonlinear algebraic equations.

- For infinite horizon case will give a ordinary differential equation instead of a partial differential equation in finite horizon case.
- We can use finite difference method to solve this ODE.Note that in every interior node, we have a algebraic equation.

36.5 Continuous-time stochastic optimal control

36.5.1 HJB equation for general nonlinear systems

Definition 36.5.1 (general nonlinear system control). *[4, p. 421]*

- *We are given a continuous-time $n-$dimensional dynamic system*

$$\dot{x}(t) = f(x(t), u(t), t) + L(t)w(t), x(0) = x_0$$

where $L(t) \in \mathbb{R}^{n\times s}$, and random disturbance $w(t)$ satisfying

$$E[w(t)] = 0, E[w(t)w(\tau)^T] = W(t)\delta(t-\tau)$$

- *The goal is to minimize*

$$J = E[\phi(x(t_f), t_f) + \int_{t_0}^{t_f} \mathcal{L}(x(t), u(t), t)dt]$$

by choosing $u(t)$ as the control input. The ϕ is the terminal cost and $\mathcal{L}(x, u, t)$ is the instantaneous cost function.

Definition 36.5.2 (value function). *The value function $V(x, t)$ is defined over the state space and the time interval $[t, t_f]$,given as*

$$V(x(t), t) = \min_{u(t), t\in[t,t_f]} E[\int_t^{t_f} \mathcal{L}(x(\tau), u(\tau), \tau)d\tau]$$

Remark 36.5.1 (interpretation). The value function is a deterministic function and is the expected optimal cost for the system starting at $x(t)$ at time t.

Theorem 36.5.1 (Hamilton-Jacobi-Bellman (HJB) equation). *Under optimal control, the value function of the optimal trajectories must satisfy the following HJB equation given as:*

$$\partial_t V(x, t) = \min_{u(t)}\{\mathcal{L}(x, u, t) + \nabla_x V(x, t)^T f(x, u) + \frac{1}{2}Tr[\nabla_x^2 V(x, t)L(t)W(t)L(t)^T]\}$$

Proof.

$$\begin{aligned}
&V(x+\Delta x, t+\Delta t)\\
=&V(x,t)+\partial_t V(x,t)\Delta t+\nabla_x V(x,t)^T\Delta x+\frac{1}{2}\Delta x^T\nabla_x^2 V(x,t)\Delta x+o(\Delta t)\\
=&V+\partial_t V\Delta t+\nabla_x V^T(f+Lw)\Delta t+(f+Lw)^T\nabla_x^2 V(x,t)(f+Lw)(\Delta t)^2+o(\Delta t)\\
=&V+\partial_t V\Delta t+\nabla_x V^T(f+Lw)\Delta t+(f+Lw)^T\nabla_x^2 V(x,t)(f+Lw)(\Delta t)^2+o(\Delta t)
\end{aligned}$$

where we use $\Delta x=(f+Lw)\Delta t$, the trace of a scalar is the scalar itself and the cyclic rule of matrix trace [Lemma A.8.8]. □

36.5.2 Linear Gaussian quadratic system

Definition 36.5.3 (linear Gaussian quadratic control). *[4, p. 421]*

- *We are given a continuous-time $n-$dimensional dynamic system*

$$\dot{x}(t)=Fx+Gu+Lw$$

where $L(t)\in\mathbb{R}^{n\times s}$, and random disturbance $w(t)$ satisfying

$$E[w(t)]=0, E[w(t)w(\tau)^T]=W(t)\delta(t-\tau)$$

- *The goal is to minimize*

$$J=\frac{1}{2}E[x^T(t_f)S_f x(t_f)+\int_{t_0}^{t_f}[x(t)^T\ u(t)^T]\begin{bmatrix}Q(t) & M(t)\\ M(t)^T & R(t)\end{bmatrix}\begin{bmatrix}x(t)\\ u(t)\end{bmatrix}$$

by choosing $u(t)$ as the control input. The $R(t), Q(t)$ are symmetric matrices and $R(t)$ is required to be positive definite.

Theorem 36.5.2 (Hamilton-Jacobi-Bellman (HJB) equation). *Under optimal control, the value function of the optimal trajectories must satisfy the following HJB equation given as:*

$$\partial_t V(x,t)=-\min_{u(t)}\frac{1}{2}\{x^TQx+2x^TMu+u^TRu+x^TS(Fx+Gu)+Tr(SLWL^T)\}$$

Proof. (use Theorem 36.5.1]. □

36.6 Stochastic dynamic programming

36.6.1 Discrete-time Stochastic dynamic programming: finite horizon

Definition 36.6.1 (basic problem of finite horizon). *[5, p. 12]*

- *We are given a discrete-time dynamic system*

$$x_{k+1} = f_k(x_k, u_k, w_k)$$

  where the state x_k is an element of a space S_k, the control u_k is an element in the control space C_k, and random disturbance w_k is an element of a space D_k.
- *A control policy π is consisting of a sequence of functions*

$$\pi = \{\mu_0, \mu_1, ..., \mu_N\}$$

  where $\mu_k : S_k \to C_k$ is a function maps states x_k to $u_k = \mu_k(x_k)$.
- *For given reward function $g_k, k = 0, 1, ..., N$, the expected cost of π starting at x_0 is*

$$J_\pi(x_0) = E[g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)]$$

  where the expectation is taken over the joint distribution of all w_k and x_k.
- *The goal is to find an optimal control policy π^* such that*

$$J_{\pi^*}(x_0) = \min_\pi J_\pi(x_0)$$

Theorem 36.6.1 (Principle of Optimality). *[5, p. 18] Let $\pi^* = \{\mu_0^*, \mu_1^*, ..., \mu_N^*\}$ be a optimal policy for the basic problem, and assume that when using π^*, a given state x_i has positive probability. Then the truncated policy $\{\mu_i^*, \mu_{i+1}^*, ..., \mu_N^*\}$ is optimal for the subproblem starting at x_i*

$$E[g_N(x_N) + \sum_{k=i}^{N-1} g_k(x_k, \mu_k(x_k), w_k)]$$

Lemma 36.6.1 (dynamic programming algorithm for basic problem of finite horizon). *The optimal cost function J^* and its associated optimal control policy $\pi^* = \{\mu_0^*, \mu_1^*, ..., \mu_{N-1}^*\}$ can be calculated using the following backward induction procedures:*

$$J_N^*(x_N) = g_N(x_N)$$

$$J_k^*(x_k) = \min_{\mu_k(x_k)} E_{w_k}[g_k(x_k, u_k, w_k) + J_{k+1}^*(f_k(x_k, \mu_k(x_k), w_k))], k = 0, 1, ..., N-1$$

Proof. Directly from principle of optimality. □

Remark 36.6.1 (interpretation). The lemma provides a way to calculate the optimal control policy.

Lemma 36.6.2 (monotonicity property of dynamic programming I). *If we change the final cost g_N to an uniformly larger cost function g_N'(i.e. $g_N'(x) \geq g_N(x), \forall x$), then all optimal cost function J_k^* will be uniformly increasing(at least not decreasing).*

Similar situation holds when g_N is changed to an uniformly smaller one.

Proof. Obviously $J_N^{*'} = g_N'$ will uniformly increase. For other k with induction,

$$J_k^{*'} = \min E[g_k + J_{k+1}^{*'}] \geq \min E[g_k + J_{k+1}^*] = J_k^*$$

□

Lemma 36.6.3 (monotonicity property of dynamic programming II). *[5, p. 60] Consider the basic problem with all functions and sets being time-invariant($S_k = S, g_k = g, f_k = f, ...$). If in the dynamic programming algorithm we have*

$$J_{N-1}^*(x) \leq J_N^*(x), \forall x \in S$$

then

$$J_k^*(x) \leq J_{k+1}^*(x), \forall x \in S, \forall k.$$

Similarly, if

$$J_{N-1}^*(x) \geq J_N^*(x), \forall x \in S$$

then

$$J_k^*(x) \geq J_{k+1}^*(x), \forall x \in S, \forall k.$$

36.6.2 Discrete-time stochastic dynamic programming: infinite horizon

36.6.2.1 *Fundamentals*

Definition 36.6.2 (basic problem of infinite horizon). *[6, p. 3]*

- *We are given a **stationary** discrete-time dynamic system*

$$x_{k+1} = f(x_k, u_k, w_k)$$

  where the state x_k is an element of a space S, the control u is an element in the control space C, and random disturbance w_k is an element of a space D.
- *A **stationary** control policy π is consisting of a sequence of functions*

$$\pi = \{\mu, \mu, ...\}$$

  where $\mu : S \to C$ is a function maps states x_k to $u_k = \mu(x_k)$.
- *For a given cost function $g, k = 0, 1, ..., N$, the expected cost of π starting at x_0 is*

$$J_\pi(x_0) = \lim_{N\to\infty} E_{w_k, k=1,...,N}[\sum_{k=0}^{N} \alpha^k g(x_k, \mu(x_k), w_k)]$$

  where $\alpha \in [0, 1)$ is the discount factor, the expectation is taken over the joint distribution of all w_k and x_k.
- *The goal is to find an optimal control policy π^* such that*

$$J_{\pi^*}(x_0) = \min_\pi J_\pi(x_0)$$

Remark 36.6.2 (what stationarity means?).

- Compared to finite horizontal problem, infinite horizon problem requires the dynamical system to be time invariant.
- If $f(x_k, u_k, w_k)$ is state dependent but not time dependent, then the dynamic system is still time-invariant. For example, we can have $f(x_k, u_k, w_k) = A(x_k)x_k + B(x_k)u_k + L(x_k)w_k$, or write as $f(x, u, w) = A(x)x + B(x)u + L(x)w$

Definition 36.6.3 (dynamic programming operator).

- $(TJ)(x) = \min_{u\in U(x)} E[g(x, u, w) + \alpha J(f(x, u, w))]$
- $(T_\mu J)(x) = E[g(x, u, w) + \alpha J(f(x, \mu(x), w))]$

36.6.2.2 *Convergence analysis*

Lemma 36.6.4 (Monotonicity lemma). *[6, p. 9] For any functions $J, J' : X \to \mathbb{R}$ such that for all $x \in X$,*

$$J(x) \leq J'(x)$$

and any stationary policy $\mu : X \to U$, we have

$$(T^k J)(x) \leq (T^k J')(x)$$

and

$$(T^k J)(x) \leq (T^k J')(x)$$

for all $x \in X$ and all $k = 1, 2, ...$

Proof. For $k = 1$, we can show its correctness. For other k use induction. □

Lemma 36.6.5 (constant shift lemma). *[6, p. 9] For every k, function $J : X \to \mathbb{R}$, stationary policy μ, scalar $r \in \mathbb{R}$, and $x \in X$, we have*

$$(T^k(J + r))(x) = (T^k J)(x) + \alpha^k r$$
$$(T_\mu^k(J + r))(x) = (T_\mu^k J)(x) + \alpha^k r$$

Proof. For $k = 1$, we can show that

$$(T(J + r))(x) = (T^k J)(x) + \alpha r$$
$$(T_\mu(J + r))(x) = (T_\mu^k J)(x) + \alpha r$$

Then we can use induction for other k. □

Theorem 36.6.2 (dynamic programming operator as a contraction mapping). *[5, p. 18] The following two operators defined as the space of bounded functions of $J : X \to \mathbb{R}$*

- $(TJ)(x) = \min_{u \in U(x)} E[g(x, u, w) + \alpha J(f(x, u, w))]$
- $(T_\mu J)(x) = E[g(x, u, w) + \alpha J(f(x, \mu(x), w))]$

are contracting mappings with respect to the sup-norm/max-norm. Note that the expectation is taken respect to distribution of w.

Proof. Denote

$$c = \max_{x \in X} \left| J(x) - J'(x) \right|,$$

so that for all $x \in X$, we have

$$J(x) - c \leq J'(x) \leq J(x) + c$$

Apply T and use Monotonicity and constant shift lemma, we have

$$TJ - \alpha c \leq TJ' \leq J + \alpha c, \forall x \in X$$

Therefore

$$\left|TJ - TJ'\right| \leq \alpha c$$

and

$$\max\left|TJ - TJ'\right| \leq \alpha \max\left|J - J'\right|$$

□

Corollary 36.6.2.1 (convergence rate). *[6, p. 18] For any two bounded functions $J, J' : X \to \mathbb{R}$, we have*

$$\max_{x \in X}\left|(T^k J)(x) - (T^k J')(x)\right| \leq \alpha^k \max_{x \in X}\left|(J)(x) - (J')(x)\right|$$

Corollary 36.6.2.2 (convergence rate). *[6, p. 18] For any two bounded functions $J, J' : X \to \mathbb{R}$ and any stationary policy μ, we have*

$$\max_{x \in X}\left|(T_\mu^k J)(x) - (T_\mu^k J')(x)\right| \leq \alpha^k \max_{x \in X}\left|(J)(x) - (J')(x)\right|$$

Remark 36.6.3 (interpretation of convergence).

- Any initial J is guaranteed to converge.
- The convergence rate depends on the initial distance between J and J^*, and the discount factor. In the extreme case of $\alpha = 0$, convergence is one single step.

36.7 Notes on bibliography

For introductory treatment on classical control theory, see [2][1]. For application of optimal control theory in finance, see [7][8][9][5]. For advanced treatment on this topic, see [10]. For an introduction to calculus of variations, see [1]. For treatment of linear state space control, see [11]. For certainty equivalence, see [5, p. 160]. For dynamic programming theory, see [12]. For reinforcement learning, see [13].

BIBLIOGRAPHY

1. Kirk, D. E. *Optimal control theory: an introduction* (Courier Corporation, 2012).
2. Luenberger, D. *Introduction to dynamic systems: theory, models, and applications* (Wiley, 1979).
3. Wikipedia. *Algebraic Riccati equation — Wikipedia, The Free Encyclopedia* [Online; accessed 1-August-2016]. 2016.
4. Stengel, R. F. *Optimal control and estimation* (Courier Corporation, 2012).
5. Bertsekas, D. *Dynamic Programming and Optimal Control* ISBN: 9781886529083 (Athena Scientific, 2012).
6. Bertsekas, D. *Dynamic Programming and Optimal Control Athena Scientific optimization and computation series* **v. 2.** ISBN: 9781886529441 (Athena Scientific, 2012).
7. Miranda, M. J. & Fackler, P. L. *Applied computational economics and finance* (MIT press, 2004).
8. Chang, F.-R. *Stochastic optimization in continuous time* (Cambridge University Press, 2004).
9. Pham, H. *Continuous-time stochastic control and optimization with financial applications* (Springer Science & Business Media, 2009).
10. Fleming, W. H. & Soner, H. M. *Controlled Markov processes and viscosity solutions* (Springer Science & Business Media, 2006).
11. Williams, R. L., Lawrence, D. A., *et al. Linear state-space control systems* (John Wiley & Sons, 2007).
12. Bertsekas, D. P. *Abstract dynamic programming* (Athena Scientific, 2018).
13. Wiering, M. & Van Otterlo, M. Reinforcement learning. *Adaptation, Learning, and Optimization* **12** (2012).

37

REINFORCEMENT LEARNING

37.1 Reinforcement learning framework

37.1.1 Notations

S_t, s_t	state at time t; S_t denotes random variable; s_t denotes one realization.
A_t, a_t	action at time t; A_t denotes random variable; a_t denotes one realization.
R_t	reward at time t.
γ	discount rate ($0 \leq \gamma \leq 1$).
G_t	discounted return at time t ($\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$).
$\mathcal{S}$	set of all states, also known as state space.
$\mathcal{S}^T$	set of all terminal states.
$\mathcal{A}$	set of all actions, also known as action space .
$\mathcal{A}(s)$	set of all actions available in state s.
$p(s'\|s,a)$	transition probability to reach next state s', given current state s and current action a.
π, μ	policy. *if deterministic*: $\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$. *if stochastic*: $\pi(a\|s) = \mathbb{P}(A_t = a\|S_t = s)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.
V^π	state-value function for policy π ($v_\pi(s) \doteq E[G_t\|S_t = s]$ for all $s \in \mathcal{S}$).
Q^π	action-value function for policy π ($q_\pi(s,a) \doteq E[G_t\|S_t = s, A_t = a]$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$).
V^*	optimal state-value function ($v_*(s) \doteq \max_\pi V^\pi(s)$ for all $s \in \mathcal{S}$).
Q^*	optimal action-value function ($q_*(s,a) \doteq \max_\pi Q^\pi(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$).

37.1.2 Overview

In essence, we can view **reinforcement learning (RL)** as a data-driven way to solve the policy optimization problem in MDP. In RL, we use a setting that contains an agent and an environment [Figure 37.1.1], where data are collected through agent-environment interactions and then further utilized to optimize policies. The agent iteratively takes action specified by a policy, interacts with the environment, and ultimately learn 'knowledge or strategies' from the interactions. At every step, the agent makes an observation of the environment (we called a state or an observation), then it chooses an action to take. The environment will respond to the agent's action by changing its state and providing a reward signal to the agent, which serves as a gauge on the quality of the agent's current action. The goal of the agent, however, is not to maximize the immediate reward of an action, but is to maximize its cumulative reward along the whole process.

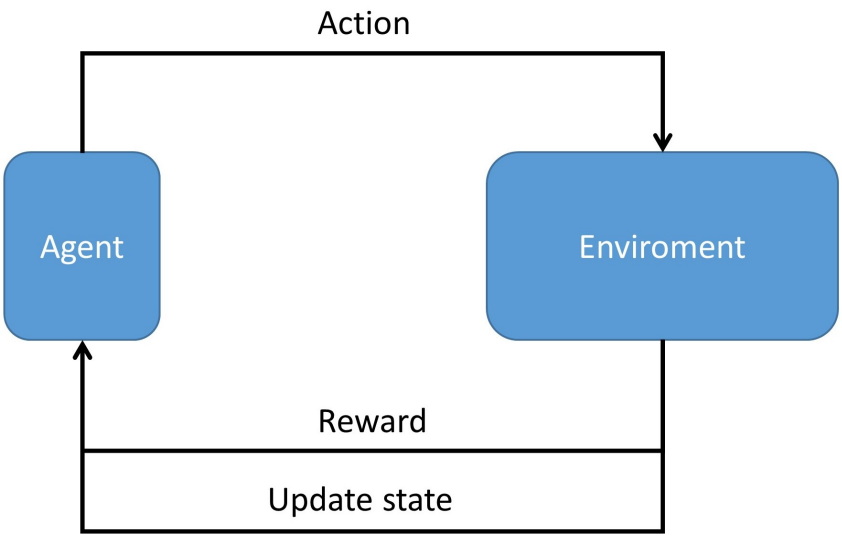


Figure 37.1.1: One core component in reinforcement learning is agent environment interaction. The agent takes actions based on observations on the environment and a decision-making module that maps observations to action. The environment model updates system state and provides rewards according to the action

For example, in the Atari game *Breakout* [Figure 37.1.2], we assume an agent controls the bat to deflect the ball to destroy the bricks. The actions allowed are moving left and moving right; the state includes positions of the ball, the bat, and all the bricks; rewards will be given to the agent if bricks are hit. The environment, represented by a physical simulator, will simulate the ball's trajectory and collision between the ball and the bricks. The ultimate goal is learn a control policy that specifies the action to take after observing a state.

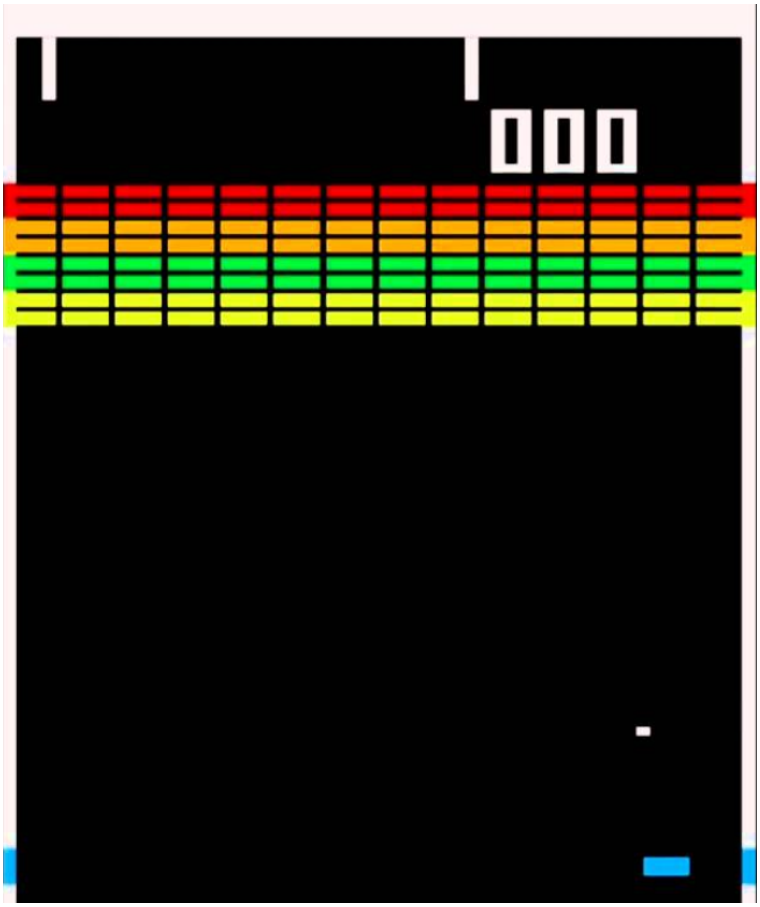


Figure 37.1.2: Scheme of the Atari game *breakout*.

In finite state Markov decision process, we introduce the two step iterative framework consisting of policy evaluation and policy improvement to seek optimal control policies. Although the agent-environment interaction paradigm seems to be vastly different from the Markov decision framework, many reinforcement learning algorithms can also be interpreted under this two step framework. We can view the agent-environment interaction process under a specified policy as policy evaluation step based on the rewards received from the environment. With the estimated value functions, we can similarly apply the policy improvement methods [Figure 37.1.3].

The sharp contrast between MDP and reinforcement learning is that reinforcement learning is model-free learning, the knowledge of the environment is from agent-environment interaction data. On the other hand, MDP is model-based learning, meaning that the agent already know beforehand all the responses and rewards from environment for every action it takes. For complex read-world problems where models are not available, reinforcement learning offers a viable approach to learning optimal control policy via gradually building up the knowledge of the environment.

How much knowledge of the environment should the agent gain before the agent starts to improve the policy? Exploring the environment sufficiently would be prohibitive; on the other hand, improving the policy based on limited knowledge can produce inferior policies. The balance of environment exploration and policy improvement is known as the **exploration-exploitation dilemma**. One common way out is the ϵ greedy action selection, where the agent has $(1-\epsilon)$ probability to continue to collect more samples based on currently perceived optimal control policy to improve current policy and ϵ probability to randomly explore the environment via random actions.

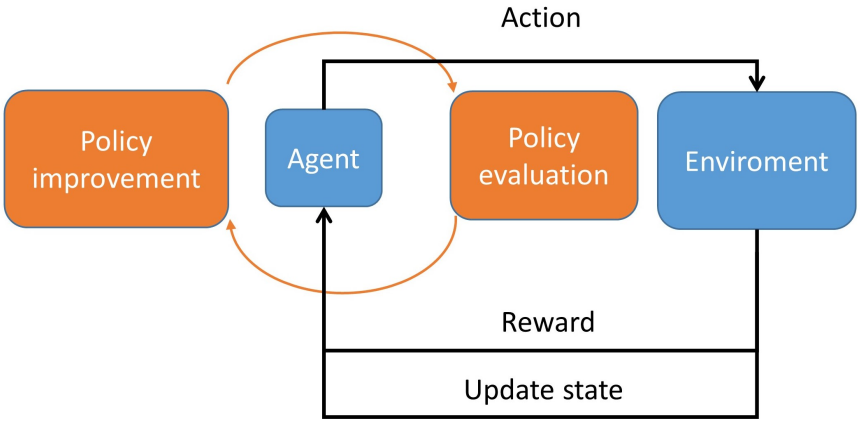


Figure 37.1.3: Policy evaluation and policy improvement framework in the context reinforcement learning.

37.1.3 Finite state Markov decision process

Definition 37.1.1 (finite state Markov decision process). *A **finite state Markov decision process (MDP)** is characterized by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P})$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, and $\mathcal{P} = \{p(s'|s,a), s, s' \in \mathcal{S}, a \in \mathcal{A}\}$ is the state transition probability. We require $\mathcal{S}, \mathcal{A}$ to have finite number of elements. The goal is compute an optimal control policy $\pi^* : \mathcal{S} \to \mathcal{A}$ such that the expected total reward in the process*

$$J = E[\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, a_t)]$$

is maximized, where $R(s,a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the one-step reward function and $\gamma \in [0,1)$ is the discount factor.[a]

[a] Here we use $R(s_{t+1}, a_t)$ to emphasize that the reward is received at state s_{t+1} after taking action a_t at state s_t.

Example 37.1.1 (examples of reward functions).

- In a navigation task, we can set $r(s_t, a_t) = \mathbf{1}(s_t \in S_{target})$.
- In a game, the state of gaining scores has a reward 1 and other states have a reward 0.

Definition 37.1.2 (value functions).

- *Let π be a given control policy. We can define a **value function** associated with this policy by*
$$V^\pi(s) = E^\pi[\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, a_t)|s_0 = s, \pi],$$
which is the expected total rewards by following a given policy π starting from initial state s.
- *Given a value function associated with a policy π, we can obtain π via*
$$\pi(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} p(s'|s, a)(r + \gamma V(s'))$$
where $r = R(s', a)$ is the reward received at state s' after taking action a at s.
- *The **optimal value function** V^* and the optimal policy π^* are connected via*
$$V^*(s) = \max_\pi V^\pi(s), \pi^*(s) = \max_\pi V^\pi(s).$$

Lemma 37.1.1 (recursive relationship of value functions). *Given a value function V^π associated with a control policy π. The value function satisfies the following backward relationship:*
$$V^\pi(s) = E_{s' \sim P(s'|s, a=\pi(s))}[R(s', a) + \gamma V^\pi(s')].$$
Particularly, we have the Bellman's equation characterizing the optimal value function by
$$V^*(s) = \max_a E_{s' \sim P(s'|s,a))}[R(s', a) + \gamma V^*(s')].$$

Proof. The definition of V^π says

$$\begin{aligned} V^\pi(s) &= E^\pi[\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, a_t)|s_0 = s, \pi] \\ &= E_{s_1 \sim P(s_1|s_0, a=\pi(s))}[R(s_1, a_0) + E^\pi[\sum_{t=1}^{\infty} \gamma^t R(s_{t+1}, a_t)|s_1 = s', \pi]|s_0 = s, \pi] \\ &= E_{s_1 \sim P(s_1|s_0, a=\pi(s))}[R(s_1, a_0) + V(s_1)|s_0 = s, \pi] \end{aligned}$$

where we have used the tower property of conditional expectation. □

37.1.4 State-action Value function (Q function)

Like value function in a finite state MDP, Q functions play the same critical role in reinforcement learning. Now we go through their formal definition and their recursive relations.

Definition 37.1.3 (state-action value function). *[1, p. 16]*

- *The **state-action value function** $Q^\pi : S \times A \to \mathbb{R}$ associated with a policy π is defined as the expected return starting from state s, taking action a and thereafter following the policy π, given as*

$$Q^\pi(s,a) = E^\pi\{\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, a_t) | s_0 = s, a_t = a\}.$$

- *The optimal state-action value function $Q^* : S \times A \to \mathbb{R}$ is defined as the expected return starting from state s, taking action a and thereafter following an optimal policy π^*, such that*

$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

- *The optimal policy π^* is related to Q^* as*

$$\pi^*(s) = \arg\max_a Q^*(s,a).$$

- *The value function associated with a policy π,*

$$V^\pi(s) = E^\pi\{\sum_{t=0}^{\infty} \gamma^t R(s_{t+1}, a_t) | s_0 = s\}.$$

- *The value function $V(s)$ is connected with $Q(s,a)$ via*

$$V^\pi(s) = Q(s, \pi(s)).$$

- *The optimal state-action value function is connected to value function via*

$$V^*(s) = \max_a Q^*(s,a).$$

Lemma 37.1.2 (recursive relations of the Q function).

- *The state-action value function will satisfy*

$$\begin{aligned} Q^{\pi}(s,a) &= E^{\pi}_{s'\sim p(s'|s,\pi(s))}[R(s',a)+\gamma Q^{\pi}(s',\pi(s'))] \\ &= E^{\pi}_{s'\sim p(s'|s,\pi(s))}[R(s',a)+\gamma V^{\pi}(s')], \end{aligned}$$

  where the expectation is taken with respect to the distribution of s'(the state after taking a at s) and we use the definition $V^{\pi}(s') = Q^{\pi}(s',\pi(s'))$.
- *Particularly, the optimal state-action value function will satisfy*

$$\begin{aligned} Q^{*}(s,a) &= E^{\pi^*}_{s'\sim p(s'|s,\pi^*(s))}[R(s',a)+\gamma \max_{a\in A(s')} Q*(s',a)] \\ &= E^{\pi^*}_{s'\sim p(s'|s,\pi^*(s))}[R(s',a)+\gamma V^{*}(s')] \end{aligned}$$

  where the expectation is taken with respect to the distribution of s'(the state after taking a at s) we use the definition $V^{*}(s') = \max_a Q^{*}(s',a) = Q^{*}(s',\pi^{*}(s'))$.

Proof. (1) From the definition $Q^{\pi}(s,a)$, we have

$$\begin{aligned} Q^{\pi}(s,a) &= E^{\pi}[\sum_{t=0}^{\infty}\gamma^t r_{t+1}|s_0=s,a_t=a] \\ &= E^{\pi}[r_1+\sum_{k=1}^{\infty}\gamma^k r_{t+1}|s_0=s,a_t=a] \\ &= E^{\pi}[r_1+E^{\pi}[\sum_{k=1}^{\infty}\gamma^k r_{t+1}|s_1=s,a_1=\pi(s_1)]|s_0=s,a_t=a] \\ &= E^{\pi}[r_1|s_0=s,a_t=a\}+Q^{\pi}(s_1,\pi(s_1))|s_0=s,a_t=a] \end{aligned}$$

where we have used the tower property of conditional expectation. (2) From (1) we have

$$Q^{*}(s,a) = E^{*}_{s'\sim p(s'|s,\pi(s))}[r+\gamma Q^{*}(s',\pi^{*}(s'))].$$

Further note that $\pi^{*}(s') = \arg\max_{a\in\mathcal{A}(s')} Q(s',a')$ □

Remark 37.1.1 (recursive relation for value functions). Recall that in Lemma 37.1.1, we have covered the recursive relation for value functions.

- Given a value function V^{π} associated with control policy π. The value function satisfies the following backward relationship:

$$V^{\pi}(s) = E_{s'\sim P(s'|s,a=\pi(s))}[R(s',a)+\gamma V^{\pi}(s')].$$

- Particularly, we have the Bellman's equation saying that

$$V^{*}(s) = \max_a E_{s'\sim P(s'|s,a))}[R(s',a)+\gamma V^{*}(s')].$$

Note that in above two cases, a is determined by either the policy function π or the maximization. a is not a free variable.

37.1.5 Policy iteration and value iteration

37.1.5.1 *Policy iteration*

The core idaa underlying policy iteration is to iteratively carry out two procedures: **policy evaluation** and **policy improvement** [Figure 37.1.4]. Given a starting policy π, we perform policy evaluation to estimate the value function V^π associated with this policy; then we improve the policy via dynamic programming principles, or the Bellman Principle.

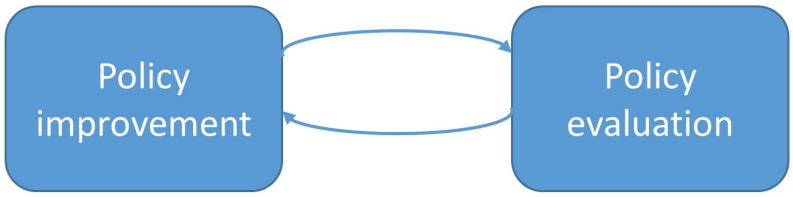


Figure 37.1.4: Policy iteration involves iteratively carrying out policy evaluation and policy improvement procedures.

Methodology 37.1.1 (iterative policy evaluation procedure). *Given a policy π, we can evaluate the policy via calculating V^π using the following iterative procedure:*

$$V^{k+1}(s) = \sum_{s' \in \mathcal{S}} p(s'|s, a = \pi(s))(r + \gamma V^k(s')), \forall s \in \mathcal{S}$$

where superscript k is the iteration index.

$V^k(s)$ will converge to value function V^π [Theorem 37.1.1].

Theorem 37.1.1 (convergence property of iterative policy evaluation). *For a finite state MDP, we can write the value function recursive relationship explicitly as*

$$V^\pi(s) = \sum_{s' \in \mathcal{S}} P(s'|s, a = \pi(s))[R(s', a) + \gamma V^\pi(s')].$$

We can express the recursive relationship as a matrix form given by

$$V = T(R + \gamma V),$$

where $R, V \in \mathbb{R}^{|\mathcal{S}|}, T \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$.

We further define $H(V) \triangleq T(R + \gamma V)$ *as the policy evaluation operator.*

We have

- *H is a contraction mapping.*
- *In iterative policy evaluation,* $V^k(s)$ *will converge to value function* V^π*. Or equivalently,* V^π *is the fixed point of H, and*

$$\lim_{n \to \infty} H^n(V) = V^\pi.$$

- *(error bound) If* $\left\| H^k(V) - H^{k-1}(V) \right\|_\infty \leq \epsilon$*, then*

$$\left\| H^k(V) - V^\pi \right\|_\infty \leq \frac{\epsilon}{1 - \gamma}$$

Proof. (1)

$$\begin{aligned}
&\|H(\tilde{V}) - H(V)\|_\infty \\
&= \|TR + \gamma T\tilde{V} - TR - \gamma TV||_\infty \text{ (by definition)} \\
&= ||\gamma T(\tilde{V} - V)||_\infty \quad \text{(simplification)} \\
&\leq \gamma ||T||_\infty ||\tilde{V} - V||_\infty \quad \left(\text{since } \|AB\| \leq \|A\|\|B\|\right) \\
&= \gamma ||\tilde{V} - V||_\infty \quad \left(\text{since } \max_s \textstyle\sum_{s'} T\left(s, s'\right) = 1\right)
\end{aligned}$$

(2) Note that from Fixed oint Theorem [Theorem 6.2.3], we have

$$\left\| H^n(V) - V^\pi \right\|_\infty \leq \gamma^n \left\| V - V^\pi \right\|_\infty.$$

Therefore,

$$\lim_{n \to \infty} H^n(V) = V^\pi.$$

(3)

$$\begin{aligned}
&\left\|H^k(V) - V^\pi\right\|_\infty \\
=&\left\|H^k(V) - H^\infty(V)\right\|_\infty \\
=&\left\|\sum_{t=1}^{\infty} H^{t+k}(V) - H^{t+k+1}(V)\right\|_\infty \\
\leq&\sum_{t=1}^{\infty}\left\|H^{t+k}(V) - H^{t+k+1}(V)\right\|_\infty \\
\leq&\sum_{t=1}^{\infty}\gamma^t\left\|H^k(V) - H^{k+1}(V)\right\|_\infty \\
\leq&\sum_{t=1}^{\infty}\gamma^t\epsilon
\end{aligned}$$

□

Remark 37.1.2 (error estimation and stopping criterion). The third property can be used as a stopping criterion during iterations. Suppose the tolerance is Tol, then we should iterate until the maximum change during consecutive iteration is small than $(1-\gamma)\times Tol$.

Given a learned value function V^π of a policy π, we can derive its Q function counterpart via

$$Q(s,a) = \sum_{s'\in\mathcal{S}} p(s'|s,a)(R(s',a) + \gamma V^\pi(s')), \forall s, a.$$

Q function offer a convenient way to improve current policy π. Indeed, by relying on following **policy improvement theorem**, we can consistently improve our policy towards the optimal one.

Theorem 37.1.2 (policy improvement theorem). *[2, p. 78] Define the Q function associated with a policy π as*

$$Q^\pi(s,a) = \sum_{s'\in\mathcal{S}} p(s'|s,a)(R(s',a) + \gamma V^\pi(s')), \forall s, a.$$

Let π and π' be two policies. If $\forall s \in \mathcal{S}$,

$$Q^\pi(s,\pi'(s)) \geq V^\pi(s),$$

then

$$V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathcal{S}.$$

That is π' is a better policy than π.[a]

[a] Note that $Q^{\pi}(s, \pi'(s))$ is the expected process reward for a process where the first step takes action $\pi'(s)$ and thereafter uses $\pi(s)$.

Proof.

$$\begin{aligned}
v_{\pi}(s) &\leq Q^{\pi}\left(s, \pi'(s)\right) \\
&= E\left[R_{t+1} + \gamma V^{\pi}\left(S_{t+1}\right) | S_t = s, A_t = \pi'(s)\right] \\
&= E^{\pi'}\left[R_{t+1} + \gamma v_{\pi}\left(S_{t+1}\right) | S_t = s\right] \\
&\leq E^{\pi'}\left[R_{t+1} + \gamma Q^{\pi}\left(S_{t+1}, \pi'\left(S_{t+1}\right)\right) | S_t = s\right] \\
&= E^{\pi'}\left[R_{t+1} + \gamma E^{\pi}\left[R_{t+2} + \gamma V^{\pi}\left(s_{t+2}\right) | S_{t+1}, A_{t+1} = \pi'\left(S_{t+1}\right)\right] | s_t = s\right] \\
&= E^{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 V^{\pi}\left(S_{t+2}\right) | S_t = s\right] \\
&\leq E^{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 r_{t+3} + \gamma^3 v_{\pi}\left(S_{t+3}\right) | S_t = s\right] \\
&\vdots \\
&\leq E^{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots | S_t = s\right] \\
&= V^{\pi'}(s)
\end{aligned}$$

□

Based on this algorithm, we can create a better policy via following greedy manner

$$\pi'(s) = \arg\max_{a \in \mathcal{A}(s)} Q(s, a), \forall s.$$

Methodology 37.1.2 (policy improvement procedure). *Given a value function V, we can improve the policy implicitly associated with the value function via two steps:*

- *First calculate the intermediate Q function*

$$Q(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a)(r + \gamma V(s')), \forall s, a.$$

- *Second improve the policy by*

$$\pi'(s) = \arg\max_{a \in \mathcal{A}(s)} Q(s, a), \forall s.$$

Notably, let π' be the improved greedy policy, if $V^{\pi'} = V^{\pi}$, then π is the optimal policy, based on the definition and recursive relation of V [Lemma 37.1.1].

The following algorithm [algorithm 65] summarizes the policy iteration method[2, p. 80].

Algorithm 65: The policy iteration algorithm for MDP

Input: MDP model, small positive number θ
Output: policy $\pi \approx \pi_*$
Initialize π arbitrarily (e.g., $\pi(a|s) = \frac{1}{|\mathcal{A}(s)|}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$)
policy-stable $\leftarrow$ *false*
repeat
  $V \leftarrow$ **Policy_Evaluation**(MDP, π, θ)
  $\pi' \leftarrow$ **Policy_Improvement**(MDP, V)
  if $\pi = \pi'$ **then**
    policy-stable $\leftarrow$ *true*
  end
  $\pi \leftarrow \pi'$
until *policy-stable* $=$ *true*;
return π

37.1.5.2 *Value iteration*

The policy iteration method iterates the two steps of evaluating policy and improving policy. On the other hand, we can directly iteratively estimate the optimal value function, the value function associated the optimal policy, without evaluating the policy associated with value functions. In fact, since policies can be directly calculated from a given value function, having the optimal value function will just give us the optimal policy.

Let $V^{(0)}$ be an initial value function. Based on the value iteration theorem [Theorem 37.1.3], we can design iteration like

$$V^{(k+1)}(s) = \max_a \sum_{s' \in \mathcal{S}} P(s'|s,a)[R(s',a) + \gamma V^{(k)}(s')], \forall s \in \mathcal{S}.$$

where k is the iteration number. The following theorem shows that this iteration procedure will lead to convergence to the optimal value function.

More formally, we have

Theorem 37.1.3 (convergence property of value iteration). *For a finite state MDP, we can write the optimal value function recursive relationship as*

$$V^*(s) = \max_a \sum_{s' \in \mathcal{S}} P(s'|s,a)[R(s',a) + \gamma V^*(s')], \forall s \in \mathcal{S}.$$

We can express the recursive relationship as a matrix form given by

$$V = T(R + \gamma V),$$

where $R, V \in \mathbb{R}^{|\mathcal{S}|}, T \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$.

We further define $H(V) \triangleq \max_a T(R + \gamma V)$ *as the value iteration operator.*

We have

- *H is a **contraction mapping**.*
- *In iterative policy evaluation, $V^k(s)$ will converge to the unique optimal value function V^*. Or equivalently, V^* is the **fixed point** of the contraction mapping H, and*

$$\lim_{n \to \infty} H^n(V) = V^*.$$

- *(error bound) If $\left\| H^k(V) - H^{k-1}(V) \right\|_\infty \leq \epsilon$, then*

$$\left\| H^k(V) - V^* \right\|_\infty \leq \frac{\epsilon}{1-\gamma}.$$

Proof. (1) Without loss of generality, for each s, we assume $H(V')(s) \geq H(V)(s)$ and let

$$a_s^* = \arg\max_a \sum_{s' \in \mathcal{S}} P(s'|s,a)[R(s',a) + \gamma V^\pi(s')].$$

Then

$$\begin{aligned}
0 \leq & H(V'(s) - H(V)(s) \\
\leq & \sum_{s' \in \mathcal{S}} P(s'|s,a)(R(s',a) + \gamma V'(s') - R(s',a) - \gamma V(s')) \\
\leq & \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)(V'(s') - V(s')) \\
\leq & \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \left\| V'(s') - V(s') \right\|_\infty \\
= & \gamma \left\| V'(s') - V(s') \right\|_\infty .
\end{aligned}$$

(2) Because H is a contraction mapping and $V^* = HV^*$, V^* is the fixed point of H. Use Banach Fixed Point Theorem [Theorem 6.2.3], we have

$$\lim_{n\to\infty} H^n(V) = V^\pi.$$

(3)

$$\begin{aligned}
&\left\|H^k(V) - V^\pi\right\|_\infty \\
=&\left\|H^k(V) - H^\infty(V)\right\|_\infty \\
=&\left\|\sum_{t=1}^{\infty} H^{t+k}(V) - H^{t+k+1}(V)\right\|_\infty \\
\leq&\sum_{t=1}^{\infty}\left\|H^{t+k}(V) - H^{t+k+1}(V)\right\|_\infty \quad \text{via triangle inequality} \\
\leq&\sum_{t=1}^{\infty}\gamma^t\left\|H^k(V) - H^{k+1}(V)\right\|_\infty \quad \text{via contraction mapping} \\
\leq&\sum_{t=1}^{\infty}\gamma^t\epsilon \\
\leq&\frac{\epsilon}{1-\gamma}.
\end{aligned}$$

We can further refer to similar proofs regarding that the dynamic programming operator is a contraction mapping. See Theorem 36.6.2 and section 6.2. □

A direct application of the value iteration theorem gives the following value iteration algorithm [algorithm 66].

Algorithm 66: Value iteration algorithm for a finite state MDP

Input: MDP, small positive number ϵ as tolerance
Output: Value function $V \approx V^*$ and policy $\pi \approx \pi^*$.

Initialize V arbitrarily. Set ($V(s) = 0$ for all $s \in \mathcal{S}^T$.)
repeat
  $\Delta = 0$
  for $s \in \mathcal{S}$ **do**
    $v = V(s)$
    $V(s) = \max_{a\in\mathcal{A}(s)} \sum_{s'\in\mathcal{S}} P(s'|s,a)(R(s',a) + \gamma V(s'))$
    $\Delta = \max(\Delta, |v - V(s)|)$
  end
until $\Delta < \epsilon$;
Compute the policy $\pi(s) = \max_{a\in\mathcal{A}(s)} \sum_{s'\in\mathcal{S}} P(s'|s,a)(R(s',a) + \gamma V(s'))$.
return V *and* π

Remark 37.1.3 (value iteration vs. policy iteration; model-based vs. model-free.).

- At the first glance, it may seem the simplicity of the value iteration method will make the policy iteration method obsolete. It is critical to know that value iteration requires the knowledge of model, which is represented by the transition probabilities $p(s'|s,a)$.
- Later we will see that for many complicated real-world decision-making problems, a model is a luxury and often unavailable. In such situations, we usually turn to reinforcement learning, a model-free, data-driven approach to learn control policies. Most reinforcement learning algorithms, from a high-level abstraction, are comprised of the two steps of policy evaluation and policy improvement.

37.2 Value-based learning methods

37.2.1 Overview

In the Markov decision process (MDP) framework, with a given model describing system state transitions under different actions, we can use policy iteration or value iteration [subsection 37.1.5]. This section we strategies to learn optimal value function and policy without an explicit model. This typically requires an estimation of the value function from trajectory data associated with a policy. Given an estimated $Q(s, a)$ function, we can achieve policy improvement simply [Theorem 37.1.2] via

$$\pi'(s) = \arg\max_a Q(s, a).$$

Based on how the estimation strategy, we have Monte Carlo method and temporal difference learning; Based on how trajectory data is sampled, we have on-policy method vs. off-policy method.

37.2.2 Monte-Carlo method

37.2.2.1 *On-policy value estimation*

Monte-Carlo method estimation of the value function associated with a policy is based on its straight-forward definition:

$$V^\pi(s) = E^\pi[\sum_{t=0}^{T} \gamma^t R(s_{t+1}, a_t)|s_0 = s, \pi].$$

$$Q^\pi(s, a) = E^\pi\{\sum_{t=0}^{T} \gamma^t R(s_{t+1}, a_t)|s_0 = s, a_t = a\}.$$

The key procedures in the estimation are the trajectory generation and aggregation. For example,

- Generate n trajectories using π, with trajectory i given by $s_0^{(i)}, a_0^{(i)}, r_1^{(i)}, \ldots, s_T^{(i)}$.
- Estimate value function based on the definition

$$V(s_0) = \frac{1}{n}\sum_{i=1}^{n} G^{(i)},$$

where $G^{(i)} = \sum_{t=0}^{T} \gamma^t r_{t+1}^{(i)}$.

In the actual implementation of the algorithm, we can use Markov property of the trajectory and make most use of the data by the following first-visit MC method [algorithm 67]. Note that Monte-Carlo method usually applies to episodic tasks, where each episodes eventually terminate no matter what actions are selected. Since we use averaging as the estimator, the estimation is unbiased and the estimation error falls as $1/\sqrt{n}$, where n is the first-visit samples at state s.

Algorithm 67: First-visit MC value function estimation

Input: policy π, positive integer *num_episodes*
Output: value function estimates V^π and Q^π (if *num_episodes* is large enough)

Initialize counter $N(s) = 0$ and $N(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.
Initialize $returns\_sum(s) = 0$ for all $s \in \mathcal{S}$
Initialize $returns\_sum(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i = 1$ ***to*** *num_episodes* **do**
    Generate an episode $s_0, a_0, r_1, \ldots, s_T$ using π.
    for $t = T - 1$ ***to*** 0 **do**
        $G_t = \sum_{i=t}^{T} r_i$.
        if S_t *is a first visit (i.e.,* s_t *does not appear in* $s_0, a_0, ..., s_{t-1}, a_{t-1}$*)* **then**
            $N(s_t) = N(s_t) + 1$
            $returns\_sum(s_t) = returns\_sum(s_t) + G_t$, where G_t is the accumulated discounted reward starting from time t. $N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$
            $returns\_sum(s_t, a_t) \leftarrow returns\_sum(s_t, a_t) + G_t$
    end
end
$V(s) \leftarrow returns\_sum(s)/N(s)$ for all $s \in \mathcal{S}$
$Q(s,a) \leftarrow returns\_sum(s,a)/N(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
return V

37.2.2.2 *Off-policy value estimation*

In the on-policy Monte-Carlo method, trajectories generated on a policy can only be used to estimate the value functions associated the policy. There are cases where we want to estimate the value function associated with π using trajectories generated under another exploratory policy β. The following theorem gives the theoretical tool to do so.

Lemma 37.2.1 (MC off-policy estimation via importance-sampling). *Given a starting state* s_t*, the probability of the subsequent state-action trajectory,* $a_t, s_{t+1}, a_{t+1}, ..., s_T$ *generated via following policy* π*. We have*

- $$Pr\{a_t, s_{t+1}, a_{t+1}, \ldots, s_T | s_t, a_{t:T-1} \sim \pi\} = \prod_{k=t}^{T-1} \pi\left(a_k|s_k\right) p\left(s_{k+1}|s_k, a_k\right).$$
- *Suppose we want to use trajectories generated under policy β to estimate value function V^π, we have*
$$E^\beta[G_t \rho_{t:T-1} | s_t = s] = V^\pi(s)$$
where $G_t = \sum_{n=t}^{T-1} R_{t+1}$, and
$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi\left(A_k|s_k\right) p\left(s_{k+1}|s_k, a_k\right)}{\prod_{k=t}^{T-1} \beta\left(A_k|s_k\right) p\left(s_{k+1}|S_k, a_k\right)} = \prod_{k=t}^{T-1} \frac{\pi\left(a_k|s_k\right)}{\beta\left(A_k|s_k\right)}$$

Proof. (1)

$$\begin{aligned}
&Pr\{a_t, s_{t+1}, a_{t+1}, \ldots, s_T | s_t, a_{t:T-1} \sim \pi\} \\
&= \pi\left(a_t|s_t\right) p\left(s_{t+1}|s_t, a_t\right) \pi\left(a_{t+1}|s_{t+1}\right) \cdots p\left(s_T|s_{T-1}, a_{T-1}\right) \\
&= \prod_{k=t}^{T-1} \pi\left(a_k|s_k\right) p\left(s_{k+1}|s_k, a_k\right)
\end{aligned}$$

(2) Note that
$$V^\pi(s) = E^\pi[G_t | s_t = s] = E^\beta[G_t \rho_{t:T-1} | s_t = s]$$

□

37.2.2.3 *MC-based reinforcement learning control*

Previous algorithms and theorems provide methods to estimate the value functions associated a policy π. Our generally policy iteration procedure suggest that we can combine value estimation with policy improvement to optimize a control policy.

To facilitate the policy improvement , we estimate Q function in the value estimation, and then improve the policy using greedy procedure $\pi(s) = \arg\max_a Q(s, a)$.

An example algorithm is showed in algorithm 68[2]. Notably, to address the exploration-exploitation dilemma, the action selection is ϵ-greedy, where there is a ϵ probability of selecting random action and an $1 - \epsilon$ probability of selecting $\pi(s) = \arg\max_a Q(s, a)$. In addition, the Q function estimation is different from the aggregation

average method; instead, it employs a stochastic gradient descent step to approach an biased target G_t.

Algorithm 68: MC-based reinforcement learning control

Input: positive integer *num_episodes*, learning rate α, ϵ
Output: policy π ($\approx \pi_*$ if *num_episodes* is large enough)

Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  $\pi \leftarrow \epsilon$-greedy(Q)
  Generate an episode $s_0, a_0, r_1, \ldots, s_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (s_t, a_t) *is a first visit (with return* G_t*)* **then**
      $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(G_t - Q(s_t, a_t))$
  end
end
return π

37.2.3 TD(0) learning

37.2.3.1 *TD(0) for value estimation*

The Monte-Carlo method estimates value function based on the connection of process return and value function. It is arguably the most straight forward method except for the possibility of high variance of the estimate (i.e., sum of random variables can have large variance).

Temporal-difference learning, also known as $TD(0)$, provides a novel route that combines Monte-Carlo and the recursive relations [Lemma 37.1.1, Lemma 37.1.2] between value function. It updates the value function $V^\pi(s)$ estimate via stochastic gradient descent to approaches its **bootstrapped target**

$$E[R(s', \pi(s)) + \gamma V^\pi(s')].$$

In comparison with Monte Carlo methods, TD(0) usually learns faster than Monte-Carlo, at the cost of using a possibly biased bootstrapped target. Monte-Carlo directly learns

the unbiased target, although the return sample is usually of high variance (because the return is the sum of random variables). The algorithm is given in algorithm 69.

Algorithm 69: TD(0) estimation of a value function.

Input: A policy π to be evaluated. Learning rate α
Output: An estimate of V^π
Initialize policy parameter $\theta \in \mathbb{R}^d, w \in \mathbb{R}^m$
repeat
  Initialize initial state s_0
  repeat
    Selection action a according to π, transition to s', and observe reward R.
    Estimate V estimate via $V(s) = V(s) + \alpha(R + \gamma V(s') - V(s))$
    Estimate Q estimate via $Q(s,a) = Q(s,a) + \alpha(R + \gamma Q(s', \pi(s')) - Q(s,a))$
    $s = s'$
  until *S is terminal*;
until *sufficient iterations*;

37.2.3.2 *On-policy reinforcement learning (SARSA)*

The on-policy Q learning involves two steps: the first step is to learn an state-action value function $Q^\pi(s,a)$ for the current behavior policy π for all states s and actions a. The learning target for $Q(s_t, a_t)$ is given by

$$(r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1}, a))).$$

The second is a policy improvement step, where we can get an improved policy at state s via $\pi(s) = \arg\max_a Q(s,a)$.

The resulting algorithm, containing ϵ greedy exploration, is given by algorithm 70.

Algorithm 70: SARSA learning

Input: policy π, ϵ
Output: value function Q
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(terminal\text{-}state, \cdot) = 0$)
repeat
  Intialize $s \notin \mathcal{S}_T$.
  repeat
    Choose an action $a = \arg\max_a Q(s,a)$ with $1-\epsilon$ probability; otherwise choose a random action (i.e., ϵ-greedy).
    Execute the action a, observe the new state s' and receive the reward r.
    Choose an action $a' = \arg\max_a Q(s',a)$ with $1-\epsilon$ probability; otherwise choose a random action(i.e., ϵ-greedy).
    $Q(s,a) = Q(s,a) + \alpha(r_t + \gamma Q(s',a') - Q(s,a))$
    $s = s'$.
  until s_t *is in terminal states* $\mathcal{S}_T$;
until *succificent iterations*;

37.2.3.3 *Off-policy reinforcement learning (Q learning)*

The off-policy Q learning involves two iterative steps: the first step is to directly learn/approximate the optimal state-action value function $Q^*(s,a)$ (based on current estimated $Q(s,a)$) independent of the current behavior policy π for all states s and actions a. The learning target is given by

$$(r_{t+1} + \gamma \max_a Q(s_{t+1}, a)).$$

The second is a policy improvement step, where we can get an improved policy at state s via $\pi(s) = \arg\max_a Q(s,a)$.

The resulting algorithm, contains ϵ greedy exploration, is given by algorithm 70.

Algorithm 71: Q-learning algorithm

Input: *epsilon*, discount factor γ, and learing rate α

Initialize Q. **repeat**

  Intialize $s \notin \mathcal{S}_T$. **repeat**

    Choose an action $a = \arg\max_a Q(s,a)$ with $1-\epsilon$ probability; otherwise choose a random action (i.e., ϵ-greedy).

    Execute the action a, observe the new state s' and receive the reward r.

    Update Q function estimate via

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a \in A(s')} Q(s',a'))$$

    $s = s'$

  until s_t *is in terminal states* $\mathcal{S}_T$;

until *sufficient iteration*;

Output: The Q function.

Remark 37.2.1. The convergence of Q learning algorithm has been addressed at [3, p. 495]. Essentially, there are two requirements:

- All state control pairs (s,a) must be generated infinitely often within the infinitely long sequence $\{(s_k, a_k)\}$.
- The stepsize/learning rate should be diminishing and satisfying the following conditions:

$$\alpha_k > 0, \forall k, \sum_{k=1}^{\infty} \alpha_k = \infty, \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

  One choice would be $\alpha_k = \frac{1}{n}$.

Remark 37.2.2 (on-policy learning (SARSA) vs. off-policy learning (Q learning)).

- The reason that SARSA is on-policy is that it updates its Q function towards the Q^π, represented by the Q-value of the next state s' and the current policy's action a''. It estimates the return for state-action pairs assuming the current policy (ϵ-greedy) continues to be followed.
- The reason that Q-learning is off-policy is that it updates its Q function towards the Q^*, represented by the Q-value of the next state s' and the greedy action a'.
- In a nutshell, Q learning is updating Q function in a more aggressive way. the bootstrapped Q function at (s_{t+1}, a_{t+1}) in SARSA is

$$Q(s_{t+1}, a_{t+1}) = \epsilon \cdot E_a[Q(s_{t+1}, a)] + (1-\epsilon) \cdot \max_a Q(s_{t+1}, a);$$

in Q-learning it is

$$Q(s_{t+1}, a_{t+1}) = \max_a Q(s_{t+1}, a).$$

37.2.4 TD(n) learning

37.2.4.1 *Motivation and concepts*

One step temporal difference learning, or TD(0) learning usually contains error due to the learning of a bootstrapped target. To see this, consider one-step Sarsa algorithm that uses the target

$$G_t = R_{t+1} + \gamma Q_\theta(s_{t+1}, a_{t+1}).$$

Likewise, in the one-step value estimation, we have target for V given by

$$G_t = R_{t+1} + \gamma V_\theta(s_{t+1}).$$

Because $Q_\theta(s_{t+1}, a_{t+1})$ and $V_\theta(s_{t+1})$ are the estimates of Q^π and V^π, i.e., the target are biased, we propagate error when we use a biased target.

On the other extreme, Monte-Carlo methods like the REINFORCE algorithm uses the entire sequence to estimate values, with the target given by

$$G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^K V(s_{t+K}), s_{t+K} \in \mathcal{S}_T.$$

The target is unbiased, but Monte-Carlo methods tend to learn slowly due to the high variance (because of the summation of multiple random variables).

TD(n) is a method between the two extremes: the target contains n-step forward rewards plus a bootstrapped component. TD(n) combines the two strengths of TD(0) and Monte Carlo and addresses their weaknesses, as TD(n) produces a target with less biased than TD(0) and of smaller variance than Monte-Carlo.

The construction of the value target in TD(n), with $1 \leq n \leq \infty$, is showed in Table 37.2.1. Clearly, TD(n) is the generalization of TD(0) and the Monte Carlo method.

n	G_t
$n = 1$	$G_t^{(1)} = R_{t+1} + \gamma V(s_{t+1})$
$n = 2$	$G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(s_{t+2})$
$\dots$	$\dots$
$n = n$	$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^n V(s_{t+n})$
$\dots$	$\dots$
$n = \infty$	$G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^K V(s_{t+K}), s_{t+K} \in \mathcal{S}_T$

Table 37.2.1: Estimating cumulative rewards $G_t^{(n)}$ of different steps n as the target for value function V. $G^{(1)}$ corresponds to temporal-difference TD(0) and $G^{(\infty)}$ corresponds to Monte-Carlo estimation. If the process terminates at K and $K < n$, then we use $G_t^{(n)} = G_t^{(K)}$. Trajectories are generated under policy π.

37.2.4.2 *TD(n) for value estimation*

Like TD(0) and Monte-Carlo Methods, we can use TD(n) to estimate the value function associated with a policy π, with the algorithm given in algorithm 72. Note that in our algorithm, we use a buffer of size n to store the previous n transitions and rewards in order to construct G_t^n.

Although this algorithm is estimating V, we can estimate $Q(s_t, a_t)$ by changing the target $G_t^{(n)}$ to

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^n Q(s_{t+n}, a_{t+n}),$$

where the first step action is a_t, not necessarily $\pi(s_t)$.

Algorithm 72: TD(n) estimation of a value function.

Input: A policy π to be evaluated. Learning rate α. A buffer of size n to store transitions and reward in the last n steps.

Output: An estimate of V^π

repeat
  Initialize initial state s_0
  repeat
    Selection action a according to π, transition to s', and observe reward R.
    Using the buffer to store R. Construct G_t^n using previous stored R is the buffer. If not enough observation for construction, then continue. Estimate V estimate via $V(s_{-n}) = V(s_{-n}) + \alpha(R + \gamma V(s') - V(s_{-n}))$
    Estimate Q estimate via

$$Q(s_{-n}, a) = Q(s_{-n}, a) + \alpha(R + \gamma Q(s', \pi(s')) - V(s_{-n})),$$

    Note hat s_{-n}, a_{-n} are state and action n steps ago, retrieved from the buffer. $s = s'$.
  until *s is terminal*;
until *sufficient iterations*;

37.2.4.3 *TD(n) for reinforcement learning control*

Similar to how we turn TD(0) value estimation algorithm to a reinforcement learning control algorithm, by adding a policy improvement steps, we get yield a n-step SARSA algorithm [algorithm 73].

Algorithm 73: n-step SARSA learning

Input: policy π, ϵ_i, learning rate α. A buffer of size n to store transitions and reward in the last n steps.

Output: value function Q

Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\textit{terminal-state}, \cdot) = 0$)

repeat

  Intialize $s \notin \mathcal{S}_T$.

  repeat

    Choose an action $a = \arg\max_a Q(s,a)$ with $1-\epsilon$ probability; otherwise choose a random action(i.e., ϵ-greedy).

    Execute the action a, observe the new state s' and receive the reward r.

    Construct G_t^n using previous stored R is the buffer. If not enough observation for construction, then continue. Choose an action $a' = \arg\max_a Q(s',a)$ with $1-\epsilon$ probability; otherwise choose a random action(i.e., ϵ-greedy).

    $$Q(s_{-n}, a_{-n}) = Q(s_{-n}, a_{-n}) + \alpha(G_t^{(n)} + \gamma Q(s', a') - Q(s_{-n}, a_{-n}))$$

    where s_{-n}, a_{-n} are state and action n steps ago, retrieved from the buffer. $s = s'$.

  until s_t *is in terminal states* $\mathcal{S}_T$;

until *succificent iterations*;

return Q

37.2.5 Standing challenges in reinforcement learning

37.2.5.1 *Curse of dimensionality*

Most RL algorithm requires the learning of $Q(s,a)$ or $V(s)$ for *all* state s and action a. In tabular RL, where Q and V are stored as multi-dimensional arrays, increasing dimensionality will exponentially increase the memory storage requirement of value functions. One popular solution is to used parameterized functions to represent Q and

V, including linear function approximators using predefined basis functions to nonlinear approximators such as neural networks.

37.2.5.2 *Sample efficiency*

Sample efficiency is regarding how many data samples are needed if we want to learn a competitive policy. Although RL have achieved remarkable performance in problems there are sufficient data, many real-world problems are facing the hurdle of data scarcity and hinder the application of RL. How to collect data sample and how to efficiently use sample have received extensive attention. Common strategies include off-policy to reuse data sample, curriculum learning to enable smart exploration of parameter space, etc.

37.2.5.3 *Exploration-exploitation dilemma*

Because of the nature of model-free learning, the knowledge of the environment is from agent-environment interaction data. In one limit, if the agent explores the environment sufficiently to obtain a model of the environment, RL reduces to model-based MDP. Complicated problems are usually characterized by their immense state space or high cost (e.g., time, money) to collect more data and it is prohibitive to construct a high fidelity model by exploring the environment comprehensively. On the other hand, if we rush for the policy based on an substantially incomplete representation of environment, we can possibly end up with a mediocre control policy.

The dilemma stems from need to make better decisions based relatively incomplete information. Different RL algorithms cope with the dilemma in a different fashion, including ϵ greedy action selection, stochastic policy with entropy penalty, addition of noise into policy parameter space, etc.

37.2.5.4 *Deadly triad*

In solving complex RL problems, it is almost inevitable to use nonlinear function approximators for state value estimations. When using bootstrapping TD methods with off-policy learning, value estimation error can propagate in a non-controllable way, leading to divergence. The three issues combined, nonlinear approximator, bootstrapping, and off policy, are to known as **the deadly triad** [4].

37.3 Policy gradient learning

37.3.1 Stochastic policy gradient fundamentals

37.3.1.1 *Preliminaries: deriative and expectation*

We first review some useful results that will be used repeatedly in the following subsections devoted to the policy gradient theory.

Lemma 37.3.1 (derivative and expectation identities for stochastic policy gradient). *Suppose functions and integrals we consider here are well-behaved such that derivative and integration can be exchanged.*

- *Let $g_\theta(x)$ be a function parameterized by θ. We have*

$$\nabla_\theta \int f(x) g_\theta(x) dx = \int f(x) \nabla_\theta g_\theta(x) dx$$

- *(**log-derivative trick**) Let p_θ be a θ-parameterized density function for random varaible x, we have*

$$\nabla_\theta E_{X\sim p_\theta}[f(X)] = E_{x\sim p_\theta}[f(X)\nabla_\theta \log p_\theta(X)],$$

  where $E_{X\sim p_\theta}$ indicates expectation is taken with respect to random variable X whose the distribution is given by $p_\theta(X)$.

- $$\nabla_\theta E_{X\sim p_\theta}[p_\theta(X)] = 0.$$

  Moreover, if we let b_0 be a constant and $b(theta)$ be a function only dedends on θ, we have

$$\nabla_\theta E_{X\sim p_\theta}[b_0 p_\theta(X)] = 0, \nabla_\theta E_{X\sim p_\theta}[b(\theta) p_\theta(X)] = 0.$$

Proof. (1) Note that $f(x)$ is not dependent on θ. Also check the following example. (2) Using (1), we have

$$\begin{aligned}
\nabla_\theta E_{X\sim p_\theta}[f(X)] &= \int f(x) \nabla_\theta p_\theta(x) dx \\
&= \int f(x) \nabla_\theta p_\theta(x) dx \\
&= \int f(x) p_\theta(x) \nabla_\theta \log p_\theta(x) dx \\
&= E_{X\sim p_\theta(\tau)}[f(x)\nabla_\theta \log p_\theta(X)].
\end{aligned}$$

(3) Use (2) in reverse. We have

$$E[\nabla_\theta \log p_\theta(X)] = \nabla_\theta E[1] = 0.$$

For the rest, we see the b_0 and $b(\theta)$ can be pulled out of the expectation.

□

Example 37.3.1. Consider $f(x) = x^2, p_\theta = \frac{1}{\sqrt{2\pi}\theta} \exp(-\frac{(x-\theta)^2}{2\sigma^2})$, we have

$$\begin{aligned}
&\frac{d}{d\theta} \int f(x) p_\theta(x) dx \\
=&\frac{d}{d\theta} \int x^2 \frac{1}{\sqrt{2\pi}\theta} \exp(-\frac{(x-\theta)^2}{2\sigma^2}) dx \\
=&\int x^2 \frac{d}{d\theta} \frac{1}{\sqrt{2\pi}\theta} \exp(-\frac{(x-\theta)^2}{2\sigma^2}) dx \\
=&\int x^2 \frac{(x-\theta)}{\sigma} \frac{1}{\sqrt{2\pi}\theta} \exp(-\frac{(x-\theta)^2}{2\sigma^2}) dx \\
=&\int f(x) \frac{dp_\theta}{d\theta} dx
\end{aligned}$$

37.3.1.2 *Theoretical framework based on finite-horizon trajectories*

Policy gradient methods aim to learn a parameterized policy to directly select actions without using value functions. Specifically, we will parameterize the policy by θ such that a stochastic policy is given by

$$\pi(a|s, \theta) = Pr(a_t = a|s_t = s, \theta),$$

and a deterministic policy is given by $\mu(a|s, \theta)$. The ultimate goal in policy gradient methods boils downs to seeking the optimal parameter θ^* such that expected cumulative rewards during decision-making processes are maximized.

The mainstream methodology to optimize the policy parameter θ is gradient descent. Therefore calculating or estimating the gradient with respect to θ is the cornerstone for nearly all policy gradient algorithms.

The pros and cons of policy gradient methods, compared with value based methods, are

- Policy-based methods usually have better convergence properties as they directly work in the policy space.

- Policy gradients are more effective in high dimensional action spaces, especially when using continuous actions.
- Policy gradient method can learn a stochastic policy, while value-function-based method cannot.

This section we will introduce basic definition and procedures regarding how to estimate policy gradient in a finite-horizon trajectory setting. Next section we cover policy gradient concept and calculation in the infinite-horizon setting. The later is theoretically more challenging than the former, but its results is elegent and shed light on the connection to the value based methods we covered in our previous sections.

We start with basic definitions in the finite-horizon setting.

Definition 37.3.1 (basic concepts).

- *(stochastic policy) A (stochastic) policy π_θ is a differential function parameterized by θ, given by*
$$\pi(a|s,\theta) = Pr(A_t = a, \theta_t = \theta), \theta \in \mathbb{R}^d.$$
- *Assume Markov model for the state transition dynamics. Consider a finite-length trajectory generated by carrying out policy πdefined by $\tau = \{s_0, a_0, s_1, ..., a_{T-1}, s_T\}$. Then we define the **trajectory probability** by*
$$p_\theta(\tau) = \rho_0(s_0) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t).$$
where ρ_0 is the initial state distribution, $p(s_{t+1}|s_t, a_t)$ is the transition probability specified by the Markov chain.
- *Let $r_1 = R(s_1, a_0), r_2 = R(s_2, a_1), ..., r_T$ be the rewards associated with above trajectory. We define trajectory reward $R(\tau)$ as*
$$R(\tau) = \sum_{t=1}^{T} r_t$$

The objective function to optimize θ is the expected trajectry rewards, which is given as follows.

Definition 37.3.2 (objective function of policy gradient method). *The objective function for policy parameter θ optimization is*
$$\max_\theta J(\theta) = \max_\theta E_{\tau \sim p_\theta(\tau)}[R(\tau)],$$

where τ is a stochastic trajectory of lenth T generated from policy $\pi_\theta(a|s)$, and $p_\theta(\tau)$ is the trajectory probabiity under policy parameter θ.

Now we are in a position to introduce the theory underlying calculating the gradient of $J(\theta)$ with respect to θ in the following Th.. Analytical calculation based on this theorem is usually intractable. We postpond numerical approaches to following sections.

Theorem 37.3.1 (stochastic policy gradient on finite-horizon trajectories). *With the reward-to-go function given by*

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}[r(\tau)] = \int r(\tau)p_\theta(\tau)d\tau,$$

it gradient is

$$\begin{aligned}\nabla_\theta J(\theta) &= \int r(\tau)\nabla_\theta p_\theta(\tau)d\tau \\ &= E_{\tau\sim p_\theta(\tau)}[(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t))(\sum_{t=0}^{T-1} R(s_{t+1}, a_t))]\end{aligned}$$

Proof. First, we have [Lemma 37.3.1]

$$\begin{aligned}\nabla_\theta J(\theta) &= \int R(\tau)\nabla_\theta p_\theta(\tau)d\tau \\ &= \int R(\tau)\nabla_\theta p_\theta(\tau)d\tau \\ &= \int R(\tau)p_\theta(\tau)\nabla_\theta \ln p_\theta(\tau)d\tau \\ &= E_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla_\theta \ln p_\theta(\tau)].\end{aligned}$$

Note that

$$\begin{aligned}p_\theta(\tau) &= p(s_1)\prod_{t=0}^{T-1}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t) \\ \implies \ln p_\theta(\tau) &= \ln \rho_0(s_0) + (\sum_{t=0}^{T-1}\ln \pi_\theta(a_t|s_t) + \ln p(s_{t+1}|s_t, a_t)) \\ \implies \nabla_\theta \ln p_\theta(\tau) &= \sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t)\end{aligned}$$

Plug into $\nabla_\theta J(\theta)$, we have

$$\begin{aligned}\nabla_\theta J(\theta) &= E_{\tau\sim p_\theta(\tau)}[(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t))(R(\tau))] \\ &= E_{\tau\sim p_\theta(\tau)}[(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t))(\sum_{t=0}^{T-1}R(s_{t+1},a_t))]\end{aligned}$$

□

37.3.1.3 *Theoretical framework based on distributions**

Besides formulating policy gradient based on finite-length trajectories, we can also approach policy gradient in the context of stationary Markov chain specified by system dynamics and control policy. However, the derivation is significantly abstract and difficult. In general, policy gradient theorem based on finite-length trajectories is sufficient to understand most of algorithms.

Definition 37.3.3. *We assume finite state Markov decision process and the Markov chain can reach equilibrium under arbitrary stochastic policy.*

- *(stochastic policy) A (stochastic) policy π_θ is a differential function parameterized by θ, given by*
$$\pi(a|s,\theta) = Pr(A_t = a, \theta_t = \theta), \theta \in \mathbb{R}^d.$$
- *stationary distribution under stochastic policy π, denoted by $d^\pi(s)$, is the stationary distribution of Markov chain under policy π. Mathematically,*
$$d^\pi(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi).$$
Note that $d^\pi(s)$ is independent of initial state.
- *The stationary reward function $J(\theta)$ is defined by*
$$J(\theta) = \sum_{s\in\mathcal{S}} d^\pi(s)V^\pi(s) = \sum_{s\in\mathcal{S}} d^\pi(s)\sum_{a\in\mathcal{A}}\pi_\theta(a|s)Q^\pi(s,a),$$
where we use the relation
$$\sum_{a\in\mathcal{A}}\pi_\theta(a|s)Q^\pi(s,a) = V^\pi(s).$$
- *Discounted reward-to-go function associated with policy π:*
$$J(\pi) = E[\sum_{k=1}^{\infty}\gamma^{k-1}r(s_k,a_)|\pi].$$

- *State-action value function associated with policy π, which is defined as the expected return starting from state s, taking action a and **thereafter following** the policy π. It is given by*

$$Q^\pi(s,a) = E[\sum_{k=1}^{\infty} \gamma^{k-1} r(s_k, a_k) | S_1 = s, A_1 = a; \pi].$$

- *Value function associated with policy π:*

$$V^\pi(s) = E[\sum_{k=1}^{\infty} \gamma^{k-1} r(s_k, a_)) | S_1 = s; \pi],$$

  which the total expected return starting from state s, taking actions specified by the policy π.
- *Discounted state distribution functions*

$$\rho(s') = \int_{\mathcal{S}} \sum_{k=1}^{\infty} \gamma^{k-1} p_1(s) p(s \to s', t, \pi),$$

  and we can write

$$J(\theta) = \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s,\theta) r(s,a) da ds = E_{s \sim \rho(s), a \sim \pi(s)}[r(s,a)].$$

Definition 37.3.4 (discounted state distribution).

- *$\rho_0(s)$ initial distribution over states.*
- *k step transition probability $\rho^\pi(s \to s', k)$ characterizes the probability distribution for a Markov chain, starting from state s, after moving k steps according to policy π.*
- *Discounted state distribution*

$$\rho^\pi(s') = \int_{\mathcal{S}} \sum_{k=1}^{\infty} \gamma^{k-1} \rho_0(s) \rho(s \to s', t, \pi),$$

  and we can write

$$J(\theta) = \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s,\theta) r(s,a) da ds = E_{s \sim \rho(s), a \sim \pi(s)}[r(s,a)].$$

- *discounted stationary reward function is given by*

$$J(\theta) = \int_{s \in \mathcal{S}} \rho^\pi(s) \int_{a \in \mathcal{A}} Q(s,a) \pi_\theta(a|s) ds da = E_{s \sim \rho^\pi}[\int_{a \in \mathcal{A}} Q^\pi(s,a) \pi_\theta(a|s)].$$

Theorem 37.3.2 (policy gradient theorem on distributions). *[2, p. 325]*

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in S} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi_\theta(a|s) \\ &\propto \sum_{s\in S} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s) \\ &= E_{s\sim d^\pi}[\sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|s)]\end{aligned}$$

note that both Q and π are functions of θ.

Proof.

$$\begin{aligned}&\nabla_\theta V^\pi(s) \\ =&\nabla_\theta \left(\sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^\pi(s,a)\right) \\ =&\sum_{a\in\mathcal{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s)\nabla_\theta Q^\pi(s,a)\right) \\ =&\sum_{a\in\mathcal{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s)\sum_{s'} P\left(s'|s,a\right)\nabla_\theta V^\pi\left(s'\right)\right) \\ =&\sum_{a\in\mathcal{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s)\sum_{s'} P\left(s'|s,a\right)\nabla_\theta V^\pi\left(s'\right)\right)\end{aligned}$$

where in the third line we remove r since it does not depend on θ.

Therefore, the gradient has the following recursive form

$$\nabla_\theta V^\pi(s) = \sum_{a\in\mathcal{A}} \left(\nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) + \pi_\theta(a|s)\sum_{s'} P\left(s'|s,a\right)\nabla_\theta V^\pi\left(s'\right)\right).$$

Denote $\phi(s) = \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)$, we have

$$
\begin{aligned}
&\nabla_\theta V^\pi(s) \\
=&\phi(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P\left(s'|s,a\right) \nabla_\theta V^\pi\left(s'\right) \\
=&\phi(s) + \sum_{s'} \sum_a \pi_\theta(a|s) P\left(s'|s,a\right) \nabla_\theta V^\pi\left(s'\right) \\
=&\phi(s) + \sum_{s'} \rho^\pi\left(s \to s', 1\right) \nabla_\theta V^\pi\left(s'\right) \\
=&\phi(s) + \sum_{s'} \rho^\pi\left(s \to s', 1\right) \left[\phi\left(s'\right) + \sum_{s''} \rho^\pi\left(s' \to s'', 1\right) \nabla_\theta V^\pi\left(s''\right)\right] \\
=&\phi(s) + \sum_{s'} \rho^\pi\left(s \to s', 1\right) \left[\phi\left(s'\right) + \sum_{s''} \rho^\pi\left(s' \to s'', 1\right) \nabla_\theta V^\pi\left(s''\right)\right] \\
=&\phi(s) + \sum_{s'} \rho^\pi\left(s \to s', 1\right) \phi\left(s'\right) + \sum_{s''} \rho^\pi\left(s \to s'', 2\right) \nabla_\theta V^\pi\left(s''\right) \\
=& \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \rho^\pi(s \to x, k) \phi(x)
\end{aligned}
$$

Note that we interpret $\sum_{k=0}^{\infty} \rho^\pi(s \to x, k)$ as a scalar proportional to the stationary distribution at x, independent of s. Therefore

$$\nabla_\theta J(\theta) = \nabla_\theta V^\pi(s) \propto \sum_{s \in S} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a) \nabla_\theta \pi_\theta(a|s)$$

□

Lemma 37.3.2. *Let $\rho(s)$ be the equilibrium distribution under policy π. Let π be parameterized by θ. Let the reward-to-go function be*

$$J(\theta) = \int_{s \in \mathcal{S}} \rho^\pi(s) \int_{a \in \mathcal{A}} Q(s,a) \pi_\theta(a|s) ds da = E_{s \sim \rho^\pi}[\int_{a \in \mathcal{A}} Q^\pi(s,a) \pi_\theta(a|s)].$$

We have

$$\nabla_\theta J(\theta) = E_{s \sim \rho(s), a \in \pi(s)}[\nabla_\theta \ln \pi(a|s,\theta) Q^\pi(s,a)]$$

Proof.

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \nabla_\theta \pi(a|s,\theta) Q^\pi(s,a) dads \\ &= \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s,\theta) \frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)} Q^\pi(s,a) dads \\ &= \int_{\mathcal{S}} \rho(s) \int_{\mathcal{A}} \pi(a|s,\theta) \nabla_\theta \ln \pi(a|s,\theta) Q^\pi(s,a) dads \\ &= E_{s\sim\rho(s), a\in\pi(s)}[\nabla_\theta \ln \pi(a|s,\theta) Q^\pi(s,a)]\end{aligned}$$

□

37.3.1.4 *Estimate policy gradient and basic algorithms*

In Theorem 37.3.1 and Theorem 37.3.2, we have established that

$$\nabla_\theta J(\theta) = E_{\tau\sim p_\theta(\tau)}[(\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t|s_t))(\sum_{t=0}^{T-1} R(s_{t+1}, a_t))].$$

We can use Monte Carlo to estimate the expectation in the following way.

Methodology 37.3.1 (unbiased estimator of total expected rewards). *Suppose we have used policy π_θ to generate N sample trajectories $\tau_1, ..., \tau_N$ of length T. Let $s_{i,t}, a_{i,t}$ represent the sample value of state and action at trajectory i and time t. Then we can estimate*

$$E_{\tau\sim p_\theta(\tau)}[\sum_{t=0}^{T-1} R(s_{t+1}, a_t)]$$

via

$$\frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} R(s_{i,t+1}, a_{i,t})$$

Remark 37.3.1 (connection maximum likelihood estimation). In the policy gradient method, we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta R(\tau_i) \ln p_\theta(\tau_i).$$

Similarly, the gradient for the log-likelihood function based on trajectories $\tau_1, ..., \tau_N$ is given by

$$\nabla_\theta L(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ln p_\theta(\tau_i).$$

Therefore, policy gradient is similar to a weighted version of log-likelihood function. The weight is given by the trajectory reward. The policy gradient method thus maximizes the likelihood of trajectories with high rewards and minimizes the likelihood of trajectories with low rewards.

Now we have our first policy gradient algorithm based on batch trajectory update, given by algorithm 74.

Algorithm 74: A generic batch policy-gradient algorithm (REINFORCE)

Input: A policy differential function parameterized $\pi(a|s,\theta)$, learning rate $\alpha > 0$.
Output: Improved policy $\pi(a|s,\theta^*)$ parameterized by θ^*

Initialize policy parameter $\theta \in \mathbb{R}^d$
repeat
  Using current policy $\pi(\cdot|\cdot,\theta)$ to generate N sample trajectories.
  Estimate policy gradient

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}[(\nabla_\theta \ln \pi_\theta(a_{i,t}|s_{i,t}))(\sum_{t=1}^{T} R(s_{i,t},a_{i,t}))].$$

  Update $\theta = \theta + \alpha\nabla_\theta J(\theta)$.
until *sufficient iterations*;
return V

Remark 37.3.2 (interpretation and issues with poor credit assignment and high variance).

- In the algorithm, we are taking gradient steps to maximize the probability of taking certain actions that lead to higher trajectory reward $R(\tau)$. Particularly, if $R(\tau)$ is high, we will increase the probability of actions observed in τ, and vice versa.
- However, this method has a severe drawback: since we only compute $R(\tau)$ at the end of an episode, if some of the actions taken in τ are truly bad then $R(\tau)$ is large, we will still promote the probability of these bad actions.
- Another draw back is the high variance of the stochastic quantity $R(\tau)$, which is the summation of a possibly very long sequence of random variables $\sum_t R(s_{t+1},a_t)$.

One way to improve the credit assignment and reduce the variance of gradient estimates is to simplify the gradient estimation via **causal relationship**. We have

$$\begin{aligned}\nabla_\theta J(\theta) &= E_{\tau\sim p_\theta(\tau)}[(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t))(\sum_{t=0}^{T-1} R(s_{t+1},a_t))] \\ &= E_{\tau\sim p_\theta(\tau)}[(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t))(\sum_{t'=t}^{T} R(s_{t'+1},a_{t'}))]\end{aligned}$$

This is because for every t, the quantity

$$E_{\tau \sim p_\theta(\tau)}[(\nabla_\theta \ln \pi_\theta(a_t|s_t))(\sum_{t'=1}^{t-1} R(s_{t'+1}, a_{t'}))]$$

does not depend on θ. Intuitively, past trajectories does not depend on $\pi_\theta(a_t|s_t)$. Because we have fewer summation terms, the variance will in general decrease.

These improvements give us a basic Monte-Carlo policy gradient algorithm [algorithm 75].

Algorithm 75: A basic Monte-Carlo policy-gradient algorithm

Input: A policy differential function parameterized $\pi(a|s, \theta)$, learning rate $\alpha > 0$.
Output: Improved policy $\pi(a|s, \theta^*)$ parameterized by θ^*

```
Initialize policy parameter θ ∈ ℝ^d
repeat
    Using current policy π(·|·, θ) to generate an episode (s_1, a_1, r_1), ..., (s_T, a_T, r_T).
    for each step in the episode t ∈ {1, ..., T} do
        G = return from step t
        θ = θ + α γ^t G ∇_θ ln π(a_t|s_t, θ)
    end
until sufficient iterations;
return V
```

37.3.1.5 *Bootstrap and Actor-Critic methods*

In our previous section, we can see that we can estimate the gradient with respect to θ from trajectories via

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} [(\nabla_\theta \ln \pi_\theta(a_{i,t}|s_{i,t}))\hat{Q}_{i,t}],$$

where $\hat{Q}_{i,t'} = \sum_{t=t'}^{T-1} R(s_{i,t+1}, a_{i,t})$. In the REINFORCE algorithm and the Monte-Carlo algorithm, we estimate the cumulative rewards $\sum_{t=0}^{T-1} R(s_{i,t}, a_{i,t-1})$ directly from samples, which in general come with large variance (also see subsection 37.2.4]. Similar to how we cope with the large variance issue in value based reinforcement learning, we can use bootstrap to reduce variance at the price of some bias. First, we can view $\hat{Q}_{i,t}$ is one sample from state-action value function defined by

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^{T-1} E_{\pi_\theta}[r(s_{t'+1}, a_{t'})|s_t, a_t]$$

In the REINFORCE algorithm and the Monte-Carlo algorithm, these samples are only used once and then discarded. Alternatively, if we maintain a function approximator that continuously improve approximate Q or V, we can write our gradient estimate by

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}[(\nabla_\theta \ln \pi_\theta(a_{i,t}|s_{i,t}))\hat{Q}(s_{i,t},a_{i,t})],$$

Now we can decompose the estimation of policy gradient into two components, first component $\nabla\pi$ and second component is $\hat{Q}(s,a)$ or $\hat{V}(s)$. The Actor-Critic is the implementation of above idea, where we use a function approximator, called Actor, responsible to generate and update the policy, and a function approximator, called Critic, to approximate the value function that will asist in the estimation of policy gradients. The critic parameters will be updated based on bootstrapping and Bellman's equation and the Actor will updated using policy gradient.

Below is a basic Actor-Critic algorithm [algorithm 76]. Note that we update the Critic by least square optimization of

$$\frac{1}{2}(y - Q(s,a|\theta^C))^2,$$

where y is the target given by $y = R(s,a) + Q(s',a';\theta^Q), a' \sim \pi(a|s',\theta^A)$. If $s \in \mathcal{S}_T, y = R(s,a)$ and update the Actor via

$$\theta^A = \theta^A + \alpha\hat{Q}(s,a;\theta^C)\nabla_\theta\pi(a|s,\theta^A).$$

Algorithm 76: Actor-Critic policy-gradient method

Input: A policy differential function parameterized $\pi(a|s,\theta^A)$, a differentiale state-value parameterization $Q(s,a;\theta^C)$, learning rate $\alpha_1 > 0, \alpha_2 > 0$.

Output: Improved policy $\pi(a|s,\theta^{A*})$

Initialize Actor and Critic parameters.

repeat

  Initialize initial state s_0

  repeat

    Take action $a \sim \pi(\cdot|S,\theta)$, observe s', r.

    Construct target for Critic: $y = R(s,a) + Q(s',a';\theta^Q), a' \sim \pi(a|s',\theta^A)$. If $s \in \mathcal{S}_T, y = R(s,a)$ Perform gradient descent update on θ^C with learning rate α_1 on the objective function

$$\frac{1}{2}(y - Q(s,a|\theta^C))^2.$$

    Perform gradient descent update on θ^A with learning rate α_1 using

$$\theta^A = \theta^A + \alpha\hat{Q}(s,a;\theta^C)\nabla_\theta \pi(a|s,\theta^A)$$

    $s = s'$

  until *s is terminal;*

until *sufficient iterations;*

return V

Remark 37.3.3 (interpretation). We can interpret the algorithm in the policy evaluation and policy improvement framework. The Actor-Critic algorithm is performing on-policy update. The improvement of the policy is achieved by gradient descent on policy parameter θ^A and policy evaluation is achieved by estimating $Q(s,a;\theta^C)$.

37.3.1.6 *Common stochastic policies and their representations*

Commonly used stochastic policies and their neural network realization for discrete and continuous action spaces are given below.

Definition 37.3.5 (categorical policy). *A categorical policy samples a set of discrete actions based on some probabilities. Its neural network architecture could be a feed-forward network with softmax layer on the final output layer, similar to neural network architecture for classification problems. The output of the network is the probability over the actions, denoted by* $\pi_\theta(a|s) = Pr_\theta(a)$.

Definition 37.3.6 (multivariate Gaussian policy). *A multivariate Gaussian policy draws actions from a multivariate Gaussian distribution, charachterized by a mean vector μ and a covariance matrix Σ. For simplicity, diagonal Σ is often assumed.*

A diagonal Gaussian policy always has a neural network that maps from observations to mean actions, $\mu_\theta(s)$. There are two different ways that the covariance matrix is typically represented.

Often, we employ feed-forward neural networks [Figure 37.3.1(a)] to directly map state s to mean vector and the log standard deviation, respectively. The mean network and the log std network might share some layers.

The final action is synthesized by

$$a = \mu_\theta(s) + \sigma_\theta(s) \odot z,$$

where $\odot$ denotes the element-wise product of two vectors, z is a standard Gaussian random vector.

Note that given a policy $\pi_{\pi,\sigma}$, the probability of sampling a is given by

$$\log \pi_{\mu,\theta}(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i-\mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right).$$

Definition 37.3.7 (generic stochastic policy). *Harnessing the universal representation power of neural networks, we can further parameterize a stochastic policy via Figure 37.3.1(b). In the network, we feed a state s and a Gaussian noise vector ϵ into the network. We can view the output of the network as a random variable transformed from z, conditioning on the information s.*

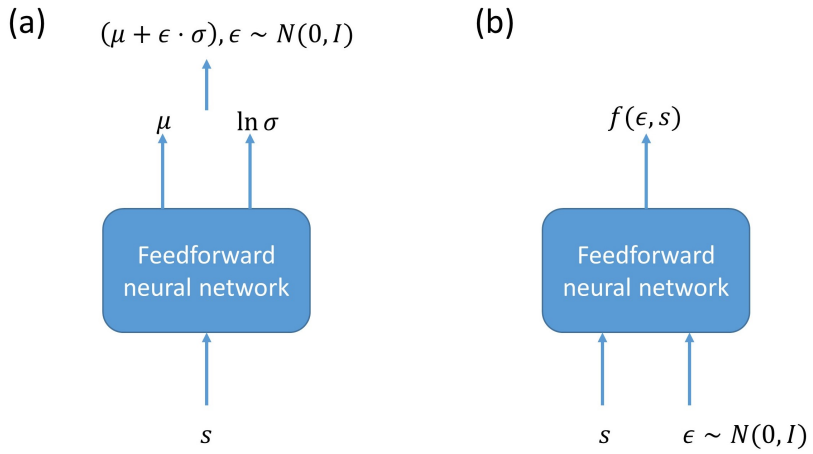


Figure 37.3.1: Neural network parameterization for a Gaussian policy (a) and (b) a generic stochastic policy.

37.3.2 Advanced methods for policy gradient estimation

37.3.2.1 *Stochastic policy gradient with baseline*

In practice, the gradient estimate from trajectories can have large variance. One way to remedy is to subtract the cumulative reward statistics by a baseline function, which is a function or depending on the trajectories. To understand how it works, let's first look at the variance of policy gradient when we subtract a baseline. Let b be the baseline, we have the modified gradient given by

$$\nabla_\theta J(\theta) E_{\tau \sim p_\theta(\tau)} \nabla_\theta \log \pi_\theta(\tau)[R(\tau) - b].$$

It is critical to note that subtracting the baseline function will not introduce bias into the policy gradient.

Lemma 37.3.3 (gradient of baseline function). *Let $b(s)$ be a baseline differential function that it is not a function of the trajectory τ. Let $\pi(a|s, \theta)$ be a θ parameterized stochastic policy. We have*

$$\begin{aligned}\nabla_\theta J(\theta) &= E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log \pi(a|s,\theta)(R(\tau)-b)] \\ &= E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log \pi(a|s,\theta)R(\tau)]\end{aligned}$$

Proof. Use Lemma 37.3.1. Note that

$$E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log \pi(a|s,\theta)b] = b \cdot E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log \pi(a|s,\theta)] = b \cdot 0 = 0.$$

□

To see how the baseline can modify the variance of the policy gradient estimate, we have

$$\begin{aligned}Var[\nabla_\theta J(\theta)] &= Var_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau)[R(\tau)-b]] \\ &= E_{\tau\sim p_\theta(\tau)}[(g(\tau)[R(\tau)-b])^2] - \Big[E_{\tau\sim p_\theta(\tau)}[(g(\tau)[R(\tau)-b]]\Big] \\ &= E_{\tau\sim p_\theta(\tau)}[(g(\tau)[R(\tau)-b])^2] - \Big[E_{\tau\sim p_\theta(\tau)}[(g(\tau)R(\tau)]\Big]\end{aligned}$$

Clearly, the variance will be minimize if we choose b properly such that $E_{\tau\sim p_\theta(\tau)}[(g(\tau)[R(\tau)-b])^2]$ is minimized (which is just a weighted least square optimization problem). With straight forward calculus, we can see the minimizer b^* is given by

$$b^* = \frac{E_{\tau\sim p_\theta(\tau)}[g(\tau)R(\tau)]}{E_{\tau\sim p_\theta(\tau)}[g(\tau)]}.$$

In practice, we simply choose $b = E_{\tau\sim p_\theta(\tau)}[R(\tau)]$ as the baseline function.

In the REINFORCE type of batch learning algorithms, we can estimate gradient with baseline by

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i)[R(\tau_i)-b], b = \frac{1}{N}\sum_{i=1}^{N} R(\tau).$$

where b can be view as an estimate of $E_{\tau\sim p_\theta(\tau)}[R(\tau)]$ and thus is a function of θ instead of τ.

In the Actor-Critic type of algorithms, one common choice of baseline is a state-value function estimate $\hat{V}^\pi(s)$, which can be approximated by Critic. Then the policy gradient becomes

$$\nabla_\theta J(\theta) = E_{s\sim\rho(s),a\in\pi_\theta(s)}[(Q^\pi(s,a) - V^\pi(s))\nabla_\theta \ln \pi(a|s,\theta)]$$

Notably, we usually define an **Advantage function** associated with policy π by

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s),$$

where $A^\pi(s,a)$ characterize the advantage of choosing a at state s over choose $\pi(a|s)$ at state s. That is the $A^\pi(s,a)$ provides the direction to update policy π.

An Actor-Critic type of algorithm with the advantage function is showed in algorithm 77, where $\hat{V}(s;w)$ will also be updated based on least-squared methods.

Remark 37.3.4 (more intuitive interpretation). To understand why subtracting mean values (i.e., state value function as the baseline) can help learning more effectively, we look at the policy gradient step that aims to increase probability of actions that receive positive action and reduce probability of actions that receive negative reward. For control tasks that involves positive rewards and negative rewards (as a form of penalty), the gradient step makes sense. However, if all rewards received are positive, but more larger than the rest, then the gradient step will promote both actions, although some at a smaller extent. By subtracting the mean, the gradient step will explicitly make actions receiving larger reward more frequently and action receiving smaller rewards less frequently.

Lemma 37.3.4 (log-derivative trick). *Let x be a random variable following a θ parameterized distribution given by p_θ. We have*

- $$\nabla_\theta E[f(x)] = E[f(x)\nabla_\theta \log p_\theta(x)].$$
- $$E[\nabla_\theta \log p_\theta(x)] = 0.$$

Proof. (1)

$$\begin{aligned}\nabla_\theta E[f(x)] &= \nabla_\theta \int p_\theta(x) f(x) dx \\ &= \int \frac{p_\theta(x)}{p_\theta(x)} \nabla_\theta p_\theta(x) f(x) dx \\ &= \int p_\theta(x) \nabla_\theta \log p_\theta(x) f(x) dx \\ &= E\left[f(x)\nabla_\theta \log p_\theta(x)\right]\end{aligned}$$

(2) Note that $f(x) = 1$. □

One common choice of baseline is a state-value function estimate $\hat{V}(s)$. Then the policy gradient becomes

$$\nabla_\theta J(\theta) = E_{s\sim\rho(s), a\in\pi_\theta(s)}[(Q^\pi(s,a) - V^\pi(s))\nabla_\theta \ln \pi(a|s,\theta)]$$

Notably, we usually define an **Advantage function** associated with policy π by

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s),$$

where $A^\pi(s,a)$ characterize the advantage of choosing a at state s over choose $\pi(a|s)$ at state s. That is the $A^\pi(s,a)$ provides the direction to update policy π.

An Actor-Critic type of algorithm with the advantage function is showed in algorithm 77, where $\hat{V}(s;w)$ will also be updated based on least-squared methods.

Algorithm 77: Policy-gradient method with a value function baseline

Input: A policy differential function parameterized $\pi(a|s,\theta)$, a differentiale state-value parameterization $\hat{V}(s,w)$, learning rate $\alpha_1, \alpha_2 > 0$.

Output: Improved policy $\pi(a|s,\theta^*)$ parameterized by θ^*

Initialize policy parameter $\theta \in \mathbb{R}^d$

repeat

  Using current policy $\pi(\cdot|\cdot,\theta)$ to generate an episode $(s_0,a_0,r_1),...,(s_{T-1},a_{T-1},r_T)$.

  for *each step in the episode* $t \in \{1,...,T\}$ **do**

    Calculate reward-to-go $G_t = \sum_{n=t}^{T} \gamma^{n-t} r_t$. Note that depending on if the task is infinite horizon process, we need to add $\hat{V}(s_T)$ to G_t is s_T is not terminal.

    Calculate advantage function $\delta = G_t - \hat{V}(s_t,w)$

    Perform gradient descent on policy π_θ

$$\theta = \theta + \alpha\gamma^t \delta \nabla_\theta \ln \pi(a_t|s_t,\theta).$$

    Perform a gradient descent step to improve value function esimation $w = w + \alpha_2(R_t - \hat{V}(s_t;w))\delta\nabla_w(r_t - \hat{V}(s_t;w))$

  end

until *sufficient iterations;*

return V

37.3.2.2 *Generalized advantage estimation*

In subsubsection 37.3.2.1, we establish that we can obtain a reduced-variance but biased policy gradient via

$$\nabla_\theta J(\theta) = E_{s\sim\rho(s),a\in\pi_\theta(s)}[(Q^\pi(s,a) - V^\pi(s))\nabla_\theta \ln \pi(a|s,\theta)].$$

In turns out we can reduce the variance and bias even more by improving the estimation accuracy of A via n step estimation using a method similar to TD(n) method [subsection 37.2.4] and by taking averages[5].

We first have the n step estimates for advantage function, given by

$$
\begin{aligned}
\hat{A}_t^{(1)} &\triangleq \delta_t^V \quad = -V(s_t) + r_t + \gamma V(s_{t+1}) \\
\hat{A}_t^{(2)} &\triangleq \delta_t^V + \gamma\delta_{t+1}^V \quad = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
\hat{A}_t^{(3)} &\triangleq \delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) \\
\hat{A}_t^{(k)} &\triangleq \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \\
\hat{A}_t^{(\infty)} &\triangleq \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V = -V(s_t) + \sum_{l=0}^{\infty} \gamma^l r_{t+l}
\end{aligned}
$$

Our way to reduce variance is by taking averages. ... proposed generalized advantage estimate to be the exponential average of n step advantage estimates, given by

$$
\begin{aligned}
\hat{A}_t^{GAE} &= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \ldots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda\left(\delta_t^V + \gamma\delta_{t+1}^V\right) + \lambda^2\left(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V\right) + \ldots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda\left(\delta_t^V + \gamma\delta_{t+1}^V\right) + \lambda^2\left(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V\right) + \cdots\right) \\
&= (1-\lambda)\left(\delta_t^V\left(1 + \lambda + \lambda^2 + \ldots\right) + \gamma\delta_{t+1}^V\left(\lambda + \lambda^2 + \lambda^2 + \ldots\right)\right. \\
&\quad \left. + \gamma^2\delta_{t+2}^V\left(\lambda^2 + \lambda^3 + \lambda^4 + \ldots\right) + \ldots\right) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right) + \ldots\right) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V
\end{aligned}
$$

37.3.2.3 *Summary of stochastic gradient descent forms*

The policy gradient can be written by

$$g = E[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)],$$

where Ψ_t can take following forms [5]:

- total reward of the trajectory

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

- total reward following action a_t

$$\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'}.$$

- total reward following action a_t with baseline

$$\sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} - b(s_t).$$

- Q function

$$Q^{\pi}(s_t, a_t).$$

- Advantage function

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t).$$

- TD residual

$$r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t).$$

37.3.3 Deterministic policy gradient

Suppose now the control policy π is a θ parameterized deterministic policy given by μ_θ[1]. And we further assume the stationary distribution of the state s under such policy is given by $\rho^{\mu}(s)$. The objective function we are trying to maximize now becomes

$$J(\mu_\theta) = \int_S \rho^{\mu}(s) Q\left(s, \mu_\theta(s)\right) ds.$$

The gradient of this objective function is given by the following theorem.

Theorem 37.3.3 (deterministic policy gradient theorem). *[6] If $\nabla_\theta \mu_\theta(s)$ and $\nabla_a Q^{\mu}(s,a)$ exist, then*

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_S \rho^{\mu}(s) \nabla_a Q^{\mu}(s,a) \nabla_\theta \mu_\theta(s) \Big|_{a=\mu_\theta(s)} ds \\ &= E_{s\sim\rho^{\mu}}[\nabla_a Q^{\mu}(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}] \end{aligned}$$

Remark 37.3.5 (interpretation).

1 In the stochastic policy gradient setting, the action function is a parameterized probability distribution over the action space by $\pi_\theta(a|s)$.

- As a comparison, the gradient of the objective function in stochastic policy setting is given by

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \mathrm{d}a \mathrm{d}s \\ &= E_{s\sim\rho^\pi, a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)\right]\end{aligned}$$

  which integrate over both the action and the state space. In deterministic policy gradient, only state space is being integrated over.
- Computing the stochastic policy gradient may require more samples, especially if the action space has many dimensions. Therefore, using deterministic policy can potentially be faster.

Because the policy is deterministic, another attractive feature in the gradient is that we will not need to adjust the importance weight as we do in stochastic policy gradient. More formally,

$$\begin{aligned}\nabla_\theta J_\beta(\mu_\theta) &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \mu_\theta(a|s) Q^\mu(s,a) \mathrm{d}s \\ &= E_{s\sim\rho^\beta}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right]\end{aligned}$$

where β, the policy used to sample the data, is different from current policy μ_θ we optimize over.

37.4 Algorithms zoo

37.4.1 Neural Fitted Q Iteration (NFQ)

Value functions are the cornerstones of reinforcement learning. In small problems, value functions are represented by a table. When tackling real-world problems involving large action and state space dimensionality, one faces the problem of an appropriate, efficient method to represent value functions. In NFQ[7], artificial neural networks, thanks to their universal approximation capability, are used to represent value functions [Figure 37.4.1]. Another key idea underlying NFQ is updating the neural value function off-line considering an entire set of transition experiences.Updating network on-line with non-linear value function approximators can lead to instability and divergence in the training. Updating offline allows the application of advanced supervised learning methods, that enables faster and reliable convergence than online gradient descent methods.

The full algorithm is showed in algorithm 78.

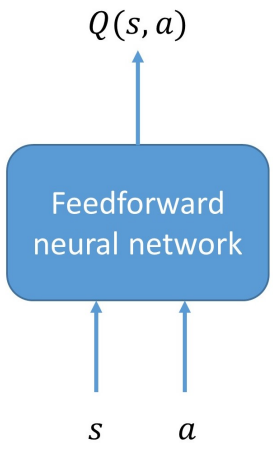


Figure 37.4.1: A typical feed-forward neural network used in NFQ to approximate the Q function.

Algorithm 78: Neural Fitted Q Iteration (NFQ)

Input: A set of transition sample $D = \{(s, a, s', r)\}$, step to learn N

Initialize network Q_0.

for $n = 1,...,N$ **do**

  Generate pattern set $P = (input, output)$, where

$$input = (s, u)$$
$$target = r + \gamma \max_b Q_n(s, b)$$

  Stochastic gradient descent traing Q network and get Q_{n+1}

end

Output: Q value function

37.4.2 Canonical deep Q learning

In the DQN framework[8], we use deep neutral network approximate the optimal action-value function

$$Q^*(s, a) = \max_\pi E\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi\right],$$

which is maximizing accumulated discounted rewards in the process over all possible policy π after achieving state s and taking an action a, where r_t is the one step reward function, γ is the discount factor. With the optimal $Q*$, the optimal action at state s is given by $a^* = \arg\max_a Q^*(s, a)$.

Alternatively, using π^* to denote the optimal policy $\pi^*(s) = \arg\max_a Q^*(s, a)$, we can re-write optimal Q function by

$$Q^*(s, a) = E\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi^*\right],$$

which is further connected to optimal state-value function by

$$V^*(s) = E\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, \pi^*\right] = \max_a Q^*(s, a).$$

A typical neural network for deep Q learning is showed in Figure 37.4.2, which takes a state or an observation, denoted by s, as the input, and outputs multiple values corresponding $Q(s, a)$. The number of outputs is the same number as the number of actions.

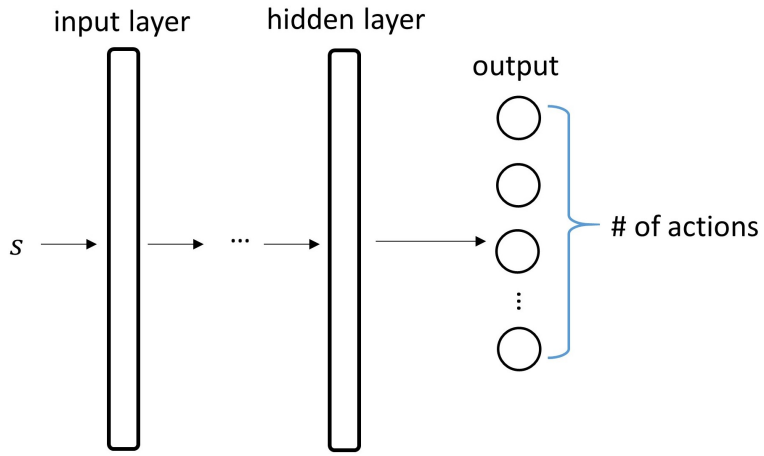


Figure 37.4.2: The network architecture for canonical deep Q learning. The network takes a state or an observation, denoted by s, as the input, and outputs multiple values corresponding $Q(s, a)$. The number of outputs equals to the number of actions.

The core of the training procedure is gradient decent. The authors in [8] proposed multiple improvements for stability and convergence:

Remark 37.4.1 (procedures to improve the stability of training).

- Before deep Q learning, target value used to train Q network was

$$r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta),$$

  which leads to unstable training process. In deep Q learning, the target use is generated by a target network Q', whose parameters are slowly updated, given by,

$$r_j + \gamma \max_{a'} Q'(s_{j+1}, a'; \theta^-).$$

- Before deep Q learning, directly using consecutively generated samples also cause instability in the training process. This is because these samples are correlated and correlated samples applied to stochastic gradient method can lead to divergence. By using a replay buffer to randomly sample experience to learn, we decorrelate the sample and reuse samples.
- A helpful trick is to clip the error term

$$y - Q(s, a; \theta)$$

  between $[-1, 1]$ or other more proper intervals. Such clipping corresponding to a more robust error function, like Huber loss function.

The final training algorithm is showed in algorithm 79.

Algorithm 79: Deep Q-learning with experience replay

Initialize replay memory $\mathcal{B}$ to capacity N.
Initialize action-value function Q with random weights θ.
Initialize target action-value function Q' with weights $\theta^- = \theta$.
for *episode* = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$.
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t, otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$.
    Execute action a_t in emulator and observe reward r_t and state x_{t+1}.
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess s_{t+1}.
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$.
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{B}$.
    Set

$$y_j = \begin{cases} r_j, \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, a'; \theta^-), \text{otherwise} \end{cases}$$

    Every C steps reset $Q' = Q$.
  end
end

37.4.3 DQN variants

37.4.3.1 *Overview*

Although Deep Q learning algorithm has achieved remarkable performance in various challenging benchmark tasks, there has been extensive discussion on its possible weaknesses.

For example,

- The Q value is calculated using bootstrapped approach. Is such method accurate or biased?
- Is random sampling of experience is best option? Some experiences might be more important in contributing to the learning processes.
- Q network is directly learning state-action function. Will it be better to learn state value and action value separately?
- Is there any better approach to explore the state space beside ϵ greedy approach?

- The network structure is not suitable to handle tasks with long memory state dynamics. How to equip Q network with RNN structure?
- Deep Q learning usually takes a long time to converge.

Since the invention of the Deep Q learning algorithm, there have been various approaches to improve the canonical Deep Q learning algorithm.

37.4.3.2 *Double Q learning*

In the canonical deep Q learning, the target of the Q network is given by

$$y_t^Q = r_{t+1} + \gamma \max_{a'} Q'(\phi_{j+1}, a'; \theta^-),$$

the maximization operator can propagate noise and overestimate Q values. In double Q-learning ([9]), we use two networks to generate the target value

$$y_t^{DoubleQ} = r_{t+1} + \gamma Q'(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t')$$

where Q' is used to generate target and Q is used to generate action. Because Q and Q' are not fully correlated, such method reduce the tendency of overestimation.

37.4.3.3 *Dueling network*

Given the Q function and the state value function V associated with a policy π, we can define the **advantage function** by

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s).$$

The advantage function shows the advantage of selecting an action with respect to the selection given by policy π.

In DQN dueling architecture [10], as showed in Figure 37.4.3, we explicitly separates the representation of state values and state-dependent action advantages via two separate subnetworks and synthesize the final Q function via $Q^\pi = V^\pi + A^\pi$.

The main advantage of explicitly separating two estimators is that the network can be focused on learning state functions without having to learn the state-action value Q function first. After all, the agent are learning to choose better action leading to states with higher values.

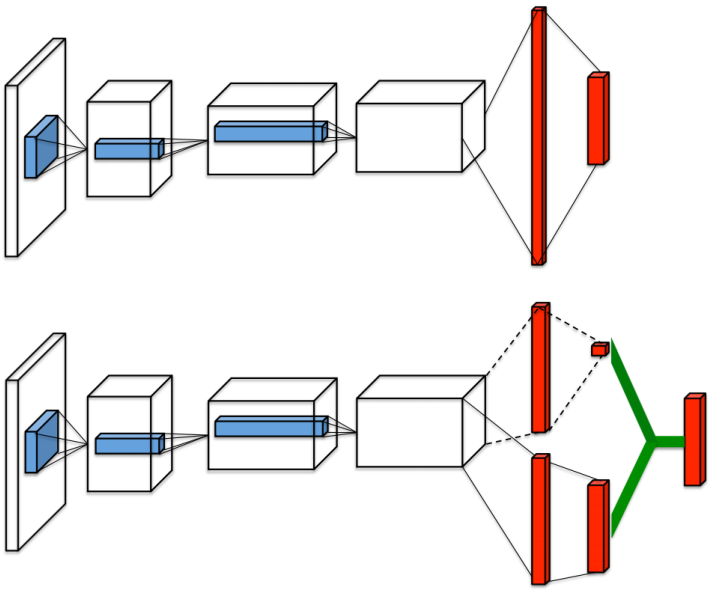


Figure 37.4.3: A typical single stream Q-network (top) in deep Q learning and a dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value and the advantage function for each action; the green output module synthesize the Q value from two streams. Both networks output Q-values for each action. [10].

37.4.3.4 *Deep Recurrent Q network (DRQN)*

The main idea in DRQN[11] is to incorporate recurrent modules into learn representations from both current and past observations, as a attempt for Markov decision process with partial observability. Many sequential decision problems requires high-dimensional observation from the past, yet classical DQN mostly optimize decision-making using current and immediate past observation. An example architecture will something like Figure 37.4.4, where sequence of past observation are fed into a neural network with recurrent layers to learn the temporal relationship between observation or paint the full pictures from relatively local observations.

Notably, the usage of recurrent layers requires the input to be sequences, we usually sample episodes from replay memory and update the network from the episode start or a random start point of the episode and proceed towards the conclusion of the episode.

37.4.3.5 *Asynchronous Methods*

Deep Q learning relies on experience replay to reduce the correlation between samples and stabilize the training process. An alternative paradigm for deep reinforcement

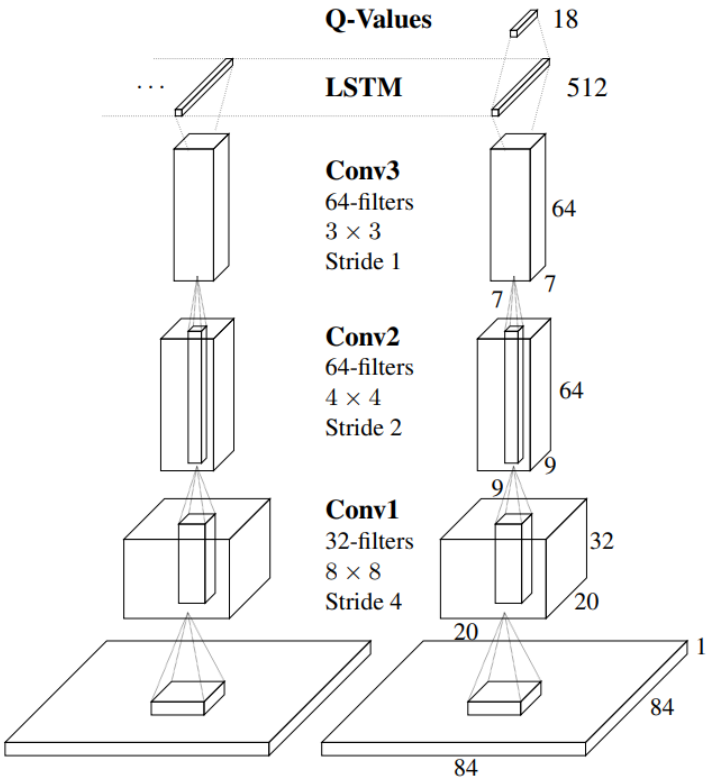


Figure 37.4.4: In a DQRN, recurrent layers are usually placed on the last layer before output. Unlike canonical DQN, we need to feed sequence of observation into the network and finally output Q values. Above scheme only unfolds to two steps.[11]

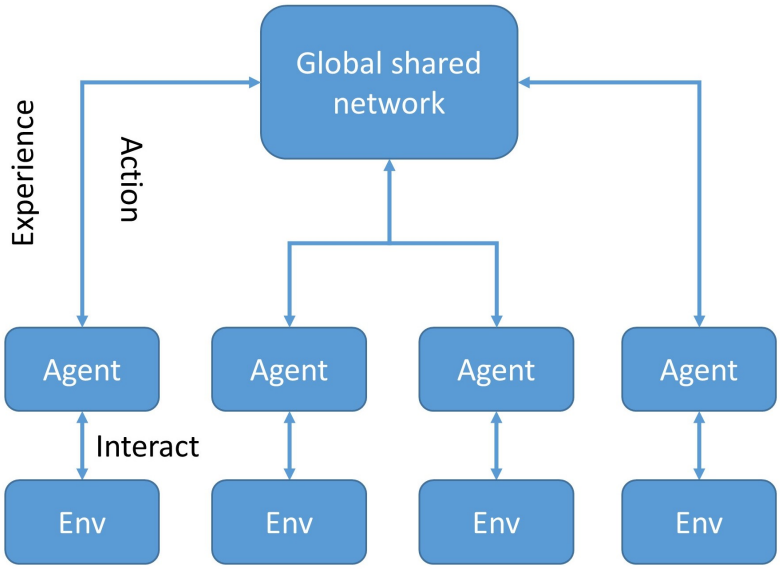


Figure 37.4.5: An example architecture that implements the asynchronous reinforcement learning paradigm. Multiple agents interact with multiple instances of environments in parallel. Agents collect experiences to train a globally shared network that learns control policies.

learning to asynchronously use multiple agents to in parallel interact with on multiple instances of the environment[12], as showed in Figure 37.4.5. This parallelism offer several critical benefits:

- Since agents are running different exploration policies at different state sub-regions, experiences are less likely to be correlated. Experiences collected online can be used for on-policy learning.
- For some DQN learning where environment simulation is prohibitable, it speeds up training processes by generating experiences in parallel.

37.4.4 Universal value function approximator

Many relatively simple, primitive sequential decision-making tasks can be abstractly viewed as taking actions to move from arbitrary starting points in a high-dimensional state to one prescribed target point. More complex, sophisticated tasks can be solved by decomposing them to a series of simpler tasks. For example, a long distance robot navigation task can be decomposed into a series of short-ranged navigation tasks to a set of goals leading to the ultimate goal. To enable such decomposition, it is necessary to learn

Algorithm 80: Asynchronous Deep Q-Learning for each thread

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T

For each thread, initialize thresh step counter $t = 0$
Initialize target network parameters $\theta' = \theta$
Get initial state s.
repeat
  With probability ϵ select a random action a_t, otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$.
  Execute action a_t and observe reward r_t and state s_{t+1}.
  Set $s_{t+1} = s_t, a_t, s_{t+1}$.
  Set

$$y_j = \begin{cases} r_j, \text{if episode terminates at step} j+1 \\ r_j + \gamma \arg\max_a Q(s_{j+1}, a^*; \theta'), \text{otherwise} \end{cases}$$

  Every C steps reset $Q' = Q$.
until *episode = 1, M;*

goal-parameterized policies. In the single goal setting, control policies are encoded by state value function V or the Q function. Similarly, we can encode goal-parameterized policies by goal-parameterized value function or Q function, known as universal value function. In canonical deep Q learning, we use a neural network to learn the control policy and the value function for a single goal [Figure 37.4.6]; in universal value function approximator, we use goals as the inputs to the neural network, which the aim to learn control policies and value function parameterized by goals, and more importantly generalizing to unseen goals.

From a broader perspective, universal value function approximator is closer to general intelligence, in which the intelligent agent can accomplish tasks of similar goals without relearning.

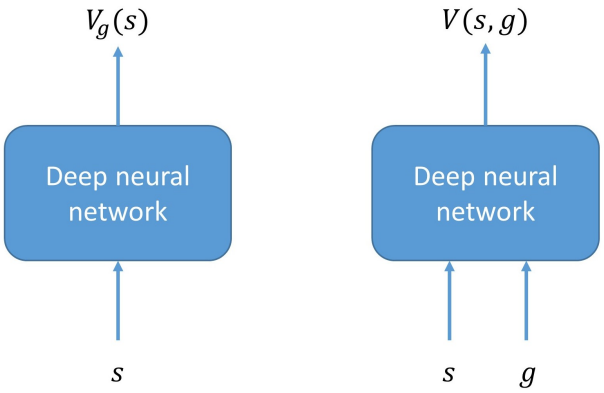


Figure 37.4.6: A comparison between a typical deep Q learning network (left) and a typical universal value function approximator network (right).

A modified DQN algorithm with universal value approximator is given by algorithm 81.

Algorithm 81: Deep Q-learning with universal value function approximator

Initialize replay memory $\mathcal{B}$ to capacity N.
Initialize action-value function Q with random weights θ.
Initialize target action-value function Q' with weights $\theta^- = \theta$.
for *episode* = 1, M **do**
  Initialize state s_1.
  Sample a goal from G. **for** $t = 1, T$ **do**
    With probability ϵ select a random action a_t, otherwise select $a_t = \arg\max_a Q(s_t, g, a; \theta)$.
    Execute action a_t in the simulator and observe reward r_t and state s_{t+1}.
    Store transition $(s_t, a_t, r_t, s_{t+1}, g)$ in $\mathcal{B}$.
    Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1}, g)$ from $\mathcal{B}$.
    Set

$$y_j = \begin{cases} r_j, \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, a', g; \theta^-), \text{otherwise} \end{cases}$$

    Every C steps reset $Q' = Q$.
  end
end

37.4.5 Deep deterministic policy gradient (DDPG) algorithm

Deep Deterministic Policy Gradient (DDPG) is one of the most important result the deterministic policy gradient theorem. DDPG employs an actor-critic network architecture [Figure 37.4.7], but there are couple of differences between DDPG and an ordinary actor-critic algorithm:

- Ordinary actor-critic algorithms will use stochastic policy, where DDPG uses deterministic policy.
- The critic in an ordinary actor-critic algorithm learns the value function, while the critic in DDPG learns the Q function.

DDPG further combines characteristics of DQN such as off-line learning and experience replay. Because it applies to decision-making problems with continuous action space, it greatly expand the application regime of deep Q learning. By contrast, DQN applies only to decision-making problems with discrete action space or small continuous action space that can be properly discretized.

In DDPG, the Q network update is based on Bellman principle and is similar to that in DQN. The actor network learns a deterministic control policy function μ, based on gradients given by

$$\begin{aligned}\nabla_\theta J(\theta) &= \int_S \rho^\mu(s) \nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s) \Big|_{a=\mu_\theta(s)} ds \\ &= E_{s\sim\rho^\mu}[\nabla_a Q^\mu(s,a) \nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}]\end{aligned}$$

Similar to DQN, DDPG employs experience replay to increase sample efficiency and target networks to stabilize the training process. Additionally, DDPG employs an OU process for exploration, analogous to the ϵ greedy rule in DQN. One critical difference between DDPG and DQN is the how to improve policy based on estimated Q.

Remark 37.4.2 (policy improvement in DDPG).

- In DQN and many other Q learning algorithms, policy improvement is achieved by taken actions via $\pi(s) = \arg\max_a Q(s,a)$. But the maximizing operation is difficult in continuously action space. Instead, DDPG relies on deterministic policy gradient and take gradient steps to improve parameterized policies.

The complete algorithm is showed in algorithm 82.

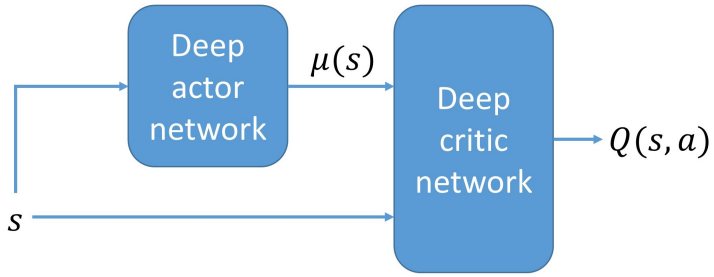


Figure 37.4.7: A typical deep neural network architecture for DDPG reinforcement learning. The actor network outputs control policy; the critic network outputs estimate of Q function. Through back-prorogation, the critic network improves estimation accuracy and the actor network improves policy.

Algorithm 82: Deep deterministic policy gradient algorithm (DDPG)

Input: A randomly initialized critic network $Q(s, a|\theta^Q)$ and an actor network $\mu(s|\theta^\mu)$ with weights θ^Q, θ^μ.

Output: Optimal critic and actor networks

Initialize target networks Q' and μ' with weights $\theta^{Q'} = \theta^Q$ and $\theta^{\mu'} = \theta^\mu$.

Initialize replay buffer $\mathcal{B}$. Initialize policy parameter $\theta \in \mathbb{R}^d, w \in \mathbb{R}^m$.

repeat

  Initialize a random process $\mathcal{N}$ for action exploration.

  Initialize initial state s_1.

  for $t = 1, T$ **do**

    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to current actor network and exploration noise.

    Execute action a_t and observe reward r_t and new state s_{t+1}.

    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$.

    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{B}$.

    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.

    Update the critic network by minimizing the loss

$$L = \frac{1}{N}\sum_i^N (y_i - Q(s_i, a_i|\theta^Q))^2.$$

    Update the actor policy network using the sampled policy graident:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{i=1}^N \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}.$$

    Update the target network

$$\theta^{Q'} = \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} = \tau\theta^Q + (1-\tau)\theta^{\mu'}$$

  end

until *sufficient iterations*;

37.4.6 Twin-delayed deep deterministic policy gradient (TD3)

Q-learning algorithms have been found to overestimate of the value function[9]. This overestimation can propagate through the training iterations and lead to inferior policies.

In DQN learning for discrete control, this issue is remedied by using Double Q-learning [9].

Twin Delayed Deep Deterministic policy gradient (short for TD3) [13] employs a couple of techniques to address the value function overestimation in DDPG.

- Clipped Double Q-learning. We uses the minimum estimation among two independent network estimates as the target. This can favor underestimation bias which is hard to propagate through training:

$$y = r + \gamma \min_{i=1,2} Q_{w_i} \left(s', \mu_{\theta_1} \left(s' \right) \right)$$

- Delayed update of Policy Networks. By updating policy network slower than the value network, we can possibly reduce the detrimental effect of value overestimation when the policy is poor. Value overestimation of a poor policy can lead to the learning of even poorer polices.
- Target Policy Smoothing. Note that deterministic policies can overfit to spiking peaks in the value function. If the actor network exploits too much the spikes in the value network, it can cause divergence or inferior policies. TD3 add noise to the selected action to make it harder for exploitation of spikes in learned value function. This approach mimics the idea of SARSA update and enforces the intuition that similar actions should have similar values.

The final algorithm is given by algorithm 83.

Algorithm 83: Twin-delayed deep deterministic policy gradient (TD3)

Input: A randomly initialized two critic networks $Q(s, a|\theta_i^Q)$ and one actor network $\mu(s|\theta_i^\mu)$ with weights $\theta_i^Q, \theta_i^\mu, i = 1, 2$

Output: Optimal critic and actor networks

Initialize two target Q networks Q' and a one target actor network μ' with weights $\theta_i^{Q'} = \theta^Q$ and $\theta^{\mu'} = \theta^\mu, i = 1, 2$.

Initialize replay buffer $\mathcal{B}$. Initialize policy parameter $\theta \in \mathbb{R}^d, w \in \mathbb{R}^m$

repeat

  Initialize a random process $\mathcal{N}$ for action exploration.

  Initialize initial state s_1.

  for $t = 1, T$ **do**

    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to current actor network and exploration noise.

    Execute action a_t and observe reward r_t and new state s_{t+1}.

    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$.

    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{B}$.

    Compute actions usng target actor network for state s_{i+1}, add noise and probably clip it to get a'.

    Set $y_i = r_i + \gamma \min_{i=1,2} Q'(s_{i+1}, a'|\theta^{Q'})$.

    Update the critic network by minimizing the loss

$$L = \frac{1}{N} \sum_i^N (y_i - Q(s_i, a_i|\theta^Q))^2.$$

    Update the actor policy network **less frequently** using the sampled policy graident:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_{i=1}^N \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}.$$

    Update the target network

$$\theta_i^{Q'} = \tau\theta^Q + (1-\tau)\theta^{Q'_i}, i = 1, 2$$

$$\theta^{\mu'} = \tau\theta^Q + (1-\tau)\theta^{\mu'}$$

  end

until *sufficient iterations;*

37.4.7 Trust Region Policy Optimization (TRPO)

37.4.7.1 *TRPO*

TRPO [14] is an improved policy gradient algorithm that aims to achieve stable and robust improvement on each gradient step. Its key idea is to introduce a KL divergence based constraint on the gradient step size at each iteration.

Suppose now we are optimize the policy parameter θ based on its previous iterate θ_k. The gradient of the objective function is the same as the gradient following objective function(also see subsubsection 37.3.2.3):

$$J(\theta) = \underset{s,a\sim\pi_{\theta_k}}{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a)\right].$$

In seeking the next iterate θ_k that can improve $J(\theta)$, we use following optimization formulation:

$$\theta_{k+1} = \arg\min_\theta L(\theta_k,\theta), \quad s.t., \overline{D}_{KL}(\theta||\theta_k) \leq \delta,$$

where $L(\theta_k,\theta)$ is given by

$$L\left(\theta_k,\theta\right) = \underset{s,a\sim\pi_{\theta_k}}{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a)\right],$$

and $D_{KL}(\theta||\theta_k)$ is an **average** KL-divergence between the two policies averaging over all states, given by

$$\overline{D}_{KL}\left(\theta\|\theta_k\right) = \underset{s\sim\pi_{\theta_k}}{E}\left[D_{KL}\left(\pi_\theta(\cdot|s)\|\pi_{\theta_k}(\cdot|s)\right)\right]$$

The theoretical TRPO update quite complicated to work with; instead, TRPO uses Taylor expand the objective and constraint to leading order around θ_k

$$\begin{aligned} L\left(\theta_k,\theta\right) &\approx g_k^T\left(\theta-\theta_k\right) \\ \overline{D}_{KL}\left(\theta\|\theta_k\right) &\approx \frac{1}{2}\left(\theta-\theta_k\right)^T H_k\left(\theta-\theta_k\right) \end{aligned}$$

where g_k and H_k are the gradient and Hessian at θ_k. which gives the following approximate optimization problem

$$\theta_{k+1} = \arg\max_\theta g_k^T(\theta-\theta_k), \quad s.t., \frac{1}{2}(\theta-\theta_k)^T H_k(\theta-\theta_k) \leq \delta.$$

This approximate optimization problem has an iterative solution [2]given by

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g_k^T H_k^{-1} g_k}} H_k^{-1} g_k.$$

A problem is that, due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint, or actually improve the surrogate advantage. TRPO uses backtracking line search,

$$\theta_{k+1} = \theta_k + \alpha \sqrt{\frac{2\delta}{g_k^T H_k^{-1} g_k}} H_k^{-1} g_k$$

such that $\alpha \in (0, 1)$ is chosen to ensure $\pi_{\theta_{k+1}}$ to satisfy the KL constraint and produce a positive surrogate advantage.

2 To see this, the constraint such be active since objective function is linear.

Finally, the gradient g_k can be obtained via automatic differentiation, the Hessian H_k via Fisher information matrix [subsubsection 37.4.7.2], and the $H^{-1}g$ can be evaluated by coordinate descent without explicitly evaluating H^{-1}.

Algorithm 84: Trust Region Policy Optimization

Input: A randomly initialized policy parameterθ_0, value function parameter ϕ_0, KL-divergence threshold δ.

Output: Optimal critic and actor networks

Initialize iteration count $k = 0$.

repeat

Collect a set of trajectories $D = \{\tau_1, ...\}$ by carrying out policy π_{θ_k} and compute their rewards-to-go for each state.

Compute advantage estimate $\hat{A}_t$ based on the current value function V_{ϕ_k}.

Estimate policy gradient by

$$\hat{g}_k = \frac{1}{|\mathcal{D}|} \sum_{\tau \in D_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t | s_t\right) \Big|_{\theta_k} \hat{A}_t$$

Update policy by

$$\theta_{k+1} = \theta_k + \alpha \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g.$$

Estimate value function via least-squared minimization of

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D|\, T} \sum_{\tau \in D} \sum_{t=0}^{T} \left(V_\phi\left(s_t\right) - \hat{R}_t\right)^2$$

until *sufficient iterations*;

37.4.7.2 *Evaluating Hessian of KL-divergence*

Based on the following Lemma 37.4.1, the Hessian of KL divergence is equivalent to Fisher matrix.

Note that given a θ parameterized distribution $p(x|\theta)$, its Fisher information matrix is given by

$$F = \underset{p(x|\theta)}{E} \left[\nabla \log p(x|\theta) \nabla \log p(x|\theta)^T\right].$$

With data samples $\{x_1, x_2, ..., x_N\}$, F can be evaluated empirically via

$$F = \frac{1}{N} \sum_{i=1}^{N} \nabla \log p\left(x_i|\theta\right) \nabla \log p\left(x_i|\theta\right)^T$$

Lemma 37.4.1 (Fisher matrix and Hessian of KL divergence). *Consider tow distribution $p(x|\theta)$ and $p(x|\theta')$ parameterized by θ and θ' vectors. The Fisher information for $p(x|\theta)$ with respect to vector θ is given by*

$$F = \underset{p(x|\theta)}{E}\left[\nabla \log p(x|\theta) \nabla \log p(x|\theta)^T\right]$$

Fisher information matrix F is the Hessian of KL divergence between two distribution , with respect to θ', evaluated at $\theta' = \theta$.

Proof. Based on the definition of KL divergence, we can write

$$KL\left[p(x|\theta) \| p\left(x|\theta'\right)\right] = \underset{p(x|\theta)}{E}[\log p(x|\theta)] - \underset{p(x|\theta)}{E}\left[\log p\left(x|\theta'\right)\right]$$

where the first term is entropy and the second term is cross-entropy.

$$\begin{aligned}
\nabla_{\theta'} KL\left[p(x|\theta) \| p\left(x|\theta'\right)\right] &= \nabla_{\theta'} E_{p(x|\theta)}[\log p(x|\theta)] - \nabla_{\theta} \lim_{p(x|\theta)}\left[\log p\left(x|\theta'\right)\right] \\
&= -E_{p(x|\theta)}\left[\nabla_{\theta} \log p\left(x|\theta'\right)\right] \\
&= -\int p(x|\theta) \nabla_{\theta'} \log p\left(x|\theta'\right) dx
\end{aligned}$$

The second derivative is

$$\nabla_{\theta}^2 KL\left[p(x|\theta) \| p\left(x|\theta'\right)\right] = -\int p(x|\theta) \nabla_{\theta}^2 \log p\left(x|\theta'\right) dx$$

The Hessian with respect to θ' evaluated at $\theta' = \theta$ is

$$H = -\int p(x|\theta) \nabla_{\theta}^2 \log p\left(x|\theta'\right)\Big|_{\theta=\theta} \mathrm{d}x = F$$

where we used the basic property of Fisher Information matrix [Theorem 13.2.1] that the Fisher information is the negative Hessian. □

37.4.8 Proximal Policy Optimization (PPO)

PPO simplifies the gradient step size regularization in TRPO by using a modified surrogate objective function. Instead of optimizing an KL related constraint in TRPO, a typical PPO formulation updates policies via

$$\theta_{k+1} = \arg\max_{\theta} E_{s,a\sim\pi_{\theta_k}}\left[L\left(s,a,\theta_k,\theta\right)\right]$$

where L is given by

$$L\left(s,a,\theta_k,\theta\right) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right)A^{\pi_{\theta_k}(s,a)}\right)$$

in which ϵ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old.

Although the objective function seems complicated at the first glance, it aims for two purposes similar to TRPO: first, it helps maintain the new policy close to the old policy; second, the new policy is better the old policy.

The simplified form has more straight forward interpretation:

Remark 37.4.3 (interpretation).

- If the advantage $A(s,a) > 0$ for (s,a), then its contribution to the objective reduces to

  $$L\left(s,a,\theta_k,\theta\right) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon)\right)A^{\pi_{\theta_k}}(s,a)$$

  Because $A(s,a) > 0$, increasing L will make a more likely, that is, it will make $\pi_\theta(a|s)$ increase.
- If the advantage $A(s,a) < 0$ for (s,a), then its contribution to the objective reduces to

  $$L\left(s,a,\theta_k,\theta\right) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right)A^{\pi_{\theta_k}}(s,a)$$

  Because $A(s,a) < 0$, increasing L will make a less likely, that is, it will make $\pi_\theta(a|s)$ decrease.
- The min operator limits how much the objective can increase. For example, if $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$, the min will put a ceiling of $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$.

Combination the valuation function estimation similar in TRPO, we have an example PPO algorithm in algorithm 85.

Algorithm 85: Proximal Policy Optimization

Input: A randomly initialized policy parameterθ_0, value function parameter ϕ_0, KL-divergence threshold δ.

Output: Optimal critic and actor networks

Initialize iteration count $k = 0$.

repeat

Collect a set of trajectories $\mathcal{D} = \{\tau_1, ...\}$ by carrying out policy π_{θ_k} and compute their rewards-to-go for each state.

Compute advantage estimate $\hat{A}_t$ based on the current value function V_{ϕ_k}.

Estimate policy gradient via

$$\hat{g}_k = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(a_t|s_t\right) \Big|_{\theta_k} \hat{A}_t$$

Update policy by stochastic gradient on

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k| \, T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta \left(a_t|s_t\right)}{\pi_{\theta_k} \left(a_t|s_t\right)} A^{\pi_{\theta_k}} \left(s_t, a_t\right), clip \left(\epsilon, A^{\pi_{\theta_k}} \left(s_t, a_t\right)\right) \right)$$

Estimate value function via least-squared minimization of

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}| \, T} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \left(V_\phi \left(s_t\right) - \hat{R}_t\right)^2$$

Set $k = k + 1$.

until *sufficient iterations;*

37.4.9 Soft Actor-Critic(SAC)

37.4.9.1 *Entropy regulated reinforcement learning*

SAC algorithm [15] aims to extend DDPG to stochastic policies. Stochastic policies can often explore the action and state space better than deterministic policies. As SAC works on stochastic policies, SAC uses entropy penalty to learn diverse, more exploration stochastic policies.

The objective function SAC optimizes is the expected process reward plus entropy penalty. By adjusting the penalty strength, we can control the trade-off of exploration-exploitation, which eventually determines the quality of learned policies. Similar to TD3, SAC also contains additional techniques to address value estimation issue. The stochasticity in the action selection also alleviate the exploitation of poorly learned value function spikes.

We first start with a set of definitions involving entropy penalties.

Definition 37.4.1 (entropy regulated reinforcement learning).

- *The value function associated with a stochastic policy $\pi(\cdot|s)$ is defined by*

$$V^{\pi}(s) = E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R\left(s_{t+1},a_t\right)+\alpha H\left(\pi\left(\cdot|s_t\right)\right)\right)|s_0=s\right],$$

  where H is the entropy of distribution $\pi(\cdot|s)$ and $\alpha>0$ control the penalty magnitude.
- *The value function associated with a stochastic policy $\pi(\cdot|s)$ is defined by*

$$Q^{\pi}(s,a) = E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R\left(s_{t+1},a_t\right)+\alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi\left(\cdot|s_t\right)\right)|s_0=s,a_0=a\right].$$

  Note that Q function does not have entropy penalty at $t=0$ when action a is known.
- *The objective is seek a control policy π that maximize value function:*

$$\pi^* = \arg\max_{\pi} E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\left(R\left(s_{t+1},a_t\right)+\alpha H\left(\pi\left(\cdot|s_t\right)\right)\right)\right].$$

It is straight forward to see that V and Q are connected by

$$V^{\pi}(s) = E_{a\sim\pi}[Q^{\pi}(s,a)] + \alpha H(\pi(\cdot|s));$$

that is, V is the sum of action averaged Q and the entropy penalty.

To use TD method to learn Q or V, it is necessary to know the Bellman principle under the entropy penalty framework. We have

$$\begin{aligned} Q^{\pi}(s,a) &= \underset{\substack{s'\sim P\\ a'\sim\pi}}{E}\left[R\left(s,a,s'\right)+\gamma\left(Q^{\pi}\left(s',a'\right)+\alpha H\left(\pi\left(\cdot|s'\right)\right)\right)\right] \\ &= \underset{s'\sim P}{E}\left[R\left(s,a,s'\right)+\gamma V^{\pi}\left(s'\right)\right] \end{aligned}$$

37.4.9.2 *The SAC algorithm*

The SAC algorithm uses different neural networks to represent stochastic policy π_θ, Q_ϕ function, and V_ψ function.

The learning of V will use Q to provide the target and take gradient descent on loss function given by

$$J_V(\psi) = E_{s_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi \left(s_t \right) - E_{a_t \sim \pi_\phi} \left[Q_\theta \left(s_t, a_t \right) - \log \pi_\phi \left(a_t | s_t \right) \right] \right)^2 \right],$$

where $\mathcal{D}$ is the sample distribution. Further, we can use two twin Q functions, similar to TD3, to reduce the overestimation bias.

The learning of Q will use V to provide the target and take gradient descent on loss function given by

$$J_Q(\theta) = E_{(s_t, a_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta \left(s_t, a_t \right) - \hat{Q} \left(s_t, a_t \right) \right)^2 \right],$$

where

$$\hat{Q} \left(s_t, a_t \right) = r \left(s_t, a_t \right) + \gamma E_{s_{t+1} \sim p} \left[V_\psi \left(s_{t+1} \right) \right]$$

To facilitate the learning of policy π, we parameterize $a_t = f_\theta(\epsilon_t; s_t)$ via a neural network transformation, where ϵ_t is an input noise vector, sampled from a fixed distribution (e.g., multivariate Gaussian). We then improve the policy by taking gradient descent steps on the objective function

$$J_\pi(\phi) = E_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} \left[\log \pi_\phi \left(f_\phi \left(\epsilon_t; s_t \right) | s_t \right) - Q_\theta \left(s_t, f_\phi \left(\epsilon_t; s_t \right) \right) \right].$$

It can be showed [15]that the gradient for the policy is given by

$$\nabla_\theta J_\pi(\theta) = \nabla_\theta \log \pi_\theta \left(a_t | s_t \right) + \left(\nabla_{a_t} \log \pi_\theta \left(a_t | s_t \right) - \nabla_{a_t} Q \left(s_t, a_t \right) \right) \nabla_\theta f_\theta \left(\epsilon_t; s_t \right)$$

The complete algorithm is given in algorithm 86.

Algorithm 86: Soft Actor-Critic (SAC) policy optimization

Input: A randomly initialized policy parameterθ_0, Q function parameter ϕ_0, and value function parameter ψ_0.

Output: Optimal policy network and value networks.

Initialize iteration count $k = 0$.

repeat

  for $t = 1, T$ **do**

    Select action $a_t = \mu(s_t|\theta^\mu)$ according to current policy network.

    Execute action a_t and observe reward r_t and new state s_{t+1}.

    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$.

    Randomly sample a batch of transition B from $\mathcal{B}$.

    Estimate value function via least-squared minimization of

$$\frac{1}{|B|}\sum_{s\in B}\left(Q_{\phi,1}\left(s, a_\theta(s)\right) - \alpha \log \pi_\theta\left(a_\theta(s)|s\right)\right)$$

    where $y_v(s) = \min_{i=1,2} Q_{\phi_i}(s,a) - \alpha \log \pi_\theta(a|s)$.

    Estimate Q function via least-squared minimization of

$$y_q\left(r, s', d\right) = r + \gamma(1-d)V_{\psi_{\text{targ}}}\left(s'\right)$$

    Update policy by taking stochastic gradient ascent by

$$\frac{1}{|B|}\sum_{s\in B}\left(Q_{\phi,1}\left(s, a_\theta(s)\right) - \alpha \log \pi_\theta\left(a_\theta(s)|s\right)\right)$$

  end

until *sufficient iterations*;

37.4.10 Evolution strategies

Evolution Strategies (ES) is an iterative black-box heuristic search optimization methods proven successful in solving complex problems in many domains. It is inspired by natural evolution process and mathematically belongs to a type of stochastic optimization method. Given a parameter vector θ to be optimized, each iteration perturbations will be added to θ to yield different modified copies of θ. A fitness function $F(\theta)$ will be then used gauge the quality of each copy. Good copies are recombined in certain way to generate a new θ for next iteration. Recently, an ES type algorithm was developed in the

context of reinforcement learning[16]. θ is policy parameter, and the fitness value can be value functions associated with this policy π_θ or other proxy functions of similar nature.

More formally, let θ be a random vector drawn from a distribution parameterized by ψ. Our goal is to optimize ψ (instead of θ), such that fitness scores averaging from $\theta \sim p_\psi(\theta)$ can be maximize. In [16], gradient types of optimization is used and the gradient the expected fittness with respect to parameter ψ is given by

Lemma 37.4.2.

$$\nabla_\psi E_{\theta \sim p_\psi(\theta)}[F(\theta)] = E_{\theta \sim p_\psi(\theta)}[F(\theta)\nabla_\psi p_\psi(\theta)]$$

Proof.

$$\begin{aligned}
\nabla_\psi E_{\theta \sim p_\psi(\theta)}[F(\theta)] &= \nabla_\psi \int F(\theta) p_\psi(\theta) d\theta \\
&= \int F(\theta) \nabla_\psi p_\psi(\theta) d\theta \\
&= \int F(\theta) p_\psi(\theta) \nabla_\psi \log p_\psi(\theta) d\theta \\
&= E_{\theta \sim p_\psi(\theta)}[F(\theta)\nabla_\psi p_\psi(\theta)]
\end{aligned}$$

□

A commonly used parameterized distribution is isotropic multivariate Gaussian $MN(\mu, \sigma^2 I), \mu \in \mathbb{R}^d$, where ψ is $(\mu, sigma)$. Then we have

Corollary 37.4.0.1.

$$\nabla_\mu E_{\theta \sim p_\psi(\theta)}[F(\theta)] = E_{\theta \sim p_\psi(\theta)}[F(\theta)\frac{(\theta - \mu)}{\sigma}]$$

Proof. Note that

$$p_\psi(\theta) = \frac{1}{(2\pi)^{d/2}\sigma^d} \exp(-\frac{(\theta - \mu)^T(\theta - \mu)}{2\sigma^2}).$$

Taking gradient with respect to μ we will get the result.

□

The gradient can be estimated the procedures: first generate perturbed copies $\theta_i = \mu + \sigma\epsilon_i, \epsilon \sim MN(0, I), i = 1, 2, ..., N$, then the estimate for the gradient is given by

$$\frac{1}{N\sigma}\sum_{i=1}^{N} F_i \epsilon_i,$$

where $F_i = F(\theta_i)$.

Wrapping together, we have one basic ES algorithm for reinforcement learning [algorithm 87].

Algorithm 87: Isotropic multivariate Gaussian evolution strategies for reinforcement learning

Input: Learning rate α, noise standard deviation σ, and initial policy distribution parametersμ_0.

for $t = 1, 2, ...$ **do**
  Sample $\epsilon_1, ..., \epsilon_n \sim N(0, I)$.
  for $n = 1, ..., n$ **do**
    Using perturbed policy parameter $\theta_i = \mu_t + \sigma\epsilon_i$ to generate episodes and get process returns, denoted by $F(\theta_i)$.
    Setting $\mu_{t+1} = \mu_t + \frac{\alpha}{\sigma n} \sum_{i=1}^{n} F_i \epsilon_i$
  end
end

Output: θ

We can also implement ES algorithm in a distributed fashion, as we show in algorithm 88.

Algorithm 88: Isotropic multivariate Gaussian parallelized evolution strategies for reinforcement learning

Input: Learning rate α, noise standard deviation σ, and initial policy distribution parametersμ_0.

Initialize n workers for known random seeds and same inital parameter μ_0 **for** $t = 1, 2, ...$ **do**
  for *each worker* $n = 1, ..., n$ **do**
    Sample $\epsilon_1, ..., \epsilon_n \sim N(0, I)$.
    Using perturbed policy parameter $\theta_i = \mu_t + \sigma\epsilon_i$ to generate episodes and get process returns, denoted by $F(\theta_i)$.
  end
  Send all the scalar F_i from each worker to every other worker.
  for *each worker* $n = 1, ..., n$ **do**
    Reconstruct all perturbations $\epsilon_j, j = 1, n$ from known random seeds.
    Setting $\mu_{t+1} = \mu_t + \frac{\alpha}{\sigma n} \sum_{i=1}^{n} F_i \epsilon_i$
  end
end

Output: θ

ES have several appealing properties compared to RL algorithm discussed before, including indifference to the distribution of rewards (sparse or dense), no need for backpropagating gradients, and tolerance of potentially arbitrarily long time horizons.

37.5 Advanced training strategies

Our previous section [section 37.4] presents a collection of core algorithms that address difficult aspects of the reinforcement learning challenges in tackling complicated decision-making problems[subsection 37.2.5]. These algorithms have been used in multiple domains and have achieved remarkable performance in many long-standing challenging problems.

Despite its remarkable success, training a DRL agent usually requires substantial human effort on hyper-parameter tuning (e.g., learning rate, neural network architecture, exploration-exploitation balance, etc.), particularly for high-dimensional problems, temporally extended, and sparse reward tasks. For these goal-oriented control tasks, deep reinforcement learning can be time-consuming and sample inefficiency because iterative improvement of the agent's behavioral policy requires determination of the long term consequences of their actions using sparse and delayed rewards.

In response to these challenges, this section we continue to present additional advanced training strategies that can speed up convergence and stabilize training.

37.5.1 Priority experience replay

In data-driven reinforcement learning, not all transitions are born equal: some transitions are of more importance because either the action receives large rewards or there is a large TD error associated with the transition. The key idea of prioritized experience replay [12] is to replay these critical experience tuples more often to speed up the Q function approximation learning process. The importance of a experience can measured by absolute TD-error.

After associate a scalar priority p with each transition tuple, they are sampled according to a probability proportional to p^α, where the parameter α determines how much priority is used (if we set $\alpha = 0$, then we will get uniform samples). To property correct the sampling weight, we further perform weighted stochastic gradient descent, which the sampling weight proportional to $1/p^\beta$, where the parameter β determines the impact of importance-sampling weight(To see this, if we set $\alpha = 0$, then we will not use any importance sampling).

Further, to ensure new transitions to be more likely sampled at least once, we assign the highest priority to new samples. Finally, reply experiences in a stochastic manner

is preferred over deterministic replay since it can help avoid the algorithm to exploit approximation error and provide robustness. The algorithm is showed in algorithm 89.

Algorithm 89: Q learning with prioritized experience replay

Input: exponents α and β.

Initialize network Q_0 and replay memory buffer $\mathcal{B}$. **repeat**

  Selection action a_t and observe transition tuple $(s_t, a_t, s_{t+1}, r_{t+1})$. Store transition $(s_t, a_t, s_{t+1}, r_{t+1})$ in $\mathcal{B}$ with priority $p_t = \max_{i \in \mathcal{B}} p_i$. Sample a minibatch of samples, denoted by tuple $(s_j, a_j, s_{j+1}, r_{j+1})$ with probability $Pr(j) = \frac{p_j^\alpha}{\sum_i p_i^\alpha}$.

  Compute importance-sampling weight $w_j \propto [\frac{1}{p_j}]^\beta$

  Perform weight stochastic gradient descent to update Q network.

  Compute TD-error $\delta_j = r_{j+1} + \gamma \max_a Q_t arget(s_{j+1}, a) - Q(s_j, a_j)$.

  Adjust the priorities of these samples by $p_j = \left|\delta_j\right|$.

until *sufficient iterations*;

Output: Q value function

37.5.2 Hindsight experience generation

Hindsight experience replay[17] can be viewed as data-sharing strategy for universal value function approximator learning, where the goal is to learn value functions and control policies that are parameterized by task goals. It exploits the fact that a given experience tuple $(s_t||g, a_t, s_{t+1}||g, r_t||g)$ for a goal g can be used to construct a experience tuple $(s_t||g', a_t, s_{t+1}||g', r_t'||g')$ for another goal g', where r' indicates reward can be goal-dependent In an off-policy learning algorithm, such as DQN and DDPG, these additional experiences almost come free of charge.

The algorithm is showed in algorithm 90.

Algorithm 90: Deep Q-learning with hindsight experience replay

Initialize replay memory $\mathcal{B}$ to capacity N.
Initialize action-value function Q with random weights θ.
Initialize target action-value function Q' with weights $\theta^- = \theta$.
for *episode* = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$.
  Sample a goal $g \in G$ **for** $t = 1, T$ **do**
    With probability ϵ select a random action a_t, otherwise select $a_t = \arg\max_a Q(s_t, g, a; \theta)$.
    Execute action a_t in emulator and observe reward r_t and state x_{t+1}.
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess s_{t+1}.
    Store transition $(s_t||g, a_t, r_t||g, s_{t+1}||g)$ in $\mathcal{B}$.
    For every H steps, sample an additional $g' \in G$, set $r' = r(s_{t+1}, a_t, g')$, and store the transition $(s_t||g', a_t, r'||g', s_{t+1}||g)$ in $\mathcal{B}$.
    Sample random minibatch of transitions $(s_j||g, a_j, r_j||g, s_{j+1}||g)$ from $\mathcal{B}$.
    Set

$$y_j = \begin{cases} r_j, \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, g, a'; \theta^-), \text{otherwise} \end{cases}$$

    Every C steps reset $Q' = Q$.
  end
end

37.5.3 Reverse goal generation

There has been considerable efforts to address the training brittleness and sample inefficiency for these goal-oriented control tasks. One simple yet effective approach is curriculum learning. Curriculum learning[18] first introduced by Bengio to speed up the training in the supervised learning, have also been applied to DRL. The core idea is to schedule the training process such that the agent learns to accomplish simpler tasks before learning to accomplish more complex tasks. For example, the reverse curriculum generation[19], or reverse goal generation, generates curriculum by expanding the start state set via random actions from the goal state and states where the agent has good chances to reach the goal state. The agent will finally learn the control strategies when the start state set covers the desired start state region. Besides the simple heuristics of reverse curriculum generation, there are more sophistic approaches that involves adaptive generation of curriculum from a trainable model[20].

37.5.4 Reverse goal generation on low-dimensional manifolds

37.5.4.1 *Key idea*

The success of the curriculum generated to cover the desired start state region relies on the Markov train induced under random action can sufficiently sample the whole state spaces. This assumption can be satisfied easily for deterministic system such as robotics. For high dimensional dynamics in nano-scale science and engineering, the target state is usually low energy state and there exists various adsorbing state or states with every small escape probability the Markov chain. Use random action to produce starting state can fail to produce diverse and gradually difficulties level curriculum.

To address such limitation, we can use manifold learning to help planning curriculum to accelerate the training of DRL agent for goal-oriented tasks [Figure 37.5.1]. Manifold learning is a subset of nonlinear dimensional reduction algorithms that aim to discover the low-dimensional intrinsic structures from high-dimensional data.

The low dimensional manifold can be used to incorporate prior knowledge to avoid trapping regions during exploration and explore regions that a solution can be found. Knowing the global structure can help design curriculum to effective explore the state space that improves sample efficiency.

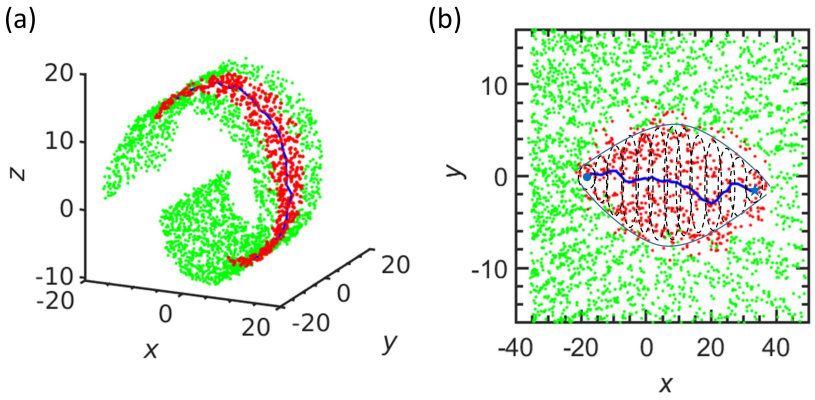


Figure 37.5.1: Illustration of planning curriculum on a swiss roll manifold. Red targets can generating along the path connecting from an intended start point to the target goal position.

37.5.4.2 *Example: navigation on a curved surface*

In our first example, we consider navigating a micro/nano-sized colloidal robot on a curved surface. Colloidal robots have demonstrated great potentials for applications

like precision surgery and targeted drug delivery [21][22][23] inside the patient's body and environmental remediation. Navigating the colloidal robot on curved surfaces is a simplified model of navigating on curved interfaces such as bubbles. The equation of motion of such particle is given by

$$\partial_t r = u + \xi(t) \tag{9}$$

$$s.t. \quad c(r) = 0 \tag{10}$$

where $r \in \mathbb{R}^3$ is the position of the particle, and $c(r) = 0$ is the constraint that the particle has to stay on the surface, and $u \in \mathbb{R}^3$ is the external force field as the control input. We allow six discrete actions, corresponding to $(5,0,0)$, $(-5,0,0)$, $(0,5,0)$, $(0,-5,0)$, $(5,0,0)$, $(5,0,0)$, The ξ is a three-dimensional zero mean white noise stochastic process satisfying $E[\xi(t)\xi(t')] = 2DI\delta(t-t')$, where D is the transnational diffusivity coefficients. The numerical simulation of equation of motion is carried out using a constrained Brownian dynamics algorithm developed in our previous work[24].

Figure 37.5.2 shows the a representative trajectory on the curved surface under a control policy learned via curriculum learning on a low-dimensional manifold.

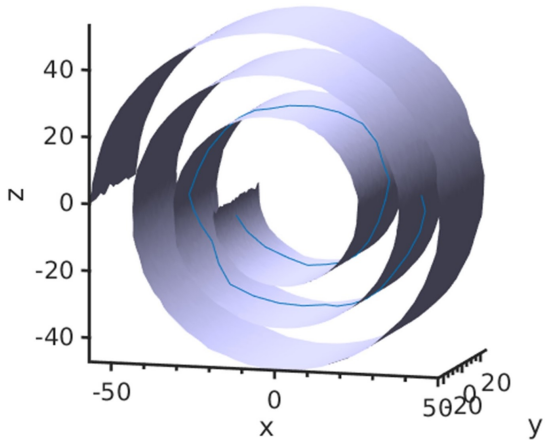


Figure 37.5.2: One representative trajectory on the curved surface via a control policy learned via curriculum learning on a low-dimensional manifold.

37.6 Notes on bibliography

For an overview on Markov decision process, see [25].

For an overview of reinforcement learning, see [2].

Excellent online resources include Spinning Up, Lil'sLog.

BIBLIOGRAPHY

1. Wiering, M. & Van Otterlo, M. Reinforcement learning. *Adaptation, Learning, and Optimization* **12** (2012).
2. Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* (MIT press, 2018).
3. Bertsekas, D. *Dynamic Programming and Optimal Control Athena Scientific optimization and computation series* **v. 2.** ISBN: 9781886529441 (Athena Scientific, 2012).
4. Achiam, J., Knight, E. & Abbeel, P. Towards Characterizing Divergence in Deep Q-Learning. *arXiv preprint arXiv:1903.08894* (2019).
5. Schulman, J., Moritz, P., Levine, S., Jordan, M. & Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).
6. Silver, D. *et al. Deterministic policy gradient algorithms* in *ICML* (2014).
7. Riedmiller, M. *Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method* in *European Conference on Machine Learning* (2005), 317–328.
8. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529 (2015).
9. Van Hasselt, H., Guez, A. & Silver, D. *Deep Reinforcement Learning with Double Q-Learning.* in *AAAI* **2** (2016), 5.
10. Wang, Z. *et al.* Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).
11. Hausknecht, M. & Stone, P. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527* **7** (2015).
12. Mnih, V. *et al. Asynchronous methods for deep reinforcement learning* in *International conference on machine learning* (2016), 1928–1937.
13. Fujimoto, S., van Hoof, H. & Meger, D. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477* (2018).
14. Schulman, J., Levine, S., Abbeel, P., Jordan, M. & Moritz, P. *Trust region policy optimization* in *International conference on machine learning* (2015), 1889–1897.
15. Haarnoja, T., Zhou, A., Abbeel, P. & Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290* (2018).

16. Salimans, T., Ho, J., Chen, X., Sidor, S. & Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).

17. Andrychowicz, M. *et al. Hindsight experience replay* in *Advances in Neural Information Processing Systems* (2017), 5048–5058.

18. Bengio, Y., Louradour, J., Collobert, R. & Weston, J. *Curriculum learning* in *Proceedings of the 26th annual international conference on machine learning* (ACM, 2009), 41–48. ISBN: 1605585165.

19. Florensa, C., Held, D., Wulfmeier, M., Zhang, M. & Abbeel, P. Reverse Curriculum Generation for Reinforcement Learning, 1–14 (2017).

20. Matiisen, T., Oliver, A., Cohen, T. & Schulman, J. Teacher-Student Curriculum Learning (2017).

21. Li, J., Ávila, B. E.-f. D., Gao, W., Zhang, L. & Wang, J. Micro / nanorobots for biomedicine : Delivery , surgery , sensing , and detoxification (2017).

22. Mallouk, T. E. & Sen, A. Powering Nanorobots. *Scientific American* **300,** 72–77 (2009).

23. Yang, Y., Bevan, M. A. & Li, B. Efficient Navigation of Active Particles in an Unseen Environment via Deep Reinforcement Learning, 1–14 (2019).

24. Yang, Y. & Li, B. A Simulation Algorithm for Brownian Dynamics on Complex Curved Surfaces. *arXiv preprint arXiv:1908.07166* (2019).

25. Puterman, M. L. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming* (John Wiley & Sons, 2014).