Machine Learning

Benjamin Köhler

Contents

Part I

Deep Learning Basics

Chapter 1

Supervised Learning

1.1 What is machine learning?

The main objective of machine learning is to learn from data sets without explicit programming. This should be seen in contrast to the AI in old computer games which used a plethora of **if**'s and **else**'s to feign intelligence to the player. The most common example is chess which was one of the first game engines to transition from programmed to machine learning intelligence. Machine learning is best suited for problems that are very complicated to hard-code solutions to, but can intuitively be judged by a human, e.g. flying an air plane.
A formal definition might be like this: A well-posed learning problem is when a computer program is said to **learn** from experience E with respect to some task T and some measure of performance P, if its performance on T, as measured by P, improves with experience E.
Example: Computers learn from me labelling mails as spam (E) how to better filter spam E-Mails (T). The performance measure is the ratio of correctly classified spam mails to falsely classified mails

1.2 Linear regression

Regression is one of the most often used and simplest machine learning methods. It is an example of a supervised learning algorithm. Here we train the algorithm with pairs of data, the first being the input and samples of output that we desire our algorithm to produce from the former. Our hope is that for input for which we don't know the correct output the algorithm then yields decent approximations. This is sketched in figure 1.1.

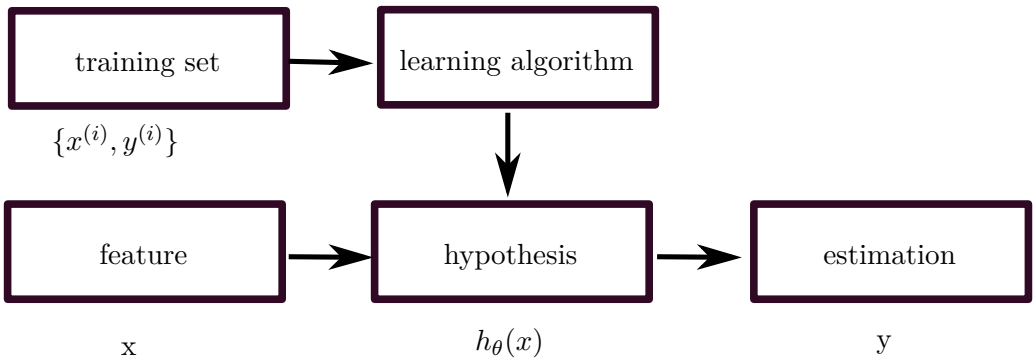


Figure 1.1: basic principle of a supervised learning algorithm

In regression problems the inputs and outputs are numbers, i.e. we learn functions. For linear regression these are linear functions. We start with univariate functions or hypotheses with only one feature x

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1.1}$$

where the θ_i are parameters. According to the definition of the last section, the task here is to find $f(x)$ as close to the desired outputs y_i for given inputs x_i. The performance P or cost function C of our algorithm is measured by the cumulated sum of the squared differences between $h_\theta(x^{(i)})$ and y:

$$C(\boldsymbol{\theta}) = \sum_{i=1}^{m} \frac{\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2}{2m}. \tag{1.2}$$

Here m is the size of the training set $\left\{x^{(i)}, y^{(i)}\right\}$. We want to minimise C with respect to θ_0 and θ_1. For this simple form of the cost function, the gradient decent method is certain to find the minimum. Its update rule for the parameters θ_0 and θ_1 is

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} C(\boldsymbol{\theta}) \tag{1.3}$$

with the learning rate α. If the whole training set is used in each gradient decent step, the procedure is called **batch gradient decent**.
Of course there can be more than one feature, i.e. $\boldsymbol{x}$ is a vector. The hypothesis is then

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta} \cdot \boldsymbol{x} \tag{1.4}$$

and the self-consistency equation for gradient decent is

$$\theta_i = \theta_i - \alpha x_i \cdot \left(\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta})\right)_i \tag{1.5}$$

with $x_0 = 1$. One has then to be careful with the size of the features. If these are of different magnitude, the learning process can be prolonged. Therefore, it is recommended to scale the features before the training process. The following transformation reduces the features to the interval $[-1, 1]$

$$\tilde{x}_i = \frac{x_i - \mu_i}{\max(x_i) - \min(x_i)} \tag{1.6}$$

with μ_i being the mean of the feature x_i in the training set.
It should be noted that a polynomial regression is easily possible by using different powers and products of the respective features as additional features. The normal equation can be used instead of gradient decent as well. Here the cost function is optimised analytically. To do so one constructs the matrix X and a vector $\boldsymbol{y}$

$$X = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & x_3^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}, \quad \boldsymbol{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}. \tag{1.7}$$

The solution is then

$$\boldsymbol{\theta} = \left(X^\intercal X\right)^{-1} X^\intercal \boldsymbol{y}. \tag{1.8}$$

The complexity of the normal equation procedure is $\mathcal{O}(m^3)$ and, therefore, not recommended for a large size of the feature space (a threshold is about 10^4). Should $X^\intercal X$ be non-invertible, there are redundant features or even more features than data points in the training set. This issue can be avoided by using **regularisation** and/or fewer features.

1.3 Classification problem (logistic regression)

Often times, the desired output is not a number but a classification, i.e. an element of a finite set. If the classes are known beforehand and there are training data, the method that is described here can be used. If the classes are not known, the problem is solved by a unsupervised learning algorithm called **clustering** which is described in section 2.1.
Examples: E-Mail is spam or not, online transaction is fraudulent or not, tumour is malignant or benign.
One possibility would be to use linear regression for this purpose. First, one could map the classes to numbers and do the training process. Second, one would classify an unknown input of features as the class-number closest to the number of the output. However, this approach is a very bad idea.
A much better choice is to use logistic regression. In principle every classification problem can be reduced to many binary classification problems. Therefore the discussion starts with this and at the end it is shown how to generalise to a multi-class problem.
First of all, one uses a hypothesis function that takes values between 0 and 1 (binary classification problem). One example is the **sigmoid** or **logistic function**

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{1 + \mathrm{e}^{-\boldsymbol{\theta}\cdot\boldsymbol{x}}}. \tag{1.9}$$

This function is interpreted as a probably that given $\boldsymbol{x}$ the element belongs to the class 1. The threshold value when one decides whether an unknown data set is classified can be chosen freely and does not have to be $\frac{1}{2}$. An appropriate cost function is

$$C(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right) + \left(1 - y^{(i)}\right)\log\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)\right]. \tag{1.10}$$

It is easily checked that this function is minimal for $y^{(i)} = h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$ and that the gradient of this function is

$$\nabla_{\boldsymbol{\theta}} \cdot C = \frac{1}{m}\sum_{i=1}^{m}\boldsymbol{x}^{(i)}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right) \tag{1.11}$$

which is similar to the regular linear regression cost function's gradient. Therefore, the update rule of the learning parameters $\boldsymbol{\theta}$ is the same. Besides a regular gradient decent, more efficient methods are conjugate gradient[1], BFGS, and L-BFGS[2]. Those have the advantage of not having to specify the learning rate of the algorithm as for the regular gradient decent and they run faster too.
To solve the **multi-class** classification problem one uses one versus all (other) classification, i.e. out of the initial n classes one builds n sets of binary classes. From those one can use the binary class algorithm to pick the classifier $h_{\boldsymbol{\theta}}^{(i)}(\boldsymbol{x})$ out of the binary classifier set $\mathcal{C}$ for an unknown object $\boldsymbol{x}$ that yields the highest probability, i.e.

$$y = \max_{i\in\mathcal{C}} h_{\boldsymbol{\theta}}^{(i)}(\boldsymbol{x}) \tag{1.12}$$

[1] An implementation of this method can be found in the example programs.
[2] As of the writing, the method of choice is **ADAM**.

1.4 Regularisation

A model is said to **underfit** the data or to have a **high bias** if it doesn't match the desired outcome of the labels in the training set. In this case the hypothesis is not appropriate to describe all the training data and it might need to be updated, e.g. by including higher orders in the polynomial. If a model does predict the labels in the training set very well, yet performs unreliable on a generic test set (which usually is a subset of the training set which was excluded from the training process), the hypothesis is said to **overfit** the data or to have a **high variance**. reasons for this are usually a too ambitious hypothesis which has too many parameters compared to the complexity of the underlying problem. In the training process, the parameters are learned so to fit all possible training points best, but for the test set, this choice does not generalise. One way to avoid such a behaviour is to reduce the complexity of the hypothesis and reduce the number of features, e.g. by removing higher orders in the polynomial.
A second way of getting rid of overfitting or high variance hypothesises is to use **regularisation**. Here all features are kept, but their individual influence is reduced by limiting their magnitude. One does this by introducing the regularisation term C_{reg} into the cost function:

$$C_{\text{reg}} = \frac{\lambda}{2m} \left(\boldsymbol{\theta} \cdot \boldsymbol{\theta} - 1 \right). \tag{1.13}$$

Here the -1 appears due to the omission of the bias parameter θ_0 which can not contribute to overfitting. The regularisation parameter λ can be used to control the amount of overfitting and underfitting. For large λ the data are underfitted and for small values the data will get overfitted. The new update rule for the learning parameters $\boldsymbol{\theta}$ with the regularisation term is

$$\begin{aligned} \theta_0 &= \theta_0 - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)} \right) x_0^{(i)} \\ \theta_j &= \theta_j \underbrace{\left(1 - \frac{\alpha\lambda}{m} \right)}_{\leq 1} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)} \right) x_j^{(i)} \end{aligned} \tag{1.14}$$

1.5 Neural networks

For a classification problem with many features, a non-linear hypothesis has exponentially more features and is, therefore, more prone to overfitting due to the huge feature space. In image recognition, e.g. one has millions of pixels. **Neural networks** are used to solve this problem. Initially they were designed to simulate or mimic the brain. Different pieces of brain tissue usually enable one to do specific tasks, e.g. to feel, hear, etc. In principle, any part of the brain can learn how to perform different tasks. People have been taught to see with their ears. The question arose whether computer neurons can do the same.
A computer neural network is shown in Figure 1.2. Every neural network has a number of input neurons and of output neurons. Optionally there are hidden layers with additional neurons. The neurons are connected with each other via free parameters $\boldsymbol{\theta}$ which can be represented by a matrix. Each neuron in a layer not in the input layer has an activation function

$$a_i^{(\mu)} = g\left(\theta_{ij}^{\mu} x_j \right). \tag{1.15}$$

Latin indices refer to the neurons that are used to calculate the value of the neurons in the next layer and Greek indices refer to the layer of the neurons. I use Einstein sum convention to keep the notation short. It is possible to represent the linear regression and classification problem with a neural network. The input layer then contains all the features x that have been selected by the data engineer to model the problem and the output layer is calculated using the activation function or hypothesis. The process of calculating the value of the output neurons with the activation function(s) from the input and optionally the neurons in the hidden layers is called the **forward propagation**.

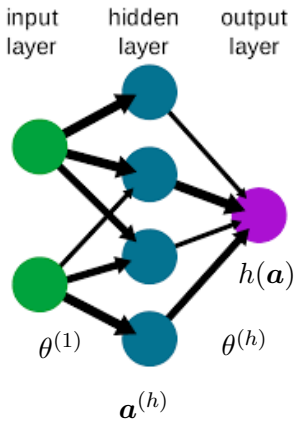


Figure 1.2: general layout of a neural network

Neural networks unfold their full potential when hidden layers are used. Then the nets learn their own features from the the defined input features. The training is done with the same cost function as before. It has proven useful to calculate the gradients needed for the training by a process called **back-propagation**[3]. From the errors due to the layer in front of a neuron, one can calculate the error due to that neuron. For the output layer the error is

$$\delta_j^{(\Lambda)} = a_j^{(L)} - y_j, \tag{1.16}$$

and for any other layer it is

$$\delta_j^{(\lambda)} = \left(\theta^{(\lambda)}\right)^{\mathsf{T}} \boldsymbol{\delta}^{(\lambda+1)} \cdot g'(z^{(\lambda)}). \tag{1.17}$$

The derivative $g'(z^{(\lambda)})$ of the activation function is with respect its parameters. For the logistic activation function, the derivative is

$$g'(z) = \left(\frac{1}{\mathrm{e}^z + 1}\right)' = g(z)(1 - g(z)). \tag{1.18}$$

The input layer has no errors by definition and $\delta^{(1)} = 0$. Without regularisation the gradient of the cost function is

$$\frac{\partial S(\boldsymbol{\theta})}{\partial \theta_{ij}^{(\lambda)}} = a_j^{(\lambda)} \delta_i^{(\lambda+1)}. \tag{1.19}$$

back-propagation algorithm:

training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

set $\Delta_{ij}^{(\lambda)} = 0 \; \forall \; i, j, \lambda$

for $n = 1$ to m

set $a^{(1)} = x^{(n)}$

perform forward propagation to compute $a^{(\lambda)}$ for $\lambda = 2, 3, \ldots \Lambda$

using $y^{(n)}$, compute $\delta^{(\Lambda)} = a^{(\Lambda)} - y^{(n)}$

compute $\delta^{(\Lambda-1)}, \delta^{(\Lambda-2)}, \ldots, \delta^{(2)}$

$\Delta_{ij}^{(\lambda)} = \Delta_{ij}^{(\lambda)} + a_j^{(\lambda)} \delta_i^{(\lambda+1)}$ or $\Delta^{(\lambda)} = \Delta^{(\lambda)} + \delta^{(\lambda+1)} \otimes a^{(\lambda)}$

[3]Mathematically it is nothing but a clever use of the chain rule

It is recommended to do gradient checking when implementing the back-propagation algorithm on one own, i.e. calculate

$$\partial_{\theta_i} S(\boldsymbol{\theta}) \approx \frac{S(\theta_i + \epsilon) - S(\theta_i - \epsilon)}{2\epsilon} \tag{1.20}$$

and check whether this is similar to the result of the back-propagation algorithm. Additionally the parameters $\boldsymbol{\theta}$ should never be set to zero as this results in $a_1^{(2)} = a_2^{(2)}, \delta_1^{(2)} = \delta_1^{(2)}$ so that after back-propagation $\theta_{01} = \theta_{02}$ after each update. The neural net can not compute interesting functions for that reason. It is recommended to initialised the parameters to a small random value[4].
To train a neural net, the following steps need to be taken:

1. pick a net architecture (connectivity between neurons): the number of input neurons is the dimension of the feature vector $\boldsymbol{x}$, the number of output neurons is the dimension of the classes or the dimension of the space the function maps into. A reasonable default is to start with one hidden layer or when starting with more hidden layers to have the same number of neurons in each (usually the more the better)

2. Training the neural net:

   - randomly initialise weights $\boldsymbol{\theta}$
   - implement forward-propagation to get $h_{\boldsymbol{\theta}}(x^{(i)})$ for any $x^{(i)}$
   - implement code to calculate the cost function
   - implement back-propagation to compute partial derivatives $\frac{\partial S(\boldsymbol{\theta})}{\partial \theta_{jk}^{(\lambda)}}$
   - use gradient checking to compare $\frac{\partial S(\boldsymbol{\theta})}{\partial \theta_{jk}^{(\lambda)}}$ computed using back-propagation vs. numerical estimate of gradient of $S(\boldsymbol{\theta})$
   - disable gradient checking code, if all is okay
   - use gradient decent or advanced optimisation method with back-propagation to try to minimise $S(\boldsymbol{\theta})$ as a function of parameters $\boldsymbol{\theta}$
     $\rightarrow$ (as $S(\boldsymbol{\theta})$ is not convex, the algorithm can get stuck in local minima;
     in practice this is not a problem, though.)

1.6 Advice for using machine learning

Here are some tips when your machine learning algorithm yields huge errors, when applied to other data sets:

- get more training examples (not always an improvement)
- try smaller sets of features (careful selection)
- try getting additional features like, e.g. polynomial features
- try decreasing the regularisation parameter λ
- try increasing the regularisation parameter λ

Some of the above options can be ruled out quickly by using **machine learning diagnostics** which test what is and what isn't working. These take some time to implement, but it is definitely worthwhile.

[4] With a random sign too!!

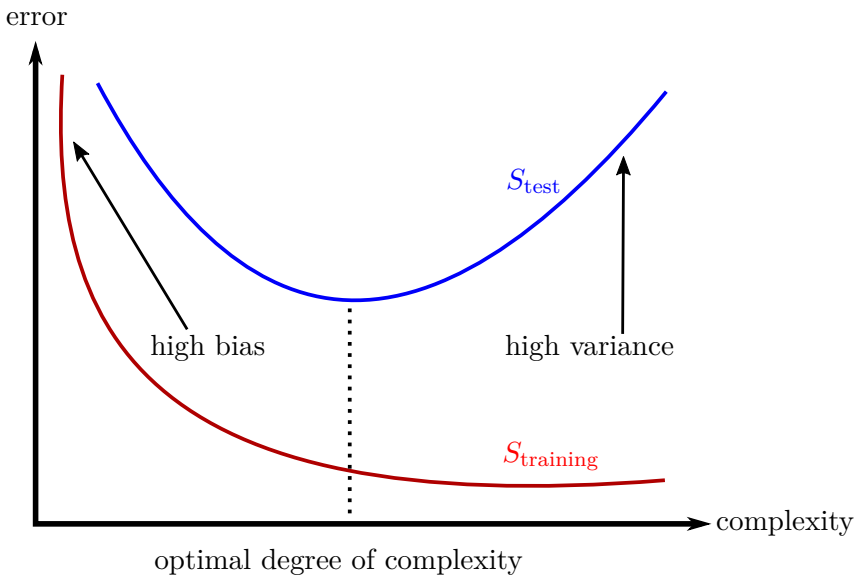


Figure 1.3: Sketch of possible error curves for the training and test set over the models complexity

1. **Determining the quality of a model:** Evaluate the hypothesis (a low training error is not a sign that it is working well) → is it overfitting/underfitting?

   It is common practise to split your training set into smaller subsets: circa 70% should be used for training the model and 30% to test the performance on independent data. Sometimes it is useful to have 10% in a additional validation set that probes the quality after different parameter configurations (learning rate, regularisation parameter etc) are probed by the training and test set. The validation set is then used to finally determine the performance of the trained model. Remember to randomly order your initial training set before the split into the subsets to avoid any coincidental order.[5]

   You should train the model with the training subset by optimising the cost function $S_{\text{training}}(\boldsymbol{\theta})$ and evaluate the performance with the cost function $S_{\text{test}}(\boldsymbol{\theta})$ which could be the misclassification error

   $$\text{err}\Big(h_{\boldsymbol{\theta}}(x_{\text{test}}), y_{\text{test}}\Big) = \frac{1}{m_{\text{test}}} \sum_{x_i \in x_{\text{test}}} \begin{cases} 1 & \text{if } h_{\boldsymbol{\theta}}(x_i) \geq p_{\text{threshold}} \text{ and } y_i = 0 \\ 0 & \text{else} \end{cases} \tag{1.21}$$

   here $p_{\text{threshold}}$ is the threshold set to classify a element as belonging to a class and m_{test} is the number of samples in the test set.

2. **Model selection problem:** If you have different choices for the hypothesis, you should increase its complexity slowly and check when the performance (measured by the test set error) is best. Note that this is often too optimistic as the extra parameter might be fit to the test set. This is the reason why it is recommended to have the validation set too.

3. **Bias versus variance problem:** Plotting the error due to the training and due to the test set over the complexity of the model (e.g. the number of features) is a good way to distinguish between a model that is overfitting (high variance) and one that is underfitting (high bias). A possible situation that might arise is sketched in Figure 1.3.

   For a model that is underfitting or that has a high bias, it is common that $S_{\text{training}} \approx S_{\text{test}}$ and for a model that is overfitting or has a high variance $S_{\text{training}} \ll S_{\text{test}}$. The effect of the regularisation parameter is the following: for high λ the algorithm favours models that underfit and for low λ it favours models that overfit the data. It is again advantageous to

[5] In the literature one finds different names for the three sets. The training subset is mostly unanimously named this way. Some authors refer to the test subset as I use it as cross validation set and name the validation subset as the training set. Be aware that different naming conventions exist!

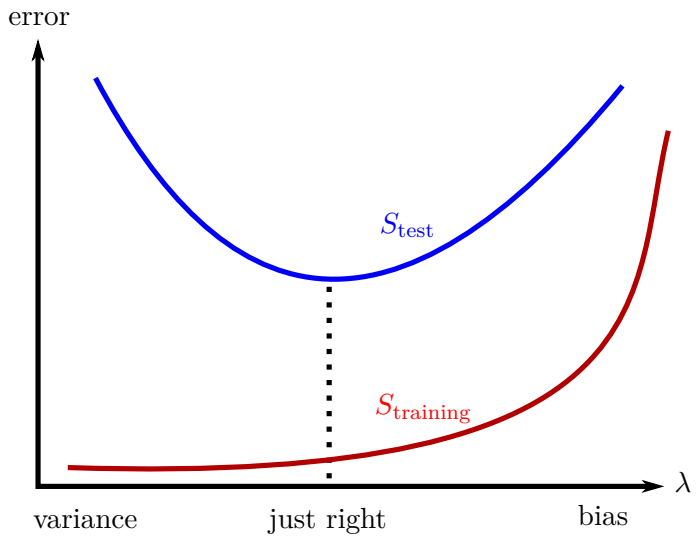
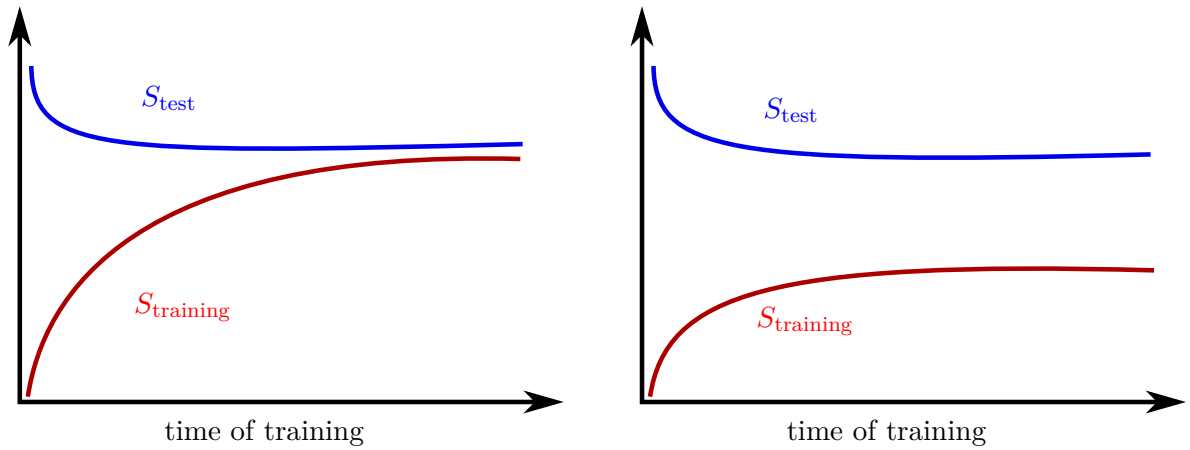


Figure 1.4: Sketch of possible error curves for the training and test set over the value of the regularisation parameter λ.

draw a graph to find the right value for λ. Here the cost functions are plotted over the regularisation parameter's value. For the test set it is better to not include the regularisation term as it is not interesting for practical use (the quality of the model is the ability to predict the right value or class y). Figure 1.4 shows a common behaviour.

It is common practise to double the regularisation parameter starting from 0.01 up to about 10 and calculate the test error S_{test} to pick the value whichever minimises it. At the end, both the model's complexity as well as the regularisation parameter should be optimised parallel to each other, i.e. to calculate S_{test} for all values of λ for all models of different complexity. Again one picks the optimal parameter configuration with respect to S_{test}. At last, the quality is evaluated using the validation set to see whether the model is a good generalisation of the problem at hand.

4. **Learning curves:** During the training process different conclusions can be drawn to determine necessary measures to improve the quality of a model. Figure 1.5 shows examples for curves with a high variance and high bias. One can see that for a model with high bias the cost functions for the training and the test set lie much closer together, but that the quality of the model is quite poor for the training set. For a model with a high variance, the quality is okay for the training set, yet the model does not generalise to the test set.



(a) high bias (b) high variance

Figure 1.5: Sketch of a training curve with (a) high bias and (b) high variance

There is a huge gap between the cost functions of both sets. For a model with high bias, getting more training date helps, while for a model with high variance this is not the case.

In general one can summarise the content of this section the following way: Options and when they are appropriate:

- get more training examples → fix high variance
- try smaller set of features → fix high variance
- try getting additional features → fix high bias
- try adding polynomials → fix high bias
- try decreasing the regularisation parameter λ → fix high bias
- try increasing the regularisation parameter λ → fix high variance

For neural networks:

- start with a small neural network → risk of underfitting, but computational cheaper
- larger neural neural networks → risk of overfitting (use regularisation) and computationally expensive
- more hidden layers → computationally more expensive (use use test set to optimise the network's layout

1.7 Machine learning system design

Many amateurs in machine learning espouse the idea that collecting a huge amount of data is the most important thing to attain a good results. Without good reason this is never a good idea. There should always be a brainstorming before starting a project. More sophisticated features and developing algorithms that process the input can be very helpful. It can also be very time-consuming without much reward, though. A widely recommended approach is the following:

- start with a simple quick to implement algorithm and test it on the test data
- plot the learning curves as introduced in the last section to decide if more data, more features, etc. are likely to help
- error analysis: manually examine the examples (in the test set) that the algorithm made errors on → can a systematic trend be identified in what type of example the algorithm fails?
- it is very important to get error results as a single numerical value → otherwise it is difficult to assess the algorithm's performance

1.8 Error metrics for skewed classes

When the number of positive and the number of negative examples, results, or outcomes is very different, one calls the respective classes **skewed**. In this case the error rate is not a good classifier. It is better to use the **precision** and **recall**. These two metrics make use of the four possible results of classification by the algorithm. The algorithm can classify an element correctly as either a positive ($y = 1$) or a negative ($y = 0$) example. In the same way, the

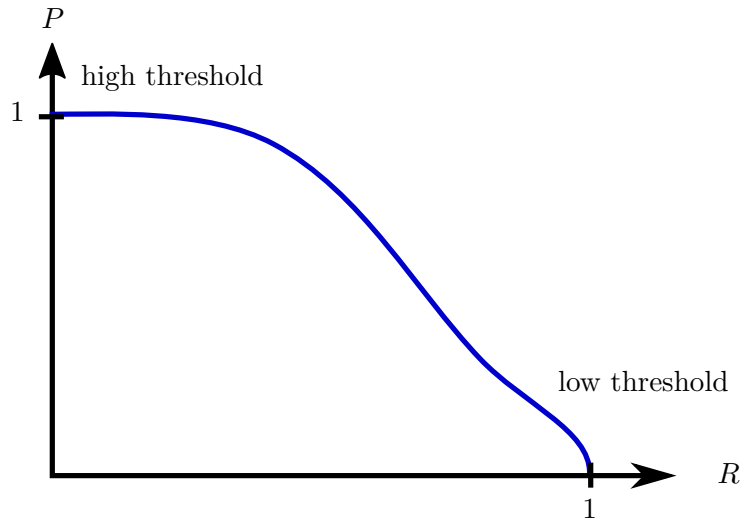


Figure 1.6: Sketch of the relationship between the precision P and the recall R as a function of the threshold value for predicting $y = 1$ for h_θ.

algorithm can classify an element falsely to both binary classes. The precision P and the recall R are defined as

$$P = \frac{\#\text{true positive}}{\#\text{predicted positive}} = \frac{\#\text{true positive}}{\#\text{true positive} + \#\text{false positive}}, \tag{1.22}$$

$$R = \frac{\#\text{true positive}}{\#\text{actual positive}} = \frac{\#\text{true positive}}{\#\text{true positive} + \#\text{false negative}}. \tag{1.23}$$

To control the trade-off between precision and recall the following measures can be taken:

- to avoid false positives: a class can be predicted, if the algorithm is very confident, i.e. h_θ is closer to 1 than to 0.5 $\rightarrow$ higher precision, lower recall
- to avoid false negative: a class can be predicted, if the algorithm is at least suspicious, i.e. h_θ is at least closer to 0.5 than to 0.0 $\rightarrow$ higher recall, lower precision
- in general one only predicts $y = 1$ if $h_\theta \leq$ threshold. The result is shown in Figure 1.6. The decision for the threshold's value can be automatised using the F_1 score

  $$F_1 = \frac{2PR}{P + R}. \tag{1.24}$$

  The worst score is for $P = 0$ or $R = 0$ and the best for $P = R = 1$.

1.9 Data for machine learning

Algorithms can outperform other algorithms with more training data even though they perform poorer with less training data. Still the feature(s) need to have sufficient information to predict outcomes accurately. A very useful test is to ask oneself: ¨Can a human expert given x confidently predict y?¨ A larger parameter space, e.g. a neural network with many hidden units, is more prone to be a low bias algorithm. However, one has to use a large test set size to prevent the possibility of overfitting and make the algorithm low variance.

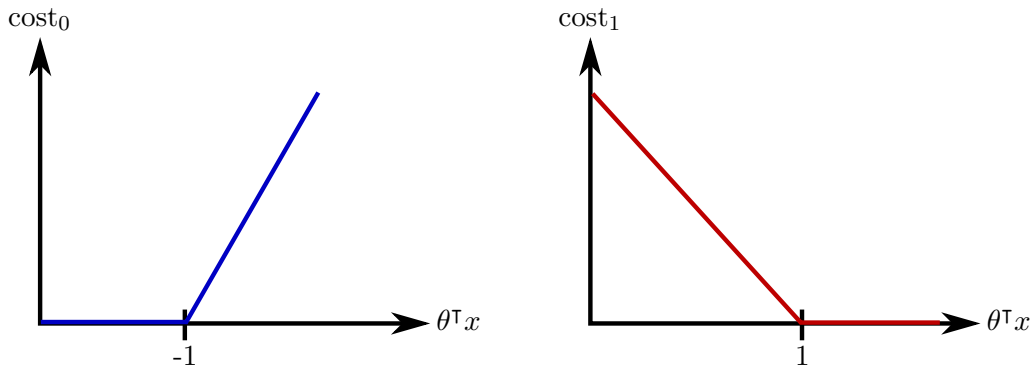


Figure 1.7: Sketch of the two cost functions cost_0 and cost_1 in equation (1.26)

1.10 Support vector machines

The logarithm of the sigmoid function in logistic regression is complicated to handle in optimisation and computation. One, therefore, uses a replacement function that has the property of being close to one for very large arguments and being close to zero for very small arguments. The replacement function can be defined piecewise to be monotonically decreasing (linearly) for arguments smaller than -1, is equal to zero from -1 to 1 and then monotonically increases (linearly) for greater arguments.

1.10.1 Linear decision boundaries

The cost function of such a *support vector machine* (SVM) is

$$S(\theta) = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}\text{cost}_1\left(\theta^\intercal x^{(i)}\right) + (1-y^{(i)})\,\text{cost}_0\left(\theta^\intercal x^{(i)}\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2. \tag{1.25}$$

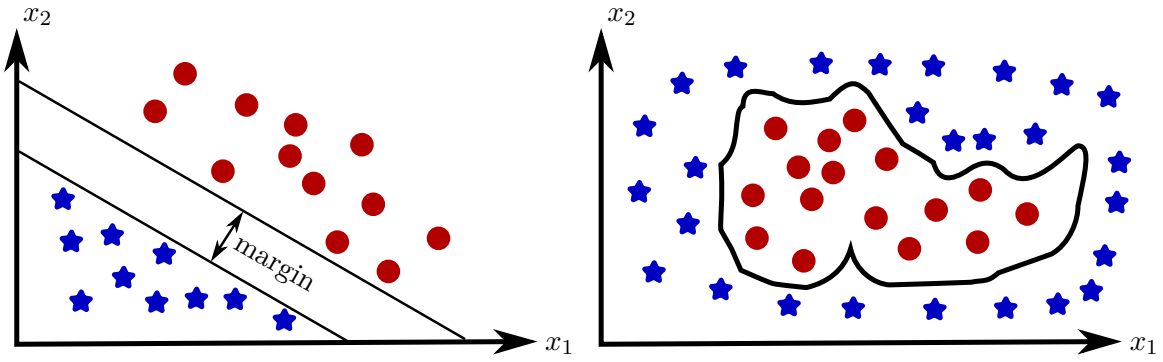
As for the logistic cost function, the objective is to minimise something like $A+\lambda B$ with matrices A and B, as well as λ as big as possible to prevent overfitting or equivalently $cA+B$ where c has to be as large as possible to prevent overfitting. The SVM objective is to find (in Figure 1.7 the two cost functions cost_0 and cost_1 are sketched)

$$\min_\theta c\sum_{i=1}^{m}\left[y^{(i)}\text{cost}_1\left(\theta^\intercal x^{(i)}\right) + (1-y^{(i)})\,\text{cost}_0\left(\theta^\intercal x^{(i)}\right) + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2\right] \tag{1.26}$$

with the hypothesis

$$h_\theta(x) = \begin{cases} 1, & \theta^\intercal x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{1.27}$$

For $y=1$ we want $\theta^\intercal x \geq 1$ (not just ≥ 0) and for $y=0$ we want $\theta^\intercal x \leq -1$ (not just ≤ 0), i.e. there needs to be a **safety margin**. For large c the term with cost functions should be very small, i.e. whenever $y^{(i)}=1 : \theta^\intercal x \geq 1$ and whenever $y^{(i)}=0 : \theta^\intercal x \leq -1$. In other words: one is looking for the minimum of the regularisation term with the constraints $\theta^\intercal x \leq -1$ if $y^{(i)}=0$ and $\theta^\intercal x \geq 1$ if $y^{(i)}=1$.
SVM tries to classify the data with as large a margin as possible. It is not sensitive to outliers though, i.e. it yields no negative consequences if c is **not** too large. The objective of the large margin classification is to minimise the regularisation term $\sum_{j=1}^{n}\theta^2$ with respect to the constrains introduced before. Geometrically the SVM's minimise the length of the vector θ and fulfil the constraints. It is easily seen that the constrains are fulfilled for very large $x||\theta||$. By construction the decision boundary is perpendicular to θ. Hence, the vector θ is either very large or the decision boundary far away from all data points. Figure 1.8(a) illustrates this for a two-dimensional example.



(a) (b)

Figure 1.8: Sketch of (a) the linear largest margin decision boundary and (b) the classification using non-linear kernel function for the decision boundary

1.10.2 kernels

For non-linear decision boundaries higher order polynomials can be used, e.g. predict $y = 1$ if $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_{n+1} x_1^2 + \theta_{n+2} x_1 x_2 + \cdots \geq 0$. The question then arises whether their is a better choice for the features. Indeed, it is best for given x to compute new features f_i depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$ (points in training data space), eg. with a Gaussian kernel:

$$\begin{aligned} f_1 &= \exp\left(-\frac{||x - l^{(1)}||^2}{2\delta^2}\right), \\ f_2 &= \exp\left(-\frac{||x - l^{(2)}||^2}{2\delta^2}\right), \\ f_3 &= \exp\left(-\frac{||x - l^{(3)}||^2}{2\delta^2}\right). \end{aligned} \tag{1.28}$$

This ensures that if x is close to $l^{(i)}$ then $f_i \approx 1$ and if it is far away from $l^{(i)}$ that $f_i \approx 0$. The landmarks define new features for which the hypothesis predicts $y = 1$ if $\theta_0 + \boldsymbol{\theta} \cdot \boldsymbol{f} \geq 0$ which defines a boundary in a much more sophisticated way. A sketch can be seen in Figure 1.8(b). It is recommended to use libraries for the minimisation (liblinear, libvsm, etc) of the respective cost function. The choice for the parameters influences the results in the following ways:

- $\mathbf{c} = \frac{\mathbf{1}}{\lambda}$: large c lead to models that are more prone to have low bias and high variance

  small c lead to algorithms that are more prone to higher bias and low variance

- $\boldsymbol{\sigma^2}$: large values lead to features varying more smoothly and therefore models with a higher bias and lower variance

  small values lead to features varying more abruptly and therefore models with a lower bias and higher variance

When SVM-software is used, one has to ensure that the features are scaled as, e.g. for the Gaussian kernel large features tend to dominate extremely. Furthermore there are more esoteric options as string kernels for text inputs. Furthermore one has to ensure that Mercer's theorem is fulfilled (the kernels are continuous symmetric non-negative definite). Again, a multi-class classification can be realised with many one-vs-all binary classifications. For deciding whether logistic or SVM-classification should be used, the following guidelines can be used:

- for a large number of features relative to the number of samples in the training set one should use logistic regression or SVM without a kernel (linear kernel)

- for a small number of features and an intermediately large training set one should use a SVM with a Gaussian kernel
- for a small number of features ad a huge training set one should add or create more features and then use logistic regression or SVM without a kernel
- neural networks are likely to work in most of the settings, but may be slower to train

Chapter 2

Unsupervised Learning

Unsupervised learning algorithms do not use labels for the data, i.e. there is no $y(x)$ for a given set of features x. One uses these algorithms to find structure in data sets. This can be determining whether there is a certain number of classes in this set or to separate rare classes from the overall data set. Examples include the organisation of computing clusters, astronomical data analysis, analysis of social networks and the like.

2.1 Clustering (k-means algorithm)

The simplest algorithm for clustering is the **k-means algorithm**. It essentially has two steps:

1. randomly initialise a given number in the data set as centroids (¨colouring¨) and all other points to whichever centroid they are closest to
2. move centroids in direction of the average points of one ¨colour¨

The input of the algorithm is the desired number of clusters k specified by the user and the training set $\left\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\right\}$ with $\boldsymbol{x}^{(j)} \in \mathbb{R}^n$. Note that the algorithm will still find exactly k clusters even though there might not be as many or even more.
The cost function that has to be minimised is the sum of all points to their respective centroid $\boldsymbol{\mu}^{(i)}$ by assigning the data points $\{\boldsymbol{x}\}$ to the correct clusters. We denote the assignment of data point $\boldsymbol{x}^{(i)}$ by $c^{(i)}$:

$$S(\{\boldsymbol{c}^{(i)}\}, \{\boldsymbol{c}\}) = \frac{1}{m} \sum_{i=1}^{m} ||\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(i)}||^2. \tag{2.1}$$

There are two parameters to be optimised. The first is the location of the centroid which is the location of the randomly picked points for the initial assignment and calculated after after each reassignment of the data points as

$$\boldsymbol{\mu}^{(i)} = \frac{1}{|c^{(i)}} \sum_{\boldsymbol{x}^{(i)} \in c^{(i)}} \boldsymbol{x}^{(i)}. \tag{2.2}$$

The reassignment is according to which centroid a data point is closets to. After a certain number of cycles the algorithm will converge which might be determined by the change in the cost function. With a bad initialisation, the centroids the algorithm converged to can be a local minima. It is, therefore, recommended to try many initialisation and choose the one with the lowest cost function.
There are two possible methods to determine the number k of clusters:

1. **elbow method:** vary k and choose the k for which the cost function stops to decrease as fast as before. this is, however not recommended
2. Sometimes one has an idea how well the metric k will fit the purpose, e.g. how many T-shirt sizes one wants (s,m,l or xs, s,m,l,xl) so one chooses the number of clusters accordingly.

2.2 Dimensionality reduction/ data compression (PCA)

For the selection of features, it is useful to determine whether some features are linearly dependent and if possible reduce the number of features by constructing new features out of a bigger set of the initial features. The **PCA-algorithm** tries to find the surface, to which the data points have the minimal distance. To obtain good results, the features need to be scaled. PCA is not linear regression as PCA minimises the orthogonal distance of the points to the surface and no values need to be predicted.
The preprocessing for PCA should rescale all the initial features x to $\tilde{x}$ by

$$\tilde{x}_i^{(j)} = \frac{x_i^{(j)} - \mu_i}{s_i} \tag{2.3}$$

where μ_i and s_i are the average value and the standard deviation for each component of the feature vectors, respectively. To reduce the data from n-dimensions to k-dimensions one

1. computes the covariance matrix

$$\Sigma = \frac{1}{m}\sigma_{i=1}^{m} \boldsymbol{x}^{(i)} \left(\boldsymbol{x}^{(i)}\right)^{\intercal} \tag{2.4}$$

2. computes the eigensystem (or better the singular value decomposition) of Σ with eigenvector matrix U and eigenvalues λ. The eigensystem needs to be ordered from the largest to the smallest eigenvalue.

3. builds the reduced eigenvector-matrix $U_{\text{red}} \in \mathbb{R}^n \times \mathbb{R}^k$

4. calculates new reduced features $\boldsymbol{z}^{(i)} = U_{\text{red}}^{\intercal} \boldsymbol{x}^{(i)}$ with by construction $\boldsymbol{z}^{(i)} \in \mathbb{R}^k$

If one wants to reconstruct the compressed representation, this is done by

$$\boldsymbol{x}_{\text{approx}}^{(i)} = U_{\text{red}} z^{(i)}. \tag{2.5}$$

A guideline for choosing the number k of principle components is the average of the squared projection error with respect to the total variation in the data

$$\frac{\sum\limits_{i=1}^{m} ||\boldsymbol{x}^{(i)} - \boldsymbol{x}_{\text{approx}}^{(i)}||^2}{\sum\limits_{j=1}^{m} ||\boldsymbol{x}^{(j)}||^2} \leq \epsilon \tag{2.6}$$

where ϵ is a threshold value for what percentage of information is allowed to be lost (common values are $\epsilon = 0.01$). One starts with $k = 1$ and then gradually increases this number until above equation is fulfilled.
The most common use of PCA is to speed up supervised learning especially for image recognition. While the dimension of the initial training set is $\mathbb{R}^{n+m}$, it is $\mathbb{R}^{k+m}$ for the new training set in the reduced basis. PCA should only be used on the training set of the data set and not on the test or cross validation sets. Later this projection from the training set only can be applied to the other subsets. PCA is also used to visualise data. A bad idea is to use PCA instead of regularisation to prevent overfitting. Before implementing PCA, one should first try running whatever with the original/raw data. Only if this does not what it should, one can implement PCA and consider using the $\boldsymbol{z}^{(i)}$.

2.3 Anomaly detection

The tasks of determining whether a certain internet behaviour is fraudulent or not, whether an aircraft engine is about to fail, and whether a product out of production is okay or not all have in common that in them the question is asked whether a new data point is much different than many other data points in a huge set of regular data samples. One assumes that the data set is created by a normal distribution with mean μ and variance σ^2 and tries to estimate these parameters from the initial data set

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}^{(i)}, \tag{2.7}$$

$$\sigma^2 = \frac{1}{m-1}\sum_{i=1}^{m} \mathrm{diag}\left(\left(\boldsymbol{x}^{(i)} - \mu\right)^{\intercal}\left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}\right)\right). \tag{2.8}$$

Should the probability

$$p(\boldsymbol{x}) = \exp\left(-\sum_{i=1}^{m}\frac{(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}}{2\sigma^2}\right)\frac{1}{\sqrt{2\pi}\det(\sigma)} \tag{2.9}$$

be smaller than a threshold value, one flags an anomaly and else does not. The algorithm can be evaluated the following way

1. assume to have **labelled** data being anomalous and normal
2. build training set containing only normal data
3. build test set with both types
4. build cross-validation set with as many anomalous data as in the test set which should not be the same

Possible evaluation metrics are

- true positive, true negative, false positive, false negative
- precision, recall
- F_1-score

The test set can be used to choose a value for the threshold value.

Comparison between anomaly detection and supervised learning

anomaly detection	supervised learning
• very small number of positive examples ($y = 1$), commonly 0 to 20.	• large number of positive and negative examples
• large number of negative examples ($y = 0$) $\Rightarrow$ skewed classes	• enough positive examples for the algorithm to get a sense of what positive examples are like and where future positive examples likely are similar to ones previously encountered in the training set
• many different types of anomalies for which it is hard to learn for any algorithm how the anomalies look like from the positive examples and where future anomalies might look very different to past anomalies	

The features should be selected in the following way:

- check whether data are really Gaussian; if they are not, try a transformation of the features
- use features with a large probability for normal examples and a small one for anomalous examples
- evaluate the results of test set by asking yourself what can be learned from the negative examples and then design new features
- choose features with extremely large or small values in anomalous events
- one can also think about using multivariate Gaussian distribution to model the likelihood of a test example being an anomaly:

$$p(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}\det\Sigma^{1/2}} \exp\left(-\frac{1}{2}\left(\boldsymbol{x}-\boldsymbol{\mu}\right)^{\intercal}\Sigma^{-1}\left(\boldsymbol{x}-\boldsymbol{\mu}\right)\right), \tag{2.10}$$

$$\Sigma = \frac{1}{m-1}\sum_{i=1}^{m}\left(\boldsymbol{x}^{(i)}-\boldsymbol{\mu}\right)\left(\boldsymbol{x}^{(i)}-\boldsymbol{\mu}\right) \tag{2.11}$$

  This way new features are already generated from the old features, but the procedure is computationally more expensive and one has to have (much) more training examples than features and no redundant (linear dependent) features.

2.4 Recommender systems

In movie and product recommendation and other applications, one asks oneself whether given a set of feature values allows one to suspect the value of other features. In the following, we denote by n_u the number of users or data sets, with n_m the number of products or features, with $r(i,j)$ a rating flag which is zero if feature i has no value in dataset j and is one otherwise and with $y(i,j)$ we denote the rating given by the user (value in the data set) j to the movie/product (feature) i. The task is to predict the value for an unrated feature and to recommend high valued unrated features to the user. There are two algorithms used to solve this issue. Both are combined to attain the best results.

2.4.1 content-based recommender systems algorithm

In this algorithm, one has class labels $x^{(i)}$ with values between zero and one for each element in the data set (movie). For zero the element does not belong to the class at all and for one it does so totally. The class can be for example a genre of the movie. One then learns a for each user j a parameter $\theta^{(j)} \in \mathbb{R}^3$ that predicts that the user would rate the movie i with $\theta^{(j)}x^{(i)}$. Each user has a parameter vector $\boldsymbol{\theta}^{(j)}$ and each movie a feature vector $\boldsymbol{x}^{(i)}$. The objective is to find

$$\min_{\{\boldsymbol{\theta}^{(i)}\}} \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left(\left(\boldsymbol{\theta}^{(j)}\right)^{\intercal}\cdot\boldsymbol{x}^{(i)} - y(i,j)\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n_m}\left(\theta_k^{(j)}\right)^2, \tag{2.12}$$

e.g. by gradient decent. The second sum in the objective runs over all rated elements of the data set (movies) by the user.

2.4.2 collaborative filtering algorithm

In this algorithm, the features are learned and there is no need for class labels. Instead the preference from other users is used to learn the values for the unrated features, if both have similar ratings in their rated elements of the data set (movies). The objective is in this case

$$\min_{\{\boldsymbol{x}^{(i)}\}} \frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}\left(\left(\boldsymbol{\theta}^{(j)}\right)^{\intercal}\cdot\boldsymbol{x}^{(i)} - y(i,j)\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n_u}\left(x_k^{(i)}\right)^2. \tag{2.13}$$

2.4.3 Complete recommender system

Collaborative filtering can estimate the user's preferences $\theta^{(u)}$ by learning the class labels (values for them) of a given element in the data set (product/movie). These preferences can be the degree of romance, action or the like. Content-based recommendation learns the ratings and determines the guess for a reasonable rating of before unrated movies. Both are combined to optimise the result, i.e. one guesses $\boldsymbol{\theta}$, then calculates $\boldsymbol{x}$, and guesses $\boldsymbol{\theta}$ again. The objective is then

$$\min_{\{\boldsymbol{x}^{(i)}\}} \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left(\left(\boldsymbol{\theta}^{(j)}\right)^{\intercal} \cdot \boldsymbol{x}^{(i)} - y(i,j) \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n_u} \left(x_k^{(i)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n_m} \left(\theta_k^{(j)}\right)^2 . \tag{2.14}$$

The best procedure is to

1. initialise $\boldsymbol{x}$ and θ to small random values
2. minimise the objective function using gradient decent or an advanced optimisation algorithm
3. for a user with parameters θ and a rated class (movie) with the feature $\boldsymbol{x}$, predict a rating of $\theta^{\intercal}\boldsymbol{x}$

2.5 Large scale machine learning

Large data sets obtain much better results compared to smaller ones, i.e. one can improve ¨inferior¨ algorithms[1] so that they perform better than ¨superior¨ algorithms with the same amount of data. However, there can be computational problems when huge data sets are used, i.e. there are many operations for a single learning step. One can ask oneself then whether it is necessary at all to use all available data to train. The learning curves introduced in section 1.6 can be used as indicators whether there is a high variance with a small training set. Other methods exist still. The most helpful are introduced in the following:

- **Stochastic gradient descent:** In contrast to batch gradient descent where all available data are used to make one step, one can make one step for each randomly picked training example. The update rule for the parameters is then

$$\theta_j = \theta_j - \alpha \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}. \tag{2.15}$$

  Every gradient decent step is faster, but it also optimises the cost solely for a single training example. On average, hence the word stochastic, the algorithm might converge still.

- **Mini-batch gradient descent:** Here one uses only a subset b of the training set in each iteration. One gets b examples and then does the gradient descent step with the averaged gradient:

$$\boldsymbol{\theta} = \theta_j - \alpha \left\langle \nabla S \right\rangle_b . \tag{2.16}$$

  The algorithm is much faster than batch gradient descent and more accurate than stochastic gradient descent[2]. The downside is that there is another parameter to optimise the algorithms performance with b.

- **Online learning:** When the training data are continuously flowing in over time, the model can learn on the fly. One simply does a stochastic gradient descent step in for each new input (or averages over several). This enables the algorithm too to adapt to changes in the user's preferences.

[1] w.r.t the performance with a smaller data set

[2] ..and faster with good vectorisation

- **Map-reduce approach:** Here one splits the learning to several machines doing n mini-batches on n machines and recombines the resulting gradients before a learning step, does the step and repeats. This method works if the learning algorithm has the bulk of the work given by a summation of functions over the training set an allows for a parallelisation on a cluster (one can use tools as Hadoop)

- **Artificial data synthesis:** This method supports real data with ¨made-up¨ data, e.g. using computer fonts for character recognition. Additionally, one can perform other operations as blurring, distortions, rotations or using different backgrounds on real or artificial data to multiply the size of the data set even further. Noise can be superimposed as well, but it usually does not help, if it is meaningless. Before getting more data, one has to make sure to really have a low bias classifier or keep increasing the number of features or hidden units in the neural network until one has one. only if it is impossible to get new real data, one should create artificial data. A good question to ask is: ¨How much work would it be to get ten-times as much data as one currently has?¨. Another option to get new data is to label and collect them by oneself or use crowd sourcing, e.g. amazon mechanical Turk).

Part II

Reinforcement Learning

Chapter 3

Fundamentals of reinforcement learning

Reinforcement learning (RL) can solve problems with less programming work than other learning algorithms and it is the most promising approach to realise general artificial intelligence so far. In contrast to other machine learning algorithms:

- There are no labelled data as in supervised learning, instead a reward function gives an agent an idea of the found solution's quality.
- The purpose is not to find the underlying structure in the data like in unsupervised learning, although it can aid the generalisation too.
- It is about learning while interacting with a complex, ever-changing world. The agents are expected to get things wrong, but to learn from their mistakes in the best case scenario.
- Online learning is very powerful in these environments and an defining feature of RL.

RL can help to make manufacturing and other industrial setups more efficient and aid experts in their decision finding. Hence, several different costs are reduced. Because RL has no correct actions, it is essential to explore the state space to search for good behaviour. The feedback the agents gets is evaluative instead of instructive and it is depending on the action taken. Instructive as well as evaluative feedback can be combined too.

3.1 k-armed bandit problem

The **k-armed bandit problem** is a good test bed for any RL algorithm. This learning problem is defined by the following characteristics:

- The agents is repeatedly faced with a choice among k-options or actions
- After its decision, it receives a numerical reward chosen from a stationary probability distribution depending on the selected action.
- The objective is to maximise the expected total reward over some time period.

One denotes the action taken at time t by A_t and the reward received afterwards by R_t. Additionally, the value of an arbitrary action a is denoted by $q_*(a)$ and by definition it is the expectation value of the overall reward received at time t if action A_t is a:

$$q_*(a) = \langle R_t | A_t = a \rangle. \tag{3.1}$$

In reality, this expectation value can only be estimated by several observations over a long time. The estimation of an actions value at time t is denoted by $Q_t(a)$ and in the best case $\lim_{t\to\infty} Q_t(a) = q_*(a)$ and the convergence is sufficiently fast.

Greedy actions are those actions that have the highest estimated reward. By selecting such actions, the current knowledge of the actions' values is exploited. They are the correct way to maximise the reward at one time. **Non-greedy actions** are all actions which are not greedy actions. By selecting these actions, one explores the action space and improves ones knowledge about all the actions' values in the action space. While **exploitation** is the right thing to maximise the expected reward momentarily, **exploration** may produce better results in the long run as better actions can be exploited more often if they are found to be the actual greeds actions by exploration. It is impossible to explore and exploit in one action. Hence there has to be a trade-off between both. The uncertainty of the actions' estimates, their estimates themselves, and the number of remaining steps decide whether it is better to explore or exploit in a specific situation. There are mathematically sophisticated methods for finding the optimal action for a given time. In reality, their requirements and assumptions are violated often times[1]. Balancing exploration and exploitation is a distinctive challenge in reinforcement learning.

3.1.1 Action value methods

Action value methods are used to estimate the value of actions and use these to make action selection decisions. Naturally, the value of an action is estimated as

$$Q_t(a) = \frac{\sum_{i=1}^{t-1}}{N_a} \tag{3.2}$$

with N_a being the number of times action a was chosen before time t. If the action was never chosen, the estimated value of the action is a customary default c. As t approaches infinity, the estimation becomes more realistic and tends to the true value for the right selection process which is usually called a **policy**. A memory-efficient way to update the action estimator is

$$\begin{aligned} Q_{n-1} &= \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) = \frac{1}{n}\left(R_n + \frac{(n-1)}{(n-1)}\sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\left(R_n + (n-1)Q_{n-1}\right) \simeq \frac{1}{n}\left(R_n + nQ_n - Q_n\right) \\ &= Q_n + \frac{1}{n}\left(R_n - Q_n\right). \end{aligned} \tag{3.3}$$

The term in brackets in the last line indicates the error in the estimation. In general, i.e. when more sophisticated estimates for the value are used, the update rule for the estimator is

$$Q_{\text{new}} = Q_{\text{old}} + \text{step size}\,(\text{target} - Q_{\text{old}}) \tag{3.4}$$

where the target indicates a desirable direction in which to move. The step size has the same function as the learning rate in supervised learning algorithms. It can be time-dependent as well as action-dependent. Usually such an update rule is a good approach for stationary bandit problems. For non-stationary processes, it makes sense to give more weight to recent rewards than to long-past rewards, i.e. to use a constant step size parameter between zero and one. The equation then becomes

$$Q_{n+1} = Q_n + \alpha\left[R_n - Q_n\right] = (1-\alpha)^n Q_1 + \alpha\sum_{i=1}^{n}(1-\alpha)^{n-i}R_i \tag{3.5}$$

the weights sum to one (geometric sum). This approach is sometimes called **exponential recency-weighted average**. To ensure convergence with probability one, the steps need to be large enough to overcome any initial conditions or random fluctuations, as well as small enough

[1] One of these assumptions being stationarity of the reward distribution

to ensure convergence, i.e.

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty, \tag{3.6}$$

$$\sum_{n=1}^{\infty} \alpha_n^2(a) < \infty, \tag{3.7}$$

respectively. Both conditions are met by sample average case $\alpha_n(a) = \frac{1}{n}$, but not by constant step size parameter $\alpha_n(a) = \alpha$. As a result, the estimates vary due to the most recently received rewards. Therefore, one should use sample average for stationary k-bandit problems and use exponential recency-weighted average for non-stationary problems.

3.1.2 Simple optimisation strategies

The so-called **exploration/exploitation trade-off** can be approached by several different techniques. Here is a short selection:

- **optimistic initial values:**The initial action-value estimates $Q_0(a)$ can change the outcome of the used method, i.e. the method is biased by their initial estimates. Prior knowledge about the level of rewards to expect can be incorporated by the user into the initial estimates. But these initial conditions can be used to encourage exploration as well. By setting all the estimates to an very optimistic value, all actions are tried several times before the estimates converge. Even when only greedy actions are taken, the agents still does a fair amount of exploration. Such a choice of optimistic initial values, is a good way to encourage exploration at early times, but not later times. Therefore, for stationary problems this method is well suited, but for non-stationary problems, it is no solution to the exploration/exploitation trade-off.

- **Unbiased constant step size trick:** For this trick, one uses a step size of

$$\beta_n = \frac{\alpha}{O_n} \text{ with } O_n = O_{n-1}(1-\alpha) + \alpha \text{ and } O_0 = 0. \tag{3.8}$$

  This leads to an exponential recency-weighted average without bias through initial values, because weights of bast rewards fade and the overall weights still go to zero

- **upper-confidence-bound action selection:** Here the idea is to select actions according to their potential of actually being optimal, i.e.

$$A_t = \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right] \tag{3.9}$$

  with $N_t(a)$ being the number the action a has been selected before time t. The parameter c determines the degree of exploration. This idea is hard to extent beyond the k-bandit problem and problematic for large state spaces and non-stationary processes.

3.2 Finite Markov decision processes

In Markov-decision-processes (MDP's) one estimates the value $q_*(s, a)$ of each action a in each state s or value of each state given optimal action selection $v_*(s)$. The most useful reinforcement learning techniques based on the concept of an MDP. The terminology used in an MDP is conceptually the same as in general reinforcement learning (see Figure 3.1):

- learner/decision maker is called an **agent**
- the thing the agent interacts with is called the **environment** which is giving him the new state and an reward

In a MDP a **trajectory** of state-action-reward-pairs is formed over time. The probability for getting the reward r and being in state s' after taking action a while being in state s is denoted by $p(s', r|s, a)$. With it, the whole dynamic of an MDP is defined. A system is said to posses the **Markov property** when $p(s', r|s, a)$ determines the whole dynamic of the environment. From this information on e can then calculate every statistical property of such a system, e.g.

- **state transition probability:**

$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{3.10}$$

  the sum runs over the whole reward space $\mathcal{R}$

- **expected reward of state action pairs:**

$$r(s, a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a) \tag{3.11}$$

  the sums run over the whole reward space $\mathcal{R}$ and state space $\mathcal{S}$, respectively.

- **reward of state-action-next-action-state triple:**

$$r(s, a, s') = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{3.12}$$

The goal of any reinforcement learning algorithm is to maximise the cumulative reward and to take actions that do so. Only the trajectory should be used by the agent to decide what action to take next in a given situation. There are no meta heuristics, e.g. in a game of chess, the agent should only get a reward for winning and not for dominating the center or taking opponents pieces. Else the agent might find another way to maximise the reward and not play to win the game.

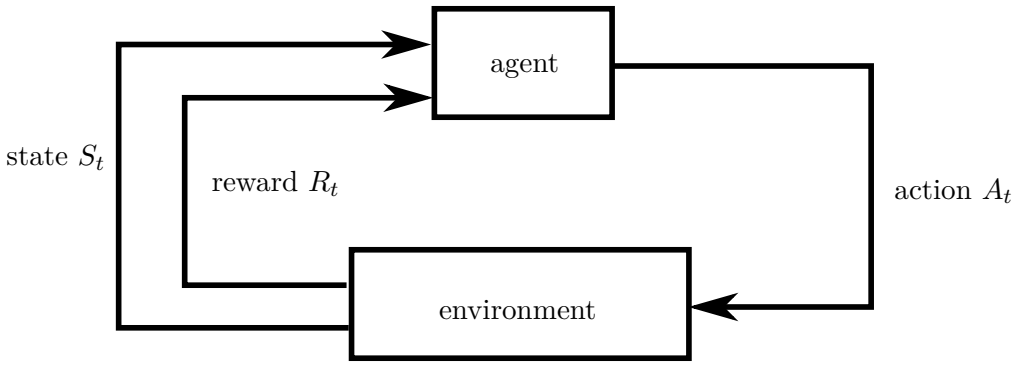


Figure 3.1: Sketch of the most important parts of a Markov-decision-process

There are two types of applications:

- those with a final or **terminal step** (such as chess) and many episodes of interactions of the agent with the environment are called **episodic tasks**
- those where the agents can interact indeterminately with the environment and where this interaction can not be broken down into episodes are called **continuing tasks**

A helpful quantity in both scenarios is the **discounted reward**

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1} \tag{3.13}$$

where γ is the so-called **discount rate**. For $\gamma = 0$ the agent is myopic and only tries to maximise R_{t+1} and nothing else and for $\gamma = 1$ it is far-sighted and might not care at all to optimise the reward in a certain situation. For episodic tasks, the final state is just a state from which one can only get reward zero and transition solely into this final state. Discounting should be used as a measure for the reward.
There are special forms of RL and techniques that should be mentioned at this place for the interested reader:

- In **inverse RL**, one observes the agent's behaviour and tries to reconstruct the reward function that was optimised in that given setting. This reward is then taken and applied to different circumstances or settings to yield good results.
- In **multi-agent RL** Several agents can be used if the correct behaviour (e.g. winning noble prizes) is hard to replicate. Agents that perform best are kept and duplicated. Successful agents pass on their knowledge to their offspring, while unsuccessful agents die out.

As a side note: maximising the the reward functions should be seen as a good approximation to achieve a goal. Finding the true optimal behaviour is often impossible, but also not necessary as the approximation is good enough already.

3.3 Policies and value functions

Almost all reinforcement learning algorithms involve estimating **value functions** of states which describe/estimate the how good it is for the agent to be in said state or perform a certain action when the measure is the expected future reward. The amount of reward is determined by the actions taken by the agent not just momentarily, but over many decisions by the agent. The way the agent acts is called a **policy** $\pi(a|s)$, i.e. the probability that the action $A_t = a$, if the state is $S_t = s$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience. The expected reward in the next time-step R_{t+1} if the current state is $S_t = s$, is

$$\langle R - t + 1 \rangle_{S_t=s} = \sum_{a\in\mathcal{A}} \pi(a|s) r(a|s) = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a). \tag{3.14}$$

The value function of a state s under a policy π, is the expected return starting in s and following π thereafter. For MDP's

$$v_\pi(s) = \langle G_t \rangle_{\pi,S_t=s} = \left\langle \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right\rangle_{\pi,S_t=s} \tag{3.15}$$

which is sometimes referred to as the **state value function** for policy π. Similarly, the value which is assumed taking action a while being in state s and thereafter following π is

$$q_\pi(s, a) = \langle G_t \rangle_{\pi,S_t=s,A_t=a} \tag{3.16}$$

called the **action-value function**. The following helpful relations can easily be derived

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a), \tag{3.17}$$

$$q_\pi(s,a) = \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]. \tag{3.18}$$

The values of $v_\pi(s)$ and $q_\pi(s,a)$ can be estimated by experience by bookkeeping of average received rewards following π while starting in state s and the rewards received performing action a in state s, respectively. Those are the so-called **Monte Carlo methods** for value function approximation. It is often not feasible to keep separate average for each state individually. One way is to instead maintain v_π and q_π as parametrised functions with less parameters than states and to adjust the parameters so that the functions match the observed returns. The accuracy of the estimates depends strongly on the parametrised approximator.
There are extremely powerful recursive relations that can be utilised:

$$\begin{aligned} v_\pi(s) &= \langle G_t\rangle_{\pi,S_t=s} = \rangle R_{t+1} + \gamma G_{t+1}\rangle_{\pi,S_t=s} \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \\ &= \sum_{a,s',r} \underbrace{\pi(a|s)p(s',r|s,a)}_{p(s|a,s',r)}\left[r + \gamma v_\pi(s')\right] \end{aligned} \tag{3.19}$$

This is the famous **Bellmann equation** (BE) for the value function $v_\pi(s)$ which expresses a relationship between the value of a state s and the value of its successor states s'. Here $v_\pi(s)$ is the unique solution of the BE. Hence the BE can be used to learn v_π.
For the action-value function $q_\pi(s,a)$ there is a similar version of the Bellmann-equation

$$\begin{aligned} q_\pi(s,a) &= \langle G_t\rangle_{\pi,S_t=s,A_t=a} = \rangle R_{t+1} + \gamma G_{t+1}\rangle_{\pi,S_t=s} \\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\right] \end{aligned} \tag{3.20}$$

Backup diagrams are used to illustrate the relationships that are the basis of the update or backup operations forming the heart of RL methods. The two versions of the Bellmann-equation encountered so far are shown in Figure 3.2.

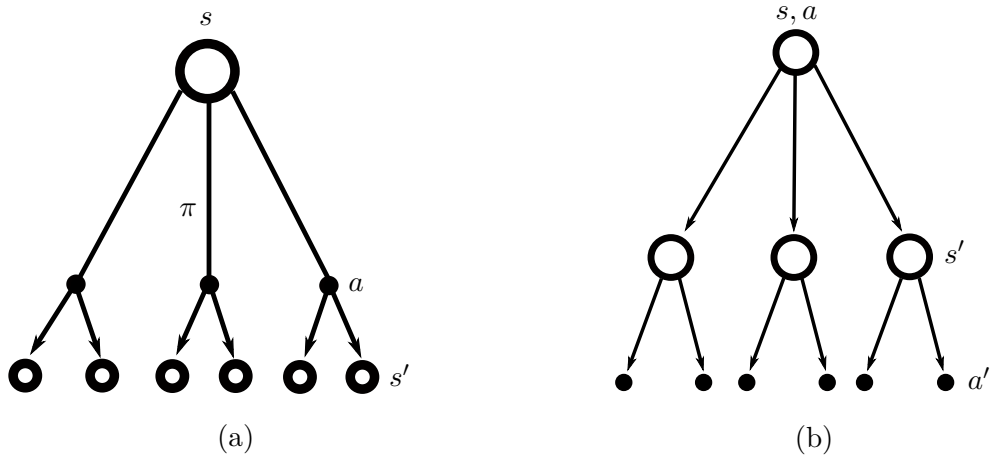


Figure 3.2: Backup diagrams for (a) the value function $v_\pi(s)$ and (b) the action-value function $q_\pi(s,a)$

A policy $\pi_*(a|s)$ is called **optimal** if it maximises the reward, i.e. $\pi(a|s) \geq \pi(a^\pi|s)$ only when $q_{\pi_*}(a,s) \geq q_{\pi_*}(a',s)$. The resulting state value function is called the **optimal state value function** $v_*(s) = \max_\pi v_\pi(s)\ \forall s \in \mathcal{S}$. Similarly the optimal action-value-function is

$$q_\pi(s,a) = \max_\pi q_\pi(s,a)\ \forall s,a \in \mathcal{S},\mathcal{A}, \tag{3.21}$$

$$= \langle R_{t+1} + v_*(S_{t+1})\rangle_{S_t=s,A_t=a}. \tag{3.22}$$

The **Bellmann optimality equations** are

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right], \tag{3.23}$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]. \tag{3.24}$$

This means that if the transition probabilities $p(s',r|s,a)$, i.e. the dynamics of the system, are known, one can solve the Bellmann equation for all states using any methods for solving systems of non-linear equations. Having $v_*(s)$ it is easy to select the optimal policy. One simply selects the actions on the state with the highest value function if there is more than one. This is the greedy policy with respect to the optimal evaluation function $v_*(s)$. Having the the optimal action-value-function $q_*(s,a)$ is even easier as no one step ahead search is required, i.e. no knowledge about the environments dynamics is necessary. It is, therefore, preferable to calculate $q_*(s,a)$ instead of $v_*(s)$ if the memory is no issue.
Most of the time it is not feasible to solve the optimal Bellmann equation as it is similar to doing an exhausted search of the state and probably the reward spaces. The assumptions are

1. Dynamics of the environment are accurately known
2. Solution of the BE's can be done at the computers at hand
3. Markov property

In RL one has to settle for approximate solutions, e.g.

- heuristic search methods which expand the right side of the BE to a given depth and approximate v_* at the leave nodes of the backup diagrams
- dynamic programming

3.4 Dynamic programming

Dynamic programming (DP) is a method/collection of algorithms that can be used to determine optimal policies given a perfect model of the environment as a MDP. Usually, the computational expense is huge, but all other approaches can be viewed to achieve the same goal/effect as DP, but with less computations and without assuming a perfect knowledge of the environment's dynamics. The key idea is to use the BE to find good approximations for the value functions and obtain good policies.
To achieve this, one is using **policy evaluation**. The BE can be seen as an iterative equation with the update rule

$$v_{k+1}(s) = \sum_s \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right] \tag{3.25}$$

and $v_*(s)$ is the fixed point of that equation as it fulfils the BE. If one does this for every state, all state-value functions are updated. This is called an **expected update**. The recursive evaluation works for the state-action functions too

$$q_{k+1}(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_k(s',a')\right]. \tag{3.26}$$

A faster convergence is achieved by updating the value functions as soon as you can, i.e. one does not wait until all other states are evaluated again. This procedure is called **in place update**. Iteration can be stopped when the value functions do not change much any more.
Given a good estimation of $v_\pi(s)$ (or $q_\pi(s,a)$) one can improve ones policy $\pi(a|s)$ by deviating from it by using π'. If $q_\pi(s,\pi'(s)) \geq v_\pi(s)$ one can use the new policy which is always better, i.e. one makes the policy greedy with respect to v_π. In the best case scenario the policy can not be improved any more

$$v_{\pi'}(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_{\pi'}(s')\right] \tag{3.27}$$

which is the optimal BE. This is called **policy improvement (control)**.
By repeating policy evaluation and policy improvement one obtains the optimal policy π_* given enough iterations. But all states need to be evaluated in the policy evaluation as well as improved. Hence the computational costs are high. In **value iteration**, one only uses one sweep for policy evaluation, combining evaluation and improvement step:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]. \tag{3.28}$$

One stops when $(v_{k+1} - v_k) \leq \epsilon_{\text{thr}}$ where ϵ_{thr} is a small threshold value. Still the complete state set is involved. Asynchronous DP algorithms update value functions in no order, e.g. value of the state $k\%N_S$ in step k and the computation can be intermixed with real-time interaction. Hence, these algorithms allow one to focus the updates onto parts of the state set that are most relevant to the agent which is a repeated theme in RL.
Finally, in the so called **generalised policy iteration** (GPI) the policy iteration has two simultaneous, interacting processes

1. making the value function consistent with the policy
2. make the policy greedy with respect to the current estimation of $v_\pi(s)$ and $q_\pi(s,a)$

As long as all states are updated, repeating both steps leads to convergence to the optimal value function and policy. Both processes ¨pull¨ in different directions. A greedier policy leads to different value functions, while making the value functions consistent with the policy results in a less greedy policy. Only when both purposes coincide, the optimal solutions are found and the BE is fulfilled, i.e. the single joint solution is the optimal solution. The process of updating the estimates on basis of other estimates is called **bootstrapping** and is often used in RL.

Chapter 4

Sample based learning methods

4.1 Monte-Carlo methods

Monte-Carlo (MC) methods are methods for the evaluation of state value functions and state-action-value functions without prior knowledge of the environment, because they learn from experience. Sample averages of states, actions, and rewards from actual or simulated interactions with the environment are used. Mostly the tasks are considered episodic for MC-sampling and one changes the value estimates and policies after each completion of an episode. Hence, MC method can be incremental in an episode by episode sense, but not in a step-by-step sense (online) sense. MC methods sample and average returns. The returns received after taking an action in one state depend on the actions taken in later states in the same episode. All action selections are undergoing learning and therefore the problem is a non-stationary one from the earlier state's point of view. Mc methods adapt the ideas from GPI, but instead of computing value functions from knowledge of the MDP, one learns the value functions and corresponding policies from sampled returns with the MDP. Value functions and policies still interact to find the optimal solution.

4.1.1 Monte Carlo prediction

The goal is to learn the value function v_π for a given policy π. This is done by average cumulative discounted reward which is the expected return starting from state s. There are two ways for this:

1. **first visit averaging:** where one calculates the average return after first visits to state s until the end of the episode (first visit MC method)

2. **every visit averaging:** where one calculates the average return following every visit to state s (every visit MC method)

The every visit MC method is more natural for function approximation and eligibility traces. Here is the algorithm:

- input is policy π
- initialise $v(s)$ arbitrarily and returns from all states as a list of zeros
- loop over episodes generating a trajectory of $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ following π
- set $G = 0$
- loop over each step of an episode from the termination time T backwards to the starting time at $t = 0$ and calculate

$$G = \gamma G + R_{t+1} \tag{4.1}$$

- for first visit MC method: unless S_t appears in the trajectory before time t, append G to returns of S_t and $v(S_t) = \langle \text{returns}(S_t) \rangle$

  for every visit MC method: append G to returns of S_t and $v(S_t) = \langle \text{returns}(S_t) \rangle$

Both algorithms give a $v(s)$ that converges to $v_\pi(s)$ as the number of episodes goes to infinity. While for DP algorithms the probabilities $p(s', r|s, a)$ need to be known beforehand, for MC algorithms only $p(s'|s, a)$ need to be known which is often much simpler for games as well as in reality. MC uses independent estimates of state values. Therefore, no bootstrapping is required and the computational expense of estimating the value function is independent of the number of states. One can start an episode with a subset of states or just one in whose v_π one is interested and perform the averaging. This is a huge advantage over DP methods.

4.1.2 Monte Carlo estimation of action values

If a model is unavailable, estimating the action values or values of state-action pairs is particularly useful as state-values use the model to calculate the value of an action and suggest a policy which is often not possible or desirable. The primary goal of MC methods is, therefore, the estimation of q_*. For this the plan is:

- First evaluate policy for action values, i.e. first estimate $q_\pi(s, a)$
- Calculate values for first visit or every visit state-action-pairs
- Because this is computationally more expensive as every state-action pair needs to be visited in order to be evaluated, one should start with a state-action-pair initialisation and give a finite probability to every action in every state to ensure more visits (or to use optimistic values)

In general this assumes exploring starts which is not possible for real interactions with the environment all the time and one uses a stochastic policy with finite weight on all state-action pairs instead.

4.1.3 Monte Carlo control

The idea of MC control is the same as for GPI:

- begin with policy evaluation of $q_\pi(s, a)$
- do policy improvement by favouring actions with highest value from a state

$$\pi'(s|a) = \underset{a}{\text{argmax}}\, q_\pi(s, a) \tag{4.2}$$

- repeat policy evaluation of the new policy π' and improve the policy to π'' and so on

This, however, takes too much time and episodes to gain a good estimate for $q_\pi(s, a)$ and it is advantageous to combine policy evaluation and improvement. After each episode, the observed returns are used for policy evaluation and as a result, the policy is improved for all states visited in that episode. One calls this approach **Monte Carlo with exploring starts** (MCES). The algorithm is the following:

initialise $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$, $Q(s, a) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(a)$, and returns$(s, a)$ to zeros for all $s \in \mathcal{S}, a \in \mathcal{A}(a)$

loop for each episode and choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that $p(s, a) \neq 0 \, \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$

generate an episode with a trajectory following π

set $G = 0$ and loop backwards through the episode calculating

$$G = R_{t+1} + \gamma G \tag{4.3}$$

and unless the pair appeared before, append G to the returns of the pair S_t, A_t, set

$$Q(S_t, A_t) = \langle \text{returns}(S_t, A_t) \rangle \tag{4.4}$$

using constant step size parameter or unbiased constant step size trick, and finally

$$\pi(S_t) = \underset{a}{\text{argmax}}\, Q(S_t, a) \tag{4.5}$$

4.1.4 Monte Carlo control without exploring starts

For real interactions with the environment, the assumption of exploring starts is often not given. For a good evaluation the agent has to select all state-action-pairs with a certain finite probability. There are two ways:

1. **Off-policy methods** evaluate or improve a policy different from that used to generate the data
2. **On-policy methods** attempt to evaluate or improve the policy that is used to make decisions, e.g. MCES

For on-policy methods the policy starts stochastically soft $\pi(a|s) > 0 \forall s \in \mathcal{S}$ and becomes more deterministic in the improvement steps. A commonly used example is the ϵ**-greedy** property:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{N_a(s)} & , \quad a \neq \underset{a}{\text{argmax}}\, q_\pi(a, s) \\ 1 - \frac{\epsilon(N_a(s)-1)}{N_a(s)} & , \quad a = \underset{a}{\text{argmax}}\, q_\pi(a, s) \end{cases} . \tag{4.6}$$

Here $N_A(s)$ is the number of times the action a was selected when the agent has been in state s. Any ϵ-greedy policy with respect to q_π is better than any ϵ-soft policy π.
The off-policy methods have one policy π_t which is called the **target policy** which is to be optimised and a **behaviour policy** π_b which generates the behaviour, i.e. picks the actions in the environment. Learning from data off the target policy is called **off-policy learning**. Often off-policy methods have greater variance and are slower to converge, but are more powerful and general than on-policy learning methods. They can learn from data generated by a conventional non-learning controler or from a human expert. It is seen as a step towrds multi-step predictive models of the world's dynamics. In order to be able to use π_b for the evaluation of q_{π_t} and estimate values of π_t, π_b must at least occasionally take actions that π_t would take, i.e. if $\pi_\text{t}(a|s) \neq 0$ then $\pi_\text{b}(a|s) \neq 0$ which is called the **assumption of coverage**. The behaviour policy needs to be stochastic, while the target policy is often deterministic in control problems.
Almost all off-policy methods utilise **importance sampling** which is a method to evaluate/estimate expected values under one distribution given samples from another. The returns are weighted by their relative probability of their trajectory occurring under the target and behaviour policies, the **importance sampling ratio**

$$p(A_t, S_{t+1}, A_{t+1}, \ldots, S_T | S_T, A_{t:T-1} \propto \pi) = \prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k) \tag{4.7}$$

with the state transition probabilities $p(s_{k+1}|s_k, a_k)$ from state s_k into s_{k+1} if action a_k is taken. These relative probabilities of a trajectory under the target and behavioural policies are

$$p_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi_\text{t}(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} \pi_\text{b}(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi_\text{t}(A_k|S_k)}{\pi_\text{b}(A_k|S_k)} \tag{4.8}$$

which means that there is no dependence of the importance sampling ratio on the MDP's dynamic!
The goal was to estimate the expected returns under the target policy, i.e. one needs to determine $v_{\pi_\mathrm{t}}(s)$ from $v_{\pi_\mathrm{b}}(s)$ or $q_{\pi_\mathrm{t}}(s,a)$ from $v_{\pi_\mathrm{b}}(s,a)$. Obviously this is

$$\langle v_{\pi_\mathrm{t}}(s)\rangle_{S_t=s} = \langle p_{t:T-1}G_t\rangle_{S_t=s} \tag{4.9}$$

which for **ordinary importance sampling** is

$$v(s) = \frac{\sum\limits_t p_{t:T-1}G_t}{\tau(s)}, \tag{4.10}$$

with $\tau(s)$ being the number of times state s was visited and for **weighted importance sampling** it is

$$v(s) = \frac{\sum\limits_t p_{t:T-1}G_t}{p_{t:T-1}}. \tag{4.11}$$

If the denominators in both expressions are zero, the state-value vanishes too by definition. Weighted importance sampling yields biased estimates, but with a lower variance and ordinary importance sampling is unbiased but with a higher variance. The weighted estimator is strongly preferred, but ordinary importance sampling is easier to extent to function approximation. For state-action functions

$$\langle q_{\pi_\mathrm{t}}(s,a)\rangle_{S_t=s,A_t=a} = \langle p_{t+1:T-1}q_{\pi_\mathrm{b}}(s,a)\rangle_{S_t=s,A_t=a} \tag{4.12}$$

where the +1 is because one action has already been performed and $p(S_{t+1}|s,a)$ cancels in the numerator and denominator.
Suppose one has an estimate for $V_n(s)$ taken by evaluation of n episodes

$$V_n(s) = \frac{\sum\limits_{k=1}^{n-1} w_k G_k}{\sum\limits_{k=1}^{n-1} w_k} \tag{4.13}$$

where $w_k = p_{t_i:T-1}$ and gets another sample return G_n. Because

$$\begin{aligned} V_{n+1} - V_n &= \frac{\sum\limits_{k=1}^{n} w_k G_k}{\sum\limits_{k=1}^{n} w_k} - \frac{\sum\limits_{k=1}^{n-1} w_k G_k}{\sum\limits_{k=1}^{n-1} w_k} = \frac{\left(\sum\limits_{k=1}^{n-1} w_k\right)\sum\limits_{k=1}^{n} w_k G_k - \left(\sum\limits_{k=1}^{n} w_k\right)\sum\limits_{k=1}^{n-1} w_k G_k}{\left(\sum\limits_{k=1}^{n} w_k\right)\left(\sum\limits_{k=1}^{n-1} w_k\right)} \\ &= \frac{w_n G_n \left(\sum\limits_{k=1}^{n-1} w_k\right) - w_n \sum\limits_{k=1}^{n-1} w_k G_k}{\left(\sum\limits_{k=1}^{n} w_k\right)\left(\sum\limits_{k=1}^{n-1} w_k\right)} = \frac{w_n G_n}{\sum\limits_{k=1}^{n} w_k} - \frac{w_n V_n}{\sum\limits_{k=1}^{n} w_k} \end{aligned} \tag{4.14}$$

we can directly infer the update rule

$$\begin{aligned} V_{n+1} &= V_n + \frac{w_n}{c_n}\left(G_n - V_n\right) \\ \text{with } c_n &= \sum_{k=1}^{n} w_k. \end{aligned} \tag{4.15}$$

The same can be derived for the state-action-value estimates Q_{n+1}.

4.1.5 Off-policy Monte Carlo prediction (policy evaluation) for estimating $Q \approx q_\pi$

input: an arbitrary target policy π

initialise for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:

$Q(s,a) \in \mathbb{R}$ arbitrary
$C(s,a) = 0$

Loop for each episode

b any policy with coverage of π

Generate a trajectory following b

$G = 0$ and $w = 1$

loop for each step of the episode backwards in time while $w \neq 0$

$$
\begin{aligned}
G &= \gamma G + R_{t+1} \\
C(S_t, A_t) &= C(S_t, A_t) + w \\
Q(S_t, A_t) &= Q(S_t, A_t) + \frac{w}{C(S_t, A_t)} \left[G - Q(S_t, A_t) \right] \\
w &= w \frac{\pi(A_t|S_t)}{b(A_t|S_t)}
\end{aligned}
$$

4.1.6 Off-policy Monte Carlo control for estimating π_*

initialise for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

$Q(s,a) \in \mathbb{R}$ arbitrarily
$C(s,a) = 0$
$\pi(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$ (with ties broken consistently)

loop for each episode:

b any soft policy

Generate a trajectory using b and set $G = 0$, $w = 1$

loop for each step of episode backwards along trajectory:

$$
\begin{aligned}
G &= \gamma G + R_{t+1} \\
C(S_t, A_t) &= C(S_t, A_t) + w \\
Q(S_t, A_t) &= Q(S_t, A_t) + \frac{w}{C(S_t, A_t)} \left[G - Q(S_t, A_t) \right] \\
\pi(S_t) &= \underset{a}{\operatorname{argmax}}\, Q(S_t, a) \quad \text{(with ties broken consistently)}
\end{aligned}
$$

if $A_t \neq \pi(S_t)$ then exit the inner loop and proceed to next episode as a better strategy/policy was found (the importance sampling ration of trajectory becomes zero and, hence, nothing new can be learned

else $w = \frac{w}{b(A_t|S_t)}$

This algorithm only learns from the tails of episodes (later states of the trajectory) up to the greedy actions and early states are hardly learned from if greedy behaviour is uncommon in b. The result is a slower convergence/learning. **Temporal difference learning** can be helpful which is discussed in the next section. **Flat partial returns** are another possibility. Here discounting can be thought of as an early termination with probability $1-\gamma$ yielding a reward/return R_1. This is done for every time-step and the flat return is calculated

$$\overline{G}_{t:h} = R_{t+1} + R_{t+2} + \cdots + R_h \tag{4.16}$$

where h is the horizon with $0 \leq t < h \leq T$. Furthermore

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \\ &= (1-\gamma) R_{t+1} + (1-\gamma)\gamma \left(R_{t+1} + R_{t+2}\right) + (1-\gamma)\gamma^2 \left(R_{t+1} + R_{t+2} + R_{t+3}\right) \\ &\quad + \cdots + (1-\gamma)\gamma^{T-t-1} \left(R_{t+1} + \ldots R_T\right) \\ &= (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \overline{G}_{t:h} + \gamma^{T-t-1} \overline{G}_{t:T}. \end{aligned} \tag{4.17}$$

Weights should be used up to h to scale importance sampling

$$V(s) = \frac{\sum\limits_t \left((1-\gamma) \sum\limits_{h=t+1}^{T-1} \gamma^{h-t-1} p_{t:h-1} \overline{G}_{t:h} + \gamma^{T-t-1} p_{t:T-1} \overline{G}_{t:T} \right)}{\sum\limits_t \left((1-\gamma) \sum\limits_{h=t+1}^{T-1} \gamma^{h-t-1} p_{t:h.1} + \gamma^{T-t-1} p_{t:T-1} \right)}. \tag{4.18}$$

One can use **per decision sampling** too. Realising that

$$p_{t:T-1} G_t = p_{t:T-1} R_{t+1} + \gamma p_{t:T-1} R_{t+2} + \cdots + \gamma^{T-t-1} p_{t:T-1} R_T \tag{4.19}$$

and

$$p_{t:T-1} R_{t+1} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1} \tag{4.20}$$

one can suspect that only the first and the last factor are related and the expected value of all other is one

$$\left\langle \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \right\rangle_b = \sum_a b(a|S_k) \frac{\pi(a|S_k)}{b(a|S_k)} = \sum_a \pi(a|S_k = 1 \tag{4.21}$$

and one can derive the following results

$$\langle p_{t:T-1} R_{t+1} \rangle_b = \langle p_{t:t} R_{t+1} \rangle \tag{4.22}$$

$$\langle p_{t:T-1} R_{t+k} \rangle_b = \langle p_{t:t+k-1} R_{t+k} \rangle \tag{4.23}$$

$$\langle p_{t:T-1} G_t \rangle_b = \langle \overline{G}_t \rangle \tag{4.24}$$

where $\overline{G}_t = p_{t:t} R_{t+1} + \gamma p_{t:t+1} R_{t+2} + \cdots + \gamma^{T-t-1} p_{t:T-1} R_T$. This is the **per decision importance sampling**. For this procedure there is no version of weighted importance sampling to date for this.

4.2 Temporal-difference learning

Temporal-difference learning (TD) is a combination of Monte Carlo ideas and Dynamic programming ideas. TD methods update the estimates based in part on other learned estimates, without waiting for an outcome, i.e. they bootstrap, and they use raw experience to learn without a model of the environment's dynamics.

4.2.1 Temporal-difference prediction

In general the update rule for any methods is

$$V_{n+1}(S_t) = V_n(S_t) + \alpha \left[G_t - V(S_t)\right] \tag{4.25}$$

with G_t being the return after time t. While MC methods wait for the end of an episode to determine the increment to V_n [1], TD methods use the reward after t and state value of the next times state

$$V_{n+1}(S_t) = V_n(S_t) + \alpha \left[\underbrace{R_{t+1} + \gamma V_n(S_{t+1})}_{\text{TD(0) or one-step TD}} - V_n(S_t)\right]. \tag{4.26}$$

More generally later times states functions can be used, e.g. TD(λ) or $\lambda+1$-step TD. While DP samples $V_n(S_{t+1})$ as it is unknown, MC samples $\langle G_t \rangle_{S_t=s}$ as it is unknown, and TD samples both. A helpful quantity is the TD error δ_t as it allows to write the TD method as an approximation to the general form:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}), \tag{4.27}$$

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k. \tag{4.28}$$

TD methods are naturally implemented in an online fashion, i.e. fully incremental. Delaying the learning until the end like in MC is bad for long processes/episodes and even worse for continuing tasks. TD methods only need one time-step to learn and they need not ignore or discount episodes on which experimental actions are taken, because they learn from each transition regardless of what subsequent actions are taken. TD(0) is guaranteed to converge to v_π for any policy π if the step size parameter decreases according to the usual stochastic approximation conditions in equations (3.6) and (3.7). It has been observed that TD converges faster than MC in stochastic tasks. MC methods minimise the mean-square error on the training set, while TD methods minimise the maximum-likelihood estimate of the MDP[2] In **batch updating** one computes the increments $R_{t+1} + \gamma V_n(S_{t+1})$ for all elements/states in the training data and updates the approximate value functions by the sum of the increments.

[1] after which G_t is known

[2] The maximum-likelihood estimate of a parameter is the value whose probability of generating the data is greatest.

4.2.2 Algorithm of TD(0) for estimating v_π

input: the policy π to be evaluated

algorithm parameter: step size $\alpha \in (0, 1]$

initialise S

loop for each episode until S is terminal:

A =action given by π for S
take action A, observe R, S'
$V(S) = V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$
$S = S'$

In order to improve the policy use GPI.

4.2.3 SARSA: on-policy temporal-difference control

One might learn the state-action function instead of the state-value function:

$$Q_{n+1}(S_t, A_t) = Q_n(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q_n(S_{t+1}, A_{t+1}) - Q - n(S_t, A_t)\right] \tag{4.29}$$

This iteration is done after each transition from a non-terminal state S_t. If S_{t+1} is terminal, then by definition $Q(S_{t+1}, A_{t+1}) = 0$, i.e. $Q(T, a) = 0\ \forall a \in \mathcal{A}$. This update rule uses each element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ and is, therefore, called the **SARSA-algorithm**.

parameters: step size $\alpha \in (0, 1]$ and a small $\epsilon > 0$

initialise $Q(s, a)\ \forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ arbitrarily except for $Q(T, \cdot) = 0$

loop for each episode until S is terminal:

initialise S
choose A from S using policy derived from Q, e.g. ϵ-greedy
loop for each step of the episode:
take action A, observe R, S'
choose A' from S' using policy derived from Q
$Q(S, A) = Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$
$S = S'$
$A = A'$

MC methods can get stuck in episodes, because they might learn a policy to stay in one state or return to it periodically. Online learning methods like SARSA do not have that problem, because they learn during an episode that such a policy is poor and switch to something else.

4.2.4 Q-learning: off-policy temporal-difference control

A greedy version of SARSA is called **Q-learning** and has the update rule

$$Q_{n+1}(S_t, A_t) = Q_n(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q_n(S_{t+1}, a) - Q_n(S_t, A_t) \right]. \tag{4.30}$$

The learned action-value function directly approximates q_* independent of the policy being followed. The only influence of the policy is to determine which state-action pairs are visited and updated.
Here is the **Q-learning algorithm for estimation of** $\pi \approx \pi_*$:

parameters: step size $\alpha \in (0, 1]$ and a small $\epsilon > 0$

initialise $Q(s, a) \, \forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ arbitrarily except for $Q(T, \cdot) = 0$

loop for each episode until S is terminal:

initialise S

loop for each step of the episode:

choose A from S using policy derived from Q, e.g. ϵ-greedy
take action A, observe R, S'
$Q(S, A) = Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
$S = S'$
$A = a$

4.2.5 expected SARSA

If instead of using the greedy action like in Q-learning, one can use the expected value taking into account how likely each action is under the current policy. This is called **expected SARSA** and has the update rule

$$Q_{n+1}(S_t, A_t) = Q_n(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1} = Q_n(S_{t+1}, a) - Q_n(S_t, A_t) \right] \tag{4.31}$$

The algorithm moves deterministically where SARSA moves in expectation. Although expected SARSA is computationally more complex, it is a significant improvement over SARSA as it eliminates the variance due to random selection of A_{t+1}. Just the state that is updated is determined by the policy and is the only randomness. The step size parameter α can be set to one without any degredation of asymptotic performance. For a greedy π expected SARSA is Q-learning, but in principle expected SARSA performs better than simple SARSA and Q-learning.

4.2.6 Double learning

SARSA, Q-learning, and expected SARSA have positive maximation bias. There are methods that avoid this maximation bias. One such example is **double learning**. Here one uses two independent estimates $Q_1(a)$ and $Q_2(a)$ for the true value of $q(a) \, \forall a \in \mathcal{A}$. $Q_1(a)$ is used to determine the maximising action

$$A^* = \operatorname*{argmax}_a Q_1(a), \tag{4.32}$$

and $Q_2(a)$ is used to provide the estimate of its value

$$Q_2(A^*) = Q_2(\operatorname*{argmax}_a Q_1(a)). \tag{4.33}$$

Afterwards their role is reversed

$$Q_1(A^*) = Q_1(\operatorname*{argmax}_a Q_2(a)). \tag{4.34}$$

Each estimate is only updated once per two episodes[3]. The memory requirement is double the one of simple Q-learning, while the computational cost is the same. Here is the algorithm:

parameters: step size $\alpha \in (0, 1]$ and a small $\epsilon > 0$

initialise $Q_1(s, a), Q_2(s, a) \forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ randomly except for $Q_i(T, \cdot) = 0$

loop for each episode until S is terminal:

initialise S

loop for each step of the episode:

choose A from S using policy greedy in $Q_1 + Q_2$

take action A, observe R, S'

with probability 0.5:

$$Q_1(S, A) = Q_1(S, A) + \alpha \left[R + \gamma Q_2(S', \underset{a}{\operatorname{argmax}}\, Q_1(S', a)) - Q_1(S, A) \right]$$

else

$$Q_2(S, A) = Q_2(S, A) + \alpha \left[R + \gamma Q_1(S', \underset{a}{\operatorname{argmax}}\, Q_2(S', a)) - Q_2(S, A) \right]$$

$S = S'$

$A = a$

Sometimes it is better to use so-called **afterstates** as value functions which incorporate that only the state after the action has to be evaluated, not the state-action pair[4], e.g. in chess the same position can be reached from different action sequences, but their ¨worth¨ is determined only by the position after the move.

Actor-critic models are another way to deal with the problem of exploration. Here the value function is not used to select the actions, but can be used to learn parameters of a policy. Hence, in these both the policy and the value functions are learned as approximations[5].

[3] They are updated in turns to be precise.

[4] state-before-action-action-pair :)

[5] more on that after function evaluation is discussed

4.3 Planning and learning with tabular methods

A model is anything an agent can use to predict how the environment will respond to its actions. For **stochastic models** either the distribution function of the probabilities is known which is then called a **distribution model** with probability of all states $p(s', r|s, a)$, or just the transition probabilities of a sample are known which is then called a **sample model** with $p(s', r|s, a)$. Distribution models are stronger, because they can be used to produce samples. Most of the time it is easier, however, to produce a sample model than a distribution model, e.g. consider k-dice rolls where their sum is estimated: it is easy to produce a sample of the dice sum and much harder to calculate the probability of all possible sums. Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition and a distribution model generates all possible transitions weighted by their probability of occurring. Given a starting state and a policy, a sample model can produce an entire episode and a distribution model can generate all possible episodes and their probability. **A model is used to simulate the environment and produce simulated experience**.

- **Planning** is the function that maps a model to a policy
- **state-space-planning** is the search through the state space to find an optimal policy or an optimal path to a goal
- **plan-space-planning** is the search through the space of plans with operators transforming one plan into another, and value functions (if any are defined at all) are defined over the space of plans. This is hard to implement in a stochastic environment or a stochastic sequential decision problem

The general form of state-space planning is

$$\text{model} \rightarrow \text{simulated experience} \overset{\text{backups}}{\rightarrow} \text{values} \rightarrow \text{policy}$$

In general the learning from experience in MC and TD algorithms can be replaced by learning from model produced experience. By planning in small incremental steps, it can be interrupted or redirected at any given time with little wasted computation. The key requirement for all this is efficiently intermixing planning with acting and with **learning of the model**. Figure 4.1 shows a schematic illustration of such a process.
There is a distinction in RL into two categories:

- **direct RL:** all the methods discussed so far (MC,TD)
- **indirect RL:** refers to the estimate for the model being improved and, therefore, as a result the value functions/policies being updated

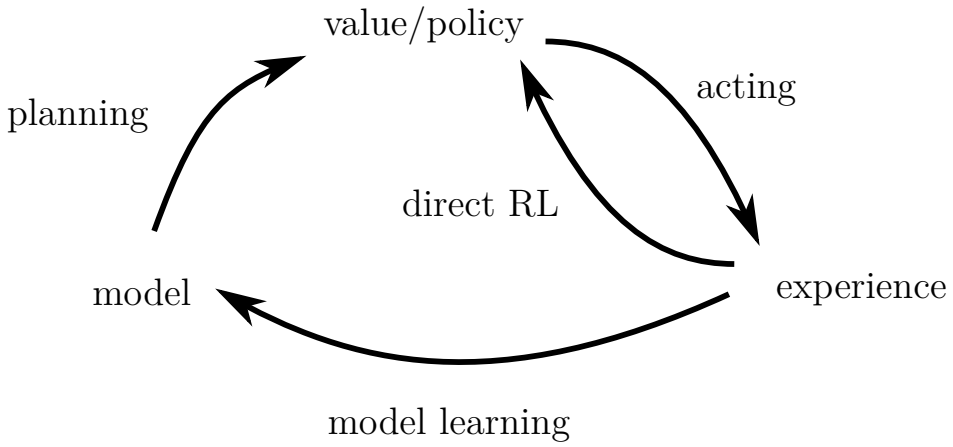


Figure 4.1: Illustration of an efficient learning process intermixing planning with learning

Chapter 5

Prediction and control with functions- approximate solution methods

In many applications, the state space is too large for an exhaustive search. A solution is to approximate the value function by similar states encountered before. In this sense, the function approximation is an instance of supervised learning and many methods that have been studied there can be utilised, but other issues such as non-stationarity, bootstrapping, and delayed targets arise.
The overall goal is to estimate the state-value function from (on-)policy data with a value function as a parametrised functional form with weights $\boldsymbol{w} \in \mathbb{R}^d$ such that $v_\pi(s, \boldsymbol{w}) \approx v_\pi(s)$, for the approximate value of the state s given weights vector $\boldsymbol{w}$. For example

- v can be a linear function in features of the state with $\boldsymbol{w}$ the vector of feature weights
- v can be a function computed by a multi-layered neural net with $\boldsymbol{w}$ being the weight of all connections in all layers

The number of weights should be much smaller than the number of states, i.e. changing one value of the weights should change the value of many states. When a single state is updated the change generalises from that state to affect the value of many other states. This is called the **generalisation property**. Generalisation makes learning potential more powerful, but also potentially more difficult to manage and understand.

5.1 Basics

5.1.1 Value function approximation

The estimated value function shifts its values at particular states towards a ¨backed-up value¨ or updated target for that state due to updates. In general this is denoted by $s \mapsto u$. Examples with before encountered quantities are

- MC update: $S_t \mapsto G_t$
- TD(0) update: $S_t \mapsto R_{t+1} + \gamma V(S_{t+1}, \boldsymbol{w}_t)$
- TD(n) update: $S_t \mapsto G_{t:t+n}$
- DP update: $s \mapsto \langle R_{r+1} + \gamma v(S_{t+1}, \boldsymbol{w}) \rangle_{S_t=s}$

This strong resemblance to supervised learning is no coincidence. In fact it is one and the same thing. Any method developed for supervised learning like artificial neural networks, decision

trees, and various kinds of multivariate multivariate regression can be used for function approximation in RL too. Some methods assume a static training set over which many passes are made over. These methods are not as well suited to be utilised in online learning where the agent learns while interacting with the environment or a model of it. Instead one should use methods that enable one to learn efficiently from incrementally acquired data and that can handle non-stationary target functions

5.1.2 The prediction objective $\overline{VE}$

In function approximation, the estimated value function can not be correct for all states, because values of the states are coupled[1]. Hence, one needs to specify an explicit objective for prediction. For this one uses the state distribution

$$\mu(s) \geq 0 \tag{5.1}$$

$$\sum_s \mu(s) = 1 \tag{5.2}$$

which should specify how much one cares about the error in each state, e.g. with mean square error as a measure. The natural objective function is the mean squared value error

$$\overline{VE} = \frac{1}{2} \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - v(s, \boldsymbol{w}) \right]^2 . \tag{5.3}$$

Often $\mu(s)$ is the fraction of time spent in a state s. Under on-policy training this is called the **on-policy distribution** which is the stationary distribution under π in continuing tasks. In episodic tasks (without discounting) one has

$$\eta(s) = h(s) + \sum_{s'} \eta(s') \sum_a \pi(a|s') p(s|s', a) \; \forall s \in \mathcal{S} \tag{5.4}$$

$$\mu = \frac{\eta(s)}{\sum_{s'} \eta(s')} \; \forall s \in \mathcal{S} \tag{5.5}$$

with $h(s)$ being probability of the episode starting in state s and $\eta(s)$ is the number of time steps spent in s on average in a single episode. When there is discounting, treat this as a termination and include it as an prefactor in the second term on the right hand side of $\eta(s)$.
The question arises whether $\overline{VE}$ is a good function to minimise in order to find the correct policy, i.e. the optimal policy. It might not be, but what other function should one use? The best value function minimising $\overline{VE}$ is not the best for selecting a better policy. The goal is, therefore, to find a global optimum so that $\overline{VE}(\boldsymbol{w}^*) \leq \overline{VE}(\boldsymbol{w}) \, \forall \boldsymbol{w}$. This is possible for linear approximation functions, but not for more complex such as neural networks and decision trees. The latter converge to a local minimum/optimum in the best case scenario and in the worst case scenario they diverge. One has to use **stochastic-gradient** and **semi-gradient methods** to minimise $\overline{VE}$. Here one observes a new example of $S_t \mapsto v_\pi(S_t)$ at time t and updates ones estimate of $v(s, \boldsymbol{w})$ by adjusting the weights

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_n - \frac{\alpha_t}{2} \nabla_{\boldsymbol{w}} \left[v_\pi(S_t) - v(S_t, \boldsymbol{w}_t) \right]^2 \\ &= \boldsymbol{w}_t + \alpha_t \left[v_\pi(S_t) - v(S_t, \boldsymbol{w}_t) \right] \nabla_{\boldsymbol{w}} v(S_t, \boldsymbol{w}_t) \end{aligned} \tag{5.6}$$

with

$$\lim_{t \to \infty} \alpha_t = 0, \quad \text{such that } \int \alpha_t \, \mathrm{d}t = \infty \text{ and } \quad \int \alpha_t^2 \, \mathrm{d}t \text{ is finite}$$

so that the method converges to a local optimum.

[1] the update does not need to be, though

In reality the update is not $S_t \mapsto v_\pi(S_t)$ but $S_t \mapsto U(S_t) = U_t$ and the update rule is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - v(S_t, \boldsymbol{w}_t)\right] \nabla_{\boldsymbol{w}} v(S_t, \boldsymbol{w}_t) \tag{5.7}$$

which only converges when U_t is an unbiased estimate, i.e. $\langle U_t \rangle_{S_t=s} = v_\pi(s)$ for each t. It is fine for MC which is a **true gradient method**, but is not guaranteed for targets that bootstrap which are **semi-gradient methods**. The latter converge reliably in important cases and enable usually significantly faster learning and continual online learning. Hence, semi-gradient methods are often preferred over true gradient methods.

5.1.3 Semi-gradient TD(0) for estimating $v \approx v_\pi$ algorithm

input: policy π to be evaluated

a differentiable function $v : \mathcal{S}^+ \times \mathbb{R}^d$ such that $v(T, \cdot) = 0$

parameter: step size $\alpha > 0$

initialise value-function's weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily, e.g. all to zero

loop over each episode:

initialise S

loop for each step of episode until S is terminal:

choose A according to $\pi(a|S)$
take action A, observe R, S'
$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[R + \gamma v(S', \boldsymbol{w}) - v(S, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} v(S, \boldsymbol{w})$
$S = S'$

In many applications, **state aggregation** is of importance. It is a simple form of generalising function approximation in which states are grouped together with one estimated value (one component of the weight vector) for each group. Values are estimated as their group's component and when the state is updated, the component alone is updated. State aggregation is a special case of stochastic gradient descent (SGD) in which $\nabla_{\boldsymbol{w}} v(S_t, \boldsymbol{w}_t)$ is one for S_t's group's component and 0 for all other components. This method is biased towards the average median of a group of states according to the distribution function $\mu(s)$.

5.1.4 Linear methods

For linear methods the ansatz for the state-value function is

$$v(s, \boldsymbol{w}) = \boldsymbol{x}(s) \cdot \boldsymbol{w} \tag{5.8}$$

where $\boldsymbol{x}$ is the **feature vector** depending on the state s. With this

$$\nabla_{\boldsymbol{w}} v(s, \boldsymbol{w}) = \boldsymbol{x}(s) \rightarrow \ \boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - v(S_t, \boldsymbol{w}_t)\right] \boldsymbol{x}(S_t) \tag{5.9}$$

which will in general converge to an optimum for normal gradient methods and converge for SGD methods if

$$A = \langle \boldsymbol{x}_t (\boldsymbol{x}_t - \gamma \boldsymbol{x}_{t+1}) \rangle \in \mathbb{R}^d \times \mathbb{R}^d \tag{5.10}$$

is positive semi-definite. The quantity $A^{-1}\boldsymbol{b} = \boldsymbol{w}$ is called the **fixed point** of the method where $\boldsymbol{b} = \langle R_{t+1} \boldsymbol{x}_t \rangle \in \mathbb{R}^d$. All this is only true if the states are updated according to the on-policy distribution if the method bootstraps.
Commonly used features are

- **polynomials:**

$$\boldsymbol{x} = (1, s_1, s_2, \ldots, s_n, s_1^2, s_2^2, \ldots, s_n^2, s_1 s_2, s_1 s_3, \ldots, s_1 s_n, \\ s_1^2 s_2, s_1^2 s_3, \ldots, s_1^2 s_n, \ldots, s_1^{k-l} s_2^l, \ldots, s_1^k, \ldots, s_n^k)$$

- **Fourier basis:**

$$\boldsymbol{x} = (1, \cos(\pi s_1), \cos(2\pi s_1), \ldots, \cos(k\pi s_n), \cos(\pi(s_1 + s_2)), \cos(2\pi(s_1 + s_2)), \ldots \\ \cos(k\pi(s_{n-1} + s_n), \cos(\pi(2s_1 + s_2)), \cos(2\pi(2s_1 + s_2)), \ldots, \cos(k\pi(2s_{n-1} + s_n), \ldots)$$

- **coarse coding:** feature is one if it is contained in some boundary in state space and zero if not

- **tile coding:** receptive fields of the features are packed into partitions of the state space. Each partition is a tiling and each element of the partition is a tile. A state is represented by the indices of the tiling it lies in (by the tiles it lies in). By slightly shifting or rotating the tiles against each other one achieves good effects (stretching is also a possibility). The feature vector $\boldsymbol{x}$ has one component for each tile in each tiling. Binary nature of features brings great computational advantage

- **radial basis functions** are the generalisation of coarse coding to continuous-valued features. Rather than each feature being 0 or 1, it can be anything in between. The most common functions used are Gaussian features

$$x_i = \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right). \tag{5.11}$$

  The features of the radial basic functions can be learned too. This is then more or less non-linear function approximation

A good choice for the step size is $\alpha = \frac{1}{\tau\langle||x||\rangle}$ as it takes at least τ encounters with the state to learn.

5.1.5 Non-linear function approximation: artificial neural networks

There are two simple examples of neural nets:

- **Feed-forward neural nets:** where all neurons are just connected by outputs in directions of the activation function, i.e. there are no paths where a unit's input can influence its output

- **recurrent neural net:** there is at least one loop in the net

In order for a neural net to be able to approximate any function, there need to be at least one hidden layer and at least one unit needs to have a non-linear activation function. In practise more hidden layers are usually required. Methods using gradient evaluation by back-propagation are best, but methods that use RL principles can be used too. Back-propagation is a method for shallow nets. In deeper nets, the gradients decay rapidly to the input side or they increase rapidly to the input side. This means slow learning for the former and unstable learning for the latter scenario, respectively.

5.1.6 Least-squares temporal difference

Instead of using the incremental improvement steps by the gradient, one can use the fixed point equation

$$\boldsymbol{w}^* = A^{-1}\boldsymbol{b} \text{ with } \boldsymbol{b} = \langle R_{t+1}\boldsymbol{x}_t\rangle \tag{5.12}$$

$$A = \langle \boldsymbol{x}_t(\boldsymbol{x}_t - \gamma\boldsymbol{x}_{t+1})^\intercal\rangle \tag{5.13}$$

by using the natural estimates

$$A_t = \sum_{k=0}^{t-1} \boldsymbol{x}_k(\boldsymbol{x}_k - \gamma\boldsymbol{x}_{t+1})^\intercal) + \epsilon\mathrm{I} \text{ and } \boldsymbol{b} = \sum_{k=0}^{t-1} R_{k+1}\boldsymbol{x}_k \tag{5.14}$$

where the addend $\epsilon\mathrm{I}$ is to ensure invertibility. The fixed point equation is then

$$\boldsymbol{w}_t = A_t^{-1}\boldsymbol{b}_t \tag{5.15}$$

$$A_t^{-1} = A_{t-1}^{-1} - \frac{A_{t-1}^{-1}\boldsymbol{x}_t(\boldsymbol{x}_t - \gamma\boldsymbol{x}_{t+1})^\intercal A_{t-1}^{-1}}{1 + (\boldsymbol{x}_t - \gamma\boldsymbol{x}_{t+1})^\intercal A_{t-1}^{-1}\boldsymbol{x}_t} \tag{5.16}$$

which is known as the **Sherman-Morrison formula**. Computationally this is more expensive than semi-gradient TD ($\simeq \mathcal{O}(d^2)$), but it learns more quickly and additionally does not forget as there is no step size parameter.
The algorithm is the following:

input: feature representation $\boldsymbol{x}, \mathcal{S} \mapsto \mathbb{R}^d$ such that $\boldsymbol{x}(\text{terminal}) = \mathbf{0}$

parameters: small $\epsilon > 0$

$A^{-1} = \frac{1}{\epsilon}\mathrm{I}_{d\times d}$

$\boldsymbol{b} = \mathbf{0}$

loop for each episode:

initialise $S, \boldsymbol{x} = \boldsymbol{x}(S)$

loop for each step of episode until S' is terminal:

choose and take action $A \propto \pi(\cdot|S)$ observe R, S'

$\boldsymbol{x}' = \boldsymbol{x}(S')$

$\boldsymbol{V} = (A^{-1})^\intercal(\boldsymbol{x} - \gamma\boldsymbol{x}')$

$A^{-1} = A^{-1} - \frac{(A^{-1}\boldsymbol{x}V^\intercal}{1+\boldsymbol{V}\cdot\boldsymbol{x}}$

$\boldsymbol{b} = \boldsymbol{b} + R\boldsymbol{x}$

$\boldsymbol{w} = A^{-1}\boldsymbol{b}$

$S = S'$

$\boldsymbol{x} = \boldsymbol{x}'$

One does not need to construct the features for linear parametric function approximators, but instead one can construct kernel functions directly without referring to the feature vectors directly. This is known as the **kernel trick**:

$$v(s, D) = \sum_{s'\in D} k(s, s')g(s') \tag{5.17}$$

here D refers to the data set, $k(s, s') = \boldsymbol{x}^\intercal(s)\boldsymbol{x}(s')$ is the kernel, and $g(s')$ is the target for state s' in a stored example. Using **momentum** in the update rule for weights in an artificial neural network

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha\nabla_{\boldsymbol{w}}S(\boldsymbol{w}_t) + \lambda\boldsymbol{M}_t \tag{5.18}$$

$$\boldsymbol{M}_{t+1} = \lambda\boldsymbol{M}_t - \alpha\nabla_{\boldsymbol{w}}S(\boldsymbol{w}_t) \text{ with } \lambda \in (0, 1) \tag{5.19}$$

and using vector step sizes, i.e. a different step size for each component can be beneficial. The goal is to find a viable approximation to the correct action value function

$$q(s, a, \boldsymbol{w}) \approx q_*(s, a) \tag{5.20}$$

with on-policy control, e.g. using the semi-gradient SARSA algorithm. Instead of using the update rule $S_t \mapsto U_t$, one now considers examples of the form $S_t, A_t \mapsto U_t$ where U_t can be any approximation of $q_\pi(S_t, A_t)$. The update of the weights for the action-value prediction is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - q(S_t, A_t, \boldsymbol{w}_t)\right] \nabla_{\boldsymbol{w}} q(S_t, A_t, \boldsymbol{w}_t), \tag{5.21}$$

$$\text{with } U_t = R_{t+1} + \gamma q(S_{t+1}, A_{t+1}, \boldsymbol{w}_t) \text{ for one SARSA step.} \tag{5.22}$$

This action-value prediction has to be coupled with policy improvement and action selection techniques. Assuming that the action set is not too large, i.e. finite and discrete, one can use ϵ-greedy action selection as the new policy for the next action-value-estimation step. The corresponding **algorithm for episodic semi-gradient one-step SARSA for estimating** $q \approx q_*$ is:

input: differentiable action-value function parametrisation $q \in \mathcal{S} \times \mathcal{A} \times \mathbb{R} \mapsto \mathbb{R}$

parameters: step size $\alpha > 0$, small $\epsilon > 0$

initialise action-value function's weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily

loop for each episode:

randomly select S, A (A e.g. by ϵ-greedy policy)

loop for each step of the episode:

Take action A, observe R, S'

if S' is terminal:

$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[R - q(S, A, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} q(S, A, \boldsymbol{w})$

go to next episode

choose A' as a function of $q(S', \cdot, \boldsymbol{w})$, e.g. by ϵ greedy policy

$\boldsymbol{w} \to \boldsymbol{w} + \alpha \left[R + \gamma q(S', A', \boldsymbol{w}) - q(S, A, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} q(S, A, \boldsymbol{w})$

$S = S'$

$A = A'$

the corresponding **multi-step SARSA (n-step SARSA) algorithm** is the following:

input: as for one-step SARSA + a policy π (if estimating q_π)

parameters: as for one-step SARSA + a positive integer n

initialise action-value function weights arbitrarily

all store and access operations (S_t, A_t, and R_t) can take their index mod $n+1$

loop for each episode:

initialise and store $S_0 \neq$ terminal
select and store an action $A_0 \propto \pi(\cdot|S_0)$ or ϵ-greedy with respect to $q(S_0, \cdot, \boldsymbol{w})$
$T = \infty$
loop for $t = 0, 1, 2, \ldots T-1$
if $t < T$ then:
take action A_t
observe and store next reward as R_{t+1} and the next state as S_{t+1}
if S_{t+1} is terminal:$T = t+1$
else:
select and store $A_{t+1} \propto \pi(\cdot|S_{t+1})$ or ϵ-greedy with respect to $q(S_{t+1}, \cdot, \boldsymbol{w})$
$\tau = t - n + 1$
if $\tau \geq 0$:
$G = \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
if $\tau + n < T$:
$G = G + \gamma^n q(S_{\tau+n}, A_{\tau+n}, \boldsymbol{w})$
$\boldsymbol{w} = \boldsymbol{w} + \alpha \left[G - q(S_\tau, A_\tau, \boldsymbol{w})\right] \nabla_{\boldsymbol{w}} q(S_\tau, A_\tau, \boldsymbol{w})$

5.2 Average reward: a new problem setting for continuing tasks

Unlike the discounted setting, the agent cares as much about immediate rewards as it does for delayed rewards. This is less problematic than the discounted setting for continuous tasks. The quality of π is defined as the average rate of rewards (**average reward** for short) while following the policy:

$$\begin{aligned} r(\pi) &= \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \langle R_t \rangle_{S_0, A_[0:t-1\propto\pi} = \langle R_t \rangle_{A_{0:t-1\propto\pi}} \\ &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r \end{aligned} \tag{5.23}$$

with the steady-state distribution μ_π under π

$$\mu_\pi(s) = \lim_{t\to\infty} p(S_t = s | A_{0:t-1\propto\pi}) \tag{5.24}$$

and assuming **ergodicity**

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s'|s, a) = \mu_\pi(s') \tag{5.25}$$

independent of the initial state S_0. The returns are defined in terms of differences between rewards and the average rewards

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \ldots \tag{5.26}$$

which is called the **differential return** and leads to **differential value functions**

$$v_\pi(s) = \langle G_t \rangle_{\pi, S_t=s}, \tag{5.27}$$
$$q_\pi(s,a) = \langle G_t \rangle_{S_t=s, A_t=a}. \tag{5.28}$$

For those the following Bellmann equations hold:

$$v_\pi(s) = \sum_a \pi(a \check{} s) \sum_{r,s'} p(s', r|s, a) \left[r - r(\pi) + v_\pi(s') \right], \tag{5.29}$$
$$q_\pi = \sum_{r,s'} p(s', r|s, a) \left[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s', a') \right], \tag{5.30}$$
$$v_*(s) = \max_a \sum_{r,s'} p(s', r|s, a) \left[r - \max_\pi r(\pi) + v_*(s') \right], \tag{5.31}$$
$$q_\pi = \sum_{r,s'} p(s', r|s, a) \left[r - \max_\pi r(\pi) + \max_{a'} q_*(s', a') \right]. \tag{5.32}$$

The TD errors are

$$\delta_t = R_{t+1} - \overline{R}_t + v(S_{t+1}, \boldsymbol{w}_t) - v(S_t, \boldsymbol{w}_t), \tag{5.33}$$
$$\delta_t = R_{t+1} - \overline{R}_t + q(S_{t+1}, A_{t+1}, \boldsymbol{w}_t) - q(S_t, A_t, \boldsymbol{w}_t) \text{for one-step SARSA}, \tag{5.34}$$
$$\delta_t = G_{t:t+n} - q(S_t, A_t, \boldsymbol{w}) \text{ for n-step SARSA} \tag{5.35}$$

with $\overline{R}_t$ being the estimate of the average reward $r(\pi)$. With these definitions, all other previously introduced algorithms work, e.g. the differential semi-gradient SARSA algorithm for estimation of $q \approx q_\pi$:

input: a differentiable action-value function parametrisation $q : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \mapsto \mathbb{R}$

parameters: step size $\alpha, \beta > 0$

initialise value function weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily

initialise average reward estimate $\overline{R} \in \mathbb{R}$ arbitrarily

initialise state S and action A

loop over each step:

take action A, observe R, S'
choose A' as a function of $q(S', \cdot, \boldsymbol{w})$
$\delta = R - \overline{R} + q(S', A', \boldsymbol{w})$
$\overline{R} = \overline{R} + \delta\beta$
$\boldsymbol{w} = \boldsymbol{w} + \alpha\delta\nabla_{\boldsymbol{w}} q(S, A, \boldsymbol{w})$
$S = S'$
$A = A'$

In function approximation there is no policy improvement theorem. Improving the policy with respect to values of states is not guaranteed to improve the overall policy. It is better to use algorithms with a **policy-gradient theorem** instead which use a parametrised policy.

5.3 Policy gradient methods

Policy gradient methods do not need to use action-value estimates, but learn a parametrised policy which can select actions without consulting a value function[2], i.e.

$$\pi(a|s,\boldsymbol{\theta}) = p(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}) \text{ with } \boldsymbol{\theta} \in \mathbb{R}^{d'}. \tag{5.36}$$

The policy's parameters are learned by maximising a performance function $J(\boldsymbol{\theta})$ according to the update rule

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\langle\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)\rangle. \tag{5.37}$$

Methods that use this principle are called **policy gradient methods**. If the method learns an estimate both for the policy and the value function, the method is called a **actor-critic method**. The **actor** is a reference to the learned policy and the **critic** is a reference to the learned value function. The policy can be parametrised in any way as long as the function is differentiable with respect to $\boldsymbol{\theta}$ and[3] never becomes deterministic. Often one forms numerical preferences $h(s,a,\boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair and assigns the following probabilities to the policies:

$$\pi(a|s,\boldsymbol{\theta}) = \frac{\mathrm{e}^{h(s,a,\boldsymbol{\theta})}}{\sum\limits_b \mathrm{e}^{h(s,b,\boldsymbol{\theta})}} \tag{5.38}$$

which is called **exponential soft-max in action distribution** and the overall method **soft-max in action preferences**.
The preference $h(s,a,\boldsymbol{\theta})$ can be calculated arbitrarily, e.g. by an artificial neural network or simply by a linear function of the features $\boldsymbol{x}(s,a)$

$$h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta} \cdot \boldsymbol{x}(s,a). \tag{5.39}$$

Including a temperature in the soft-max distribution would require too much information about the action-space to be useful when using value functions. Policy approximating methods can find stochastic optimal policies in contrast to action-value methods which can only find deterministic policies. The policy is often a much simpler function to approximate than a action-value parametrisation.

5.3.1 Policy gradient theorem

The action probabilities change smoothly as a function of the learned parameters. Hence, there are stronger convergence guarantees as those can change dramatically when action values change slightly, e.g. in a ϵ-greedy policy. The performance measure is defined by the value of the start state s_0

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{5.40}$$

which is the true value function for $\pi_{\boldsymbol{\theta}}$ determined by $\boldsymbol{\theta}$. The performance depends both on the action selection and the distribution of states in which those are made. The former and the latter are affected by the policy's parameters. The effect of the parameters on the reward can be easily calculated given a state and the parametrisation. The effect of the policy on the state distribution, which is a function of the environment, is typically unknown. How can one evaluate the performance with respect to the policy parameters when the gradient depends on the unknown effect of policy changes on the state distribution?

[2] Although it might be used to learn the policy's parameters
[3] to ensure exploration

First one can try to evaluate the gradient of the state-value function

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} v_\pi(s) &= \nabla_{\boldsymbol{\theta}} \left(\sum_a \pi(a|s) q_\pi(s,a) \right) \forall\, s \in \mathcal{S} \\ &= \sum_a \left[(\nabla_{\boldsymbol{\theta}} \pi(a|s)) q_\pi(s,a) + \pi(a|a) \nabla_{\boldsymbol{\theta}} q_\pi(s,a) \right]. \end{aligned} \tag{5.41}$$

Using $q_\pi = \sum_{s',r} p(s', r|s, a) \left[r + v_\pi(s') \right]$ the relation can further be manipulated

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} v_\pi(s) &= \sum_a \left[(\nabla_{\boldsymbol{\theta}} \pi(a|s)) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla_{\boldsymbol{\theta}} v_\pi(s') \right] \\ &= \sum_a \Bigg[(\nabla_{\boldsymbol{\theta}} \pi(a|s)) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \times \Bigg[\\ &\quad \times \left[\sum_{a'} (\nabla_{\boldsymbol{\theta}} \pi(a'|s')) q_\pi(s',a') + \pi(a',s') \sum_{s''} p(s''|s',a') \nabla_{\boldsymbol{\theta}} v_\pi(s'') \right] \\ &= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} p(s \to x, k, \pi) \sum_a (\nabla_{\boldsymbol{\theta}})\, q_\pi(x,a) \end{aligned} \tag{5.42}$$

where $p(s \to x, k, \pi)$ is the probability of transitioning from state s into state x in k steps under policy π. For the performance measure this means the following

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} v_\pi(s_0) &= \sum_s \left(\sum_{k=0}^{\infty} p(s \to x, k, \pi) \sum_a (\nabla_{\boldsymbol{\theta}} \pi(a|s))\, q_\pi(s,a) \right) \\ &= \sum_s \eta(s) \sum_a (\nabla_{\boldsymbol{\theta}} \pi(a|s)) q_\pi(s,a) \\ &= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'}} \sum_a (\nabla_{\boldsymbol{\theta}} \pi(a|s)) q_\pi(s,a) \\ &= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a (\nabla_{\boldsymbol{\theta}} \pi(a|s)) q_\pi(s,a) \\ &\propto \sum_s \mu(s) \sum_a (\nabla_{\boldsymbol{\theta}} \pi(a|s)) q_\pi(s,a). \end{aligned} \tag{5.43}$$

The proportionality constant in the last line is the average length of an episode for episodic tasks and 1 for continuing tasks. The quintessence of this equality is that **the gradient of the performance with respect to the policy's parameters does not involve the gradient/derivative of the distribution function!**
The following relation:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \left\langle \sum_a q_\pi(S_t,a) \nabla_{\boldsymbol{\theta}} \pi(a|S_t,\boldsymbol{\theta}) \right\rangle_\pi = \left\langle \sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a|S_t,\boldsymbol{\theta}}{\pi(a|S_t,\boldsymbol{\theta}} \right\rangle_\pi \\ &= \left\langle q_\pi(S_t,A_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}}{\pi(A_t|S_t,\boldsymbol{\theta}} \right\rangle_\pi = \left\langle G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}}{\pi(A_t|S_t,\boldsymbol{\theta}} \right\rangle_\pi \end{aligned} \tag{5.44}$$

is helpful as it shows that the gradient can be sampled in every time step to do a so-called **reinforce update**:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}}{\pi(A_t|S_t,\boldsymbol{\theta}} \tag{5.45}$$

$$\text{or } \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta}_t) \text{ (withouth discounting)} \tag{5.46}$$

$$\text{or } \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta}_t) \text{ (for discounted case)} \tag{5.47}$$

5.3.2 REINFORCE with baselines

One can generalise the policy gradient theorem to include a comparison of the action value to an arbitrary baseline $b(s)$

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \left(q_\pi(s,a) - b(s)\right) \nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{theta}) \tag{5.48}$$

as long as $b(s)$ is no function of the action a as one does then subtract some constant function in $\boldsymbol{\theta}$ and

$$\sum_a b(s) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}) = b(s) \sum_a \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}) = b(s) \nabla_{\boldsymbol{\theta}} \underbrace{\sum_a \pi(a|s,\boldsymbol{\theta})}_{=1} = 0. \tag{5.49}$$

The update rule of the parameters is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t)\right) \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta}_t) \tag{5.50}$$

and a natural choice for $b(s)$ are the value functions $v_\pi(S_t, \boldsymbol{w})$ with the weights $\boldsymbol{w}$ being learned beforehand.
The **REINFORCE algorithm with baseline for estimating** $\pi_{\boldsymbol{\theta}} \approx \pi_*$ is

input: a differentiable policy parametrisation $\pi(A|s,\boldsymbol{\theta})$ and a differentiable state-value function parametrisation $v(s,\boldsymbol{w})$

parameters: step size $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\boldsymbol{w}} > 0$

initialise policy parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and the state-value weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily

for each episode or forever loop:

generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
loop for each step of the episode $t = 0, 1, \ldots, T-1$:

$$\begin{aligned} G &= \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \\ \delta &= G - v(S_t,\boldsymbol{w}) \\ \boldsymbol{w} &= \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla_{\boldsymbol{w}} v(S_t,\boldsymbol{w}) \\ \boldsymbol{\theta} &= \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta}) \end{aligned}$$

5.4 Actor-Critic methods

5.4.1 Basic idea

It is also possible to use the baseline as a critic for bootstrapping, i.e for updating the value estimate for a state from estimated values of subsequent states. This introduces a bias and an asymptotic dependence on the quality of the function approximation. As a result, the variance is reduced and the learning is accelerated. All the previous methods such as, e.g. TD(0), SARSA(0), Q-learning, can be used for learning online and faster. The update rules for the policy's parameters are

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - v(S_t, \boldsymbol{w})\right) \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t, \boldsymbol{\theta}_t), \\ &= \boldsymbol{\theta}_t + \alpha \left(R_{t+1} + \gamma v(S_t, \boldsymbol{w}) - v(S_c, \boldsymbol{w})\right) \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t, \boldsymbol{\theta}_t), \\ &= \boldsymbol{\theta}_t + \alpha \delta_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t, \boldsymbol{\theta}_t),\end{aligned} \tag{5.51}$$

respectively.
The **algorithm for an actor-critic with eligibility traces (episodic) for estimating** $\pi_{\boldsymbol{\theta}} \approx \pi_*$ is:

input: a differentiable policy parametrisation $\pi(a|s, \boldsymbol{\theta})$ and a differentiable state-value function parametrisation $v(s, \boldsymbol{w})$

parameters: trace decay rates $\lambda^{\boldsymbol{\theta}} \in [0, 1], \lambda^{\boldsymbol{w}} \in [0, 1]$, step sizes $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\boldsymbol{w}} > 0$

initialise policy parameters $\theta \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^d$

loop forever/for each episode:

initialise S (first state of the episode)
$\boldsymbol{z}^{\boldsymbol{\theta}} = \mathbf{0}$ (d' dimensional eligibility trace vector)
$\boldsymbol{z}^{\boldsymbol{w}} = \mathbf{0}$ (d dimensional eligibility trace vector)
$I = 1$
loop while S is not terminal (for each time step):
$A \propto \pi(\cdot|S, \boldsymbol{\theta})$
take action A, observe S', R
$\delta = R + \gamma v(S', \boldsymbol{w}) - v(S, \boldsymbol{w})$ (if S' is terminal then $v(S', \boldsymbol{w}) = 0$)
$z^{\boldsymbol{w}} = \gamma \lambda^{\boldsymbol{w}} \boldsymbol{z}^{\boldsymbol{w}} + \nabla_{\boldsymbol{w}} v(S', \boldsymbol{w})$
$z^{\boldsymbol{\theta}} = \gamma \lambda^{\boldsymbol{\theta}} \boldsymbol{z}^{\boldsymbol{\theta}} + I \nabla_{\boldsymbol{\theta}} \pi(A|S, \boldsymbol{\theta})$
$\boldsymbol{w} = \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \boldsymbol{z}^{\boldsymbol{w}}$
$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \boldsymbol{z}^{\boldsymbol{\theta}}$
$I = \gamma I$
$S = S'$

For continuing tasks one uses an additional parameter α^R and a randomly initialised variable $\overline{R}$. The update rules to use are:

$$\begin{aligned}\delta &= R - \overline{R} + v(S', \boldsymbol{w}) - v(S, \boldsymbol{w}) \\ \overline{R} &= \overline{R} + \alpha^{\overline{R}} \delta \\ z^{\boldsymbol{w}} &= \lambda^{\boldsymbol{w}} \boldsymbol{z}^{\boldsymbol{w}} + \nabla_{\boldsymbol{w}} v(S, \boldsymbol{w}) \\ z^{\boldsymbol{\theta}} &= \lambda^{\boldsymbol{\theta}} \boldsymbol{z}^{\boldsymbol{\theta}} + \nabla_{\boldsymbol{\theta}} \pi(A|S, \boldsymbol{\theta}) \\ \boldsymbol{w} &= \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \boldsymbol{z}^{\boldsymbol{w}} \\ \boldsymbol{\theta} &= \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \boldsymbol{z}^{\boldsymbol{\theta}} \\ S &= S'\end{aligned}$$

5.4.2 Policy parametrisation for continuous actions

When the action space is not discrete but is continuous, a normal distributed policy is used often

$$\pi(a|s,\boldsymbol{\theta}) = \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right) \tag{5.52}$$

with $\mu : \mathcal{S} \times \mathbb{R}^{d'} \mapsto \mathbb{R}$ and $\sigma : \mathcal{S} \times \mathbb{R}^{d'} \mapsto \mathbb{R}$ being two parametrised function approximators

$$\boldsymbol{\theta} = \begin{pmatrix} \boldsymbol{\theta_\mu} \\ \boldsymbol{\theta}_\sigma \end{pmatrix} \text{ with } \mu(s,\boldsymbol{\theta}) = \boldsymbol{\theta_\mu} \cdot \boldsymbol{x}_\mu(s),\ \sigma(s,\boldsymbol{\theta}) = \exp\left(\boldsymbol{\theta}_\sigma \cdot \boldsymbol{x}_\sigma(s)\right) \tag{5.53}$$

where $\boldsymbol{x}_\mu, \boldsymbol{x}_\sigma$ are state feature vectors of arbitrary construction. The gradient of the policy becomes

$$\nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}) = \frac{(a-\mu(s,\boldsymbol{\theta}_\mu))}{\sigma^2} \boldsymbol{x}_\mu(s) + \left(\frac{(a-\mu(s,\boldsymbol{\theta}_\mu))^2}{\sigma(s,\boldsymbol{\theta}_\sigma)^2} - 1\right) \cdot \boldsymbol{x}_\sigma(s). \tag{5.54}$$

In Figure 5.1 the way how actor and critic interact is sketched.

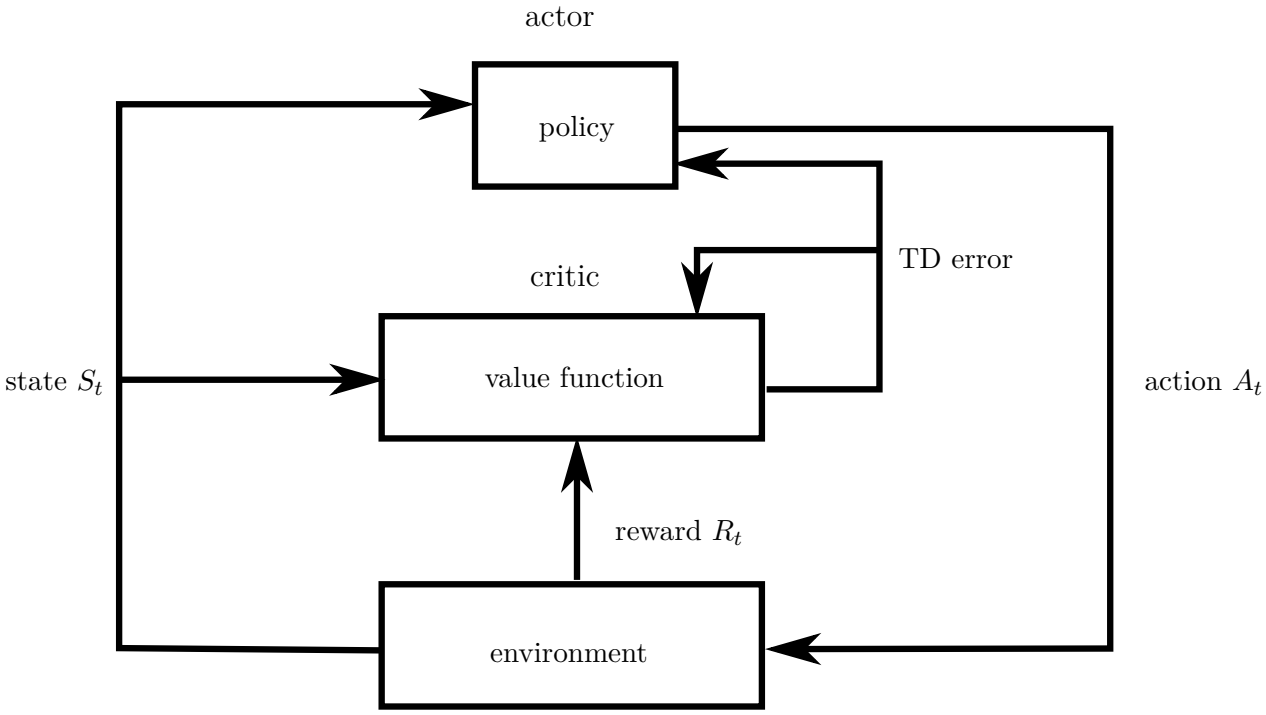


Figure 5.1: A schematic illustration of how actor and critic interact with each other in the general reinforcement learning framework.

5.4.3 Advantage actor critic methods (A2C)

In **advantage actor critic (A2C) methods** the agents learn the advantage function

$$\mathtt{A}_\pi(s,a) = q_\pi(s,a) - v_\pi(s) \tag{5.55}$$

as the reinforcing signal. The action is selecting based on how well it performs relative to the other available actions in a particular state. The actor is similar to REINFORCE, except that the new update rule for the policy parameters is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \langle \mathtt{A}_{t,\pi} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \rangle \,. \tag{5.56}$$

The critic is different:

$$\langle \mathtt{A}_\pi(s,a) \rangle = 0. \tag{5.57}$$

When all actions are equivalent then $\mathtt{A}_\pi = 0 \ \forall a \in \mathcal{A}(s)$ and the policy will not change. No action is penalised for being in a bad state or is given credit for being in a particular good state. For estimating $\mathtt{A}_p i(s,a)$ there are different procedures. One firstly learns $v_\pi(s)$ and estimates $q_\pi(s,a)$ in all of it

$$\begin{aligned} q_\pi(s_t,a_t) &= \langle r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n r_{t+n} \rangle + \gamma^{n+1} v_\pi(s_{t+n+1}) \\ &\approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n r_{t+n} + \gamma^{n+1} v_\pi(s_{t+n+1}). \end{aligned} \tag{5.58}$$

Here n controls the trade-off between bias and variance. For larger n one gets a high variance and a low bias. For a small n one gets a high bias, but a low variance. This estimation is called the **n-step estimation of the advantage function**

$$\mathtt{A}_\pi^{\text{n-step}}(s_t,a_t) \approx r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n r_{t+n} + \gamma^{n+1} v_\pi(s_{t+n+1}) - v_\pi(s_t). \tag{5.59}$$

The variance can be reduced while keeping the bias as low as possible by weighting different results for different n (exponentially). This is the **generalised advantage estimation** (GAE)

$$\mathtt{A}_\pi^{\text{GAE}}(s_t,a_t) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l} \tag{5.60}$$

$$\text{with } \delta_t = r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t) \tag{5.61}$$

where λ controls how fast contributions from higher-variance, lower-bias estimators decay. A larger λ leads to a higher variance. Here is a proof of that property: one can write the general advantage estimation as a sum of n-step estimators $\mathtt{A}_\pi(t,n)$ of different depths

$$\mathtt{A}_\pi^{\text{GAE}}(s_t,a_t) = (1-\lambda)\left[\mathtt{A}_\pi(t,1) + \lambda \mathtt{A}_\pi(t,2) + \lambda^2 \mathtt{A}_\pi(t,3) + \cdots + \lambda^\infty \mathtt{A}_\pi(t,\infty)\right]$$

because by telescope sum expansion

$$\begin{aligned} \mathtt{A}_\pi(t,n) = {} & r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t) + \gamma\left(r_{t+1} + \gamma v_\pi(s_{t+2}) - v_\pi(s_{t+1})\right) \\ & + \gamma^2\left(r_{t+2} + \gamma v_\pi(s_{t+3}) - v_\pi(s_{t+2})\right) + \gamma^3\left(r_{t+3} + \gamma v_\pi(s_{t+4}) - v_\pi(s_{t+3})\right) \\ & + \ldots \\ & + \gamma^n\left(r_{t+n} + \gamma v_\pi(s_{t+n+1}) - v_\pi(s_{t+n})\right) \\ & \sum_{l=0}^{n-1} \gamma^l \delta_{t+l} \end{aligned}$$

which proves above equality.
The advantage function is learned by parametrising v_π with $\boldsymbol{\theta}$ and generates v_π^{tar} for each of the experiences an agent gathers. Next, the difference between both is minimised, e.g. by using MSE, and that is repeated for many steps. Possible target values are

- $v_\pi^{\text{tar}}(s_t) = \sum_{l=0}^{n} \gamma^l r_{t+l} + \gamma^{n+1} v_\pi(s_{t+n+1})$ which is best for estimation with n-step return
- $v_\pi^{\text{tar}}(s_t) = \mathtt{A}_\pi^{\text{GAE}}(s_t, a_t) + v_\pi(s_t)$
- trust region method, which is a method that evaluates the gradient's step size before the gradient by looking at the predicted vs. the actual improvement in the last step

The **A2C algorithm is the following**:

set $\beta \geq 0$ (entropy regularisation weight)

set $\alpha_A \geq 0$ (actor learning rate)

set $\alpha_C \geq 0$ (critic learning rate)

randomly initialise the actor and critic parameters $\boldsymbol{\theta}_A, \boldsymbol{\theta}_C$

for each episode do:

gather and store trajectories by acting in the environment using the current policy

for each step until termination do

calculate predicted v-value $v_\pi(s_t)$ using the critic network $\boldsymbol{\theta}_C$
calculate the advantage $\mathtt{A}_\pi(s_t, a-t)$ using the critic network $\boldsymbol{\theta}_C$
calculate $v_\pi^{\text{tar}}(s_t)$ using the critic network $\boldsymbol{\theta}_C$ and/or trajectory data
optionally calculate the entropy H_t of the policy distribution using the actor network $\boldsymbol{\theta}_A$. Otherwise set $\beta = 0$

calculate value loss, e.g. using MSE:

$$L_{\text{val}}(\boldsymbol{\theta}_C) = \frac{1}{T} \sum_{t=0}^{T} \Big(v_\pi(s_t) - v_\pi^{\text{tar}}(s_t) \Big)^2$$

calculate policy loss:

$$L_{\text{pol}}(\boldsymbol{\theta}_C) = -\frac{1}{T} \sum_{t=0}^{T} \mathtt{A}_\pi(s_t, a_t) \log \pi_{\boldsymbol{\theta}_A}(a_t|s_t) + \beta H_t$$

update the critic parameters, e.g. using SGD:

$$\boldsymbol{\theta}_C = \boldsymbol{\theta}_C + \alpha_C \nabla_{\boldsymbol{\theta}_C} L_{\text{val}}(\boldsymbol{\theta}_C)$$

update the actor parameters, e.g. using SGD:

$$\boldsymbol{\theta}_A = \boldsymbol{\theta}_A + \alpha_C \nabla_{\boldsymbol{\theta}_A} L_{\text{pol}}(\boldsymbol{\theta}_A)$$

5.4.4 Actor critic using Kronecker-factored trust region (ACKTR)

The **ACKTR algorithm** is essentially the same algorithm as A2C, but with an optimiser that uses the curvature instead of just the gradient. In particular the **Fisher-information** is used as it is a measure of how sharp the found maximum of the **log-likelihood** is. A lower Fisher information means that there are many points close by with essentially the same log-likelihood whereas a high Fisher information means that the log-likelihood is clearly different when moving away from the maximum. For high-dimensional optimisation problems this defines the **Fisher-information metric** which is the Hessian of the relative entropy and allows confidence region estimates for maximum likelihood estimations. The Fisher information metric can be approximated by Monte-Carlo estimates of the negative log-likelihood function's Hessian with estimates based on values of the negative log-likelihood function's values or gradients. Hence, there is no need for a analytical calculation of the Hessian (second derivative) needed. For a Gaussian distribution[4] the Fisher information metric $I(\boldsymbol{\beta}, \boldsymbol{\theta})$ has a convenient form:

$$I(\boldsymbol{\beta}, \boldsymbol{\theta}) = \operatorname{diag}\left(I(\boldsymbol{\beta}), I(\boldsymbol{\theta})\right) \tag{5.62}$$

$$\text{with } I(\boldsymbol{\beta})_{m,n} = \frac{\partial \mu^{\intercal}}{\partial \beta_m} \Sigma^{-1} \frac{\partial \mu}{\partial \beta_n} \tag{5.63}$$

$$\text{and } I(\boldsymbol{\theta})_{m,n} = \frac{1}{2} \operatorname{Tr}\left(\sigma^{-1} \frac{\partial \Sigma}{\partial \theta_m} \Sigma^{-1} \frac{\partial \Sigma}{\partial \theta_n}\right) \tag{5.64}$$

5.4.5 Proximal policy optimisation (PPO)

A common challenge when training agents with policy gradient algorithms is that they are susceptible to performance collapse, i.e. an agent suddenly starts to perform badly. Agents start to generate poor trajectories which are then used to optimise the policy. Furthermore, on-policy algorithms are sample inefficient because they can not reuse data. The **proximal policy optimisation (PPO)** algorithms addresses both issues by introducing a surrogate objective which avoids performance collapse by guaranteeing monotonic policy improvement and reusing off-policy data in training.

Performance collapse

Optimising policies by using policy gradients is an indirect method as one is looking for an optimal policy in the **policy space** which one has no direct control over. During optimisation one searches over a sequence of policies $\pi_1, \ldots, \pi_n$ within a set of all policies $\Pi = \{\pi_i\}$ by using a parametrisation from **parameter space** $\Theta = \{\theta \in \mathbb{R}^m\}$ where m is the number of parameters. The optimal policy is found by searching in parameter space for the right parameters $\theta \in \Theta$. Hence, the control is in Θ and not in Π. The step size is controlled by learning rate α

$$\Delta \boldsymbol{\theta} = \alpha \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}). \tag{5.65}$$

Unfortunately, both spaces do not map congruently, i.e. distances in both spaces do not correspond

$$\left\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\right\|_{\Theta} = \left\|\boldsymbol{\theta}_2 - \boldsymbol{\theta}_3\right\|_{\Theta} \not\Leftrightarrow \left\|\pi_1 - \pi_2\right\|_{\Pi} = \left\|\pi_2 - \pi_3\right\|_{\Pi}. \tag{5.66}$$

It is hard to estimate a good step size in the parameter space so that the learning is not too slow or gets stuck in a local maximum, does not overshoot the neighbourhood of good policies and cause a performance collapse. A constant step size does not avoid this issue as it can map to different differences in policy space. Therefore, one needs to look for an algorithm that automatically adapts the step size in policy space. One needs to find a way to measure differences in performance between two policies. Such a measure is the **relative performance**

[4] with mean vector $\boldsymbol{\mu}$ and covariance matrix Σ

identity[5]

$$J(\pi') - J(\pi) = \left\langle \sum_{t=0}^{T} \gamma^t \mathtt{A}_\pi(s_t, a_t) \right\rangle_{\tau \propto \pi'} \tag{5.67}$$

measuring the difference in performance of two policies. This is proven as follows

$$\begin{aligned}
\mathtt{A}_\pi(s_t, a_t) &= q_\pi(s_t, a_t) - v_\pi(s_t) = \langle r_t + \gamma v_\pi(s_{t+1}) \rangle_{s_{t+1}, r_t \propto p(s_{t+1}, r_t | s_t, a_t)} - v_\pi(s_t) \\
\left\langle \sum_{t\geq 0} \gamma^t A_\pi(s_t, a_t) \right\rangle_{\tau \propto \pi'} &= \left\langle \sum_{t\geq 0} \gamma^t \left(r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t) \right) \right\rangle_{\tau \propto \pi'} \\
&= \left\langle \sum_{t\geq 0} \gamma^t r_t \right\rangle_{\pi'} + \left\langle \sum_{t\geq 0} \gamma^{t+1} v_\pi(s_{t+1}) - \sum_{t\geq 0} \gamma^t v_\pi(s_t) \right\rangle_{\tau \propto \pi'} \\
&= J(\pi') + \left\langle \sum_{t\geq 1} \gamma^t v_\pi(s_t) - \sum_{t\geq 0} \gamma^t v_\pi(s_t) \right\rangle_{\tau \propto \pi'} \\
&= J(\pi') - \langle v_\pi(s_0) \rangle_{\tau \propto \pi'} = J(\pi') - \langle J(\pi) \rangle_{\tau \propto \pi'} \\
&= \underline{\underline{J(\pi') - J(\pi)}} \quad \text{(q.e.d.)}
\end{aligned}$$

This metric is used to measure policy improvements. If it is positive, the new policy is better. If it is negative, the old one is better. One has to choose the new policy such that the difference is maximised. Hence the objective is to maximise $J(\pi')$, e.g. by gradient ascent

$$\max_{\pi'} J(\pi') \Leftrightarrow \max_{\pi'} \left(J(\pi') - J(\pi) \right). \tag{5.68}$$

Additionally, one can ensure non-negative monotonic improvement by leaving $\pi = \pi'$ if the difference is negative. With this there can be no performance collapse throughout training. Unfortunately, one can not directly use this quantity as it requires the trajectories to be sampled by the new policy π' before the update, i.e. when it is not available.
Instead one can assume that successive policies π, π' are relatively close to each other and use importance sampling. The distance between two probability distributions P and Q is measured by the **Kullback-Leibler divergence**

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}. \tag{5.69}$$

The importance sampling is called **conservative policy iteration (CPI)**

$$\begin{aligned}
J(\pi') - J(\pi) &= \left\langle \sum_{t\geq 0} \mathtt{A}_\pi(s_t, a_t) \right\rangle_{\tau \propto \pi'} = \left\langle \sum_{t\geq 0} \mathtt{A}_\pi(s_t, a_t) \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)} \right\rangle_{\tau \propto \pi} \\
&= J_\pi^{\mathrm{CPI}}(\pi').
\end{aligned} \tag{5.70}$$

for the surrogate objective. The gradient of the surrogate objective is equal to the policy gradient:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J^{\mathrm{CPI}}_{\boldsymbol{\theta}_{\mathrm{old}}}\big|_{\boldsymbol{\theta}_{\mathrm{old}}} &= \nabla_{\boldsymbol{\theta}} \left\langle \sum_{t\geq 0} \mathtt{A}_{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}} \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(a_t, s_t)} \right\rangle_{\tau \propto \pi_{\boldsymbol{\theta}_{\mathrm{old}}}} = \left\langle \sum_{t\geq 0} \mathtt{A}_{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t|s_t)\big|_{\boldsymbol{\theta}_{\mathrm{old}}}}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(a_t, s_t)} \right\rangle_{\tau \propto \pi_{\boldsymbol{\theta}_{\mathrm{old}}}} \\
&= \left\langle \sum_{t\geq 0} \mathtt{A}_{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t|s_t)\big|_{\boldsymbol{\theta}_{\mathrm{old}}}}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(a_t, s_t)\big|_{\boldsymbol{\theta}_{\mathrm{old}}}} \right\rangle_{\tau \propto \pi_{\boldsymbol{\theta}_{\mathrm{old}}}} = \left\langle \sum_{t\geq 0} \mathtt{A}_{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}}(s_t, a_t) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \Big|_{\boldsymbol{\theta}_{\mathrm{old}}} \right\rangle_{\tau \propto \pi_{\mathrm{old}}} \\
&= \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) \Big|_{\boldsymbol{\theta}_{\mathrm{old}}}.
\end{aligned} \tag{5.71}$$

[5] the advantage $\mathtt{A}_\pi(s_t, a_t)$ is always calculated from the older policy

$J_\pi^{\text{CPI}}(\pi')$ is a linear approximation to $J(\pi') - J(\pi)$ since their first derivatives are equal. Hence, to ensure that $J_\pi^{\text{CPI}} \geq 0$ is actually an improvement for the policy, the error needs to be understood. If the policies π, π' are close together with respect to their Kullback-Leibler divergence, one can write a **policy performance bound**

$$\left| J(\pi') - J(\pi) - J_\pi^{\text{CPI}}(\pi') \right| \leq C\sqrt{\langle D_{\text{KL}}\left(\pi'(a_t|s_t) \parallel \pi(a_t, s_t)\right)\rangle_t} \tag{5.72}$$

where C is a constant to be chosen. This bound can be used to decide whether to accept an update when

$$J(\pi') - J(\pi) \geq J_\pi^{\text{CPI}}(\pi') - C\sqrt{\langle D_{\text{KL}}\left(\pi'(a_t|s_t) \parallel \pi(a_t, s_t)\right)\rangle_t} \tag{5.73}$$

or decline an update when this is not the case. This procedure is called **monotonic policy improvement**. It does not ensure that the optimal policy is found as one can still get stuck in a local optimum..
Instead one should use a limit

$$\delta \geq \langle D_{\text{KL}}(\pi' \parallel \pi)\rangle \tag{5.74}$$

as a threshold for how far the new policy can deviate from the old one. This is called the **trust region** or **trust region constraint condition**. δ is a hyperparameter that needs to be tuned.

Trust region policy optimisation

The objective is to find

$$\max_{\boldsymbol{\theta}} \langle \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t|s_t)} \mathtt{A}^t_{\pi_{\boldsymbol{\theta}_{\text{old}}}} \rangle_t, \tag{5.75}$$

subject to

$$\langle D_{\text{KL}}(\pi_{\boldsymbol{\theta}}(a_t|s_t) \parallel \pi_{\boldsymbol{\theta}_{\text{old}}})\rangle_t \leq \delta. \tag{5.76}$$

There are a number of algorithms that are proposed to solve the trust region optimisation problem:

- Natural policy gradient (NPG)
- trust region policy optimisation (TRPO)
- constraint policy optimisation (CPO)

All are fairly complex, difficult to implement, their gradients are expensive to compute and it is difficult to obtain a good value for δ. In contrast to this, PPO is easy to implement, computationally inexpensive, and one has not to choose δ. There are two variants:

1. **adaptive KL penalty**
2. **clipped objective**

Abbreviating $r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t|s_t)}$ the surrogate objective function is

$$J^{\text{CPI}}(\boldsymbol{\theta}) = \langle r_t(\boldsymbol{\theta})\mathtt{A}_t \rangle. \tag{5.77}$$

PPO with adaptive KL penalty turns the constraint into an penalty which is subtracted from the importance weighted average and a new objective needs to be maximised

$$J^{\text{KLPEN}}(\boldsymbol{\theta}) = \max_{\boldsymbol{\theta}} \langle r_t(\boldsymbol{\theta})\mathtt{A}_t - \beta D_{\text{KL}}(\pi_{\boldsymbol{\theta}} \parallel \pi_{\boldsymbol{\theta}_{\text{old}}})\rangle_t \tag{5.78}$$

with a adaptive coefficient β.

adaptive algorithm for β

set target value for expectation of KL δ_{tar}

initialise β to a random value

use multiple epochs of minibatch SGD, optimise (maximise) the KL-penalised surrogate objective $J^{\text{KLPEN}}(\boldsymbol{\theta})$ from equation (5.78)

compute $\delta = \langle D_{\text{KL}}(\pi_{\boldsymbol{\theta}} \parallel \pi_{\boldsymbol{\theta}_{\text{old}}})\rangle_t$:

if $\delta < \frac{2}{3}\delta_{\text{tar}} \rightarrow \beta = \beta/2$
else if $\delta > \frac{3}{2}\delta_{\text{tar}} \rightarrow \beta = 2\beta$
else $\rightarrow$ pass

δ has to still be selected and D_{KL} can be expensive to calculate. It, therefore, advisable to use PPO with a clipped surrogate objective

$$J^{\text{CLIP}} = \langle \min\left(r_t(\boldsymbol{\theta})\mathtt{A}_t, \text{clip}\left(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon\right)\mathtt{A}_t\right)\rangle_t \tag{5.79}$$

where ϵ is a hyperparameter that has to be tuned and can decay during training. The clip function $\text{clip}(A, B, C)$ bounds A to lie between B and C. The clipped implementation usually outperforms penalty implementation.

PPO algorithm with clipping, extending actor-critic

set $\beta \geq 0$ (entropy regularisation weight)

set $\epsilon \geq 0$ (clipping variable)

set K (number of epochs)

set N (number of actors)

set T (number of time horizon)

set $M \leq NT$ (mini-batch size)

set $\alpha_{\text{A}} \geq 0$ (actor learning rate)

set $\alpha_{\text{C}} \geq 0$ (critic learning rate)

randomly initialise the actor and critic network parameters $\boldsymbol{\theta}_{\text{A}}$ and $\boldsymbol{\theta}_{\text{C}}$

initialise the ¨old¨ actor network $\boldsymbol{\theta}_{\text{A}_{\text{old}}}$

for $i = 1, 2, \ldots$ do

set $\boldsymbol{\theta}_{\text{A}_{\text{old}}} = \boldsymbol{\theta}_{\text{A}}$
for actor=$1, 2, \ldots, N$ do
run policy $\boldsymbol{\theta}_{\text{A}_{\text{old}}}$ in an environment for T time steps and collect the trajectories
compute advantages $\mathtt{A}_1, \ldots, \mathtt{A}_T$ using $\boldsymbol{\theta}_{\text{A}_{\text{old}}}$
calculate $v_{\pi}^{\text{tar},1}, \ldots, v_{\pi}^{\text{tar},T}$ using the critic network $\boldsymbol{\theta}_{\text{C}}$ and/or trajectory data
let batch with size NT consist of the collected trajectories, advantages, and target v-values
for epoch $= 1, 2, \ldots, K$ do
for minibatch m in batch do
(the following are computed over the mini-batch m)

calculate $J_m^{\text{CLIP}}(\boldsymbol{\theta}_{\boldsymbol{A}})$ using advantages $\mathtt{A}_m$ from the mini-batch and $r_m(\boldsymbol{\theta}_{\text{A}})$ calculate entropies H_m using the actor network $\boldsymbol{\theta}_{\text{A}}$ calculate the policy loss:

$$L_{\text{pol}}(\boldsymbol{\theta}_C) = -\frac{1}{T}\sum_{t=0}^{T} \mathtt{A}_\pi(s_t, a_t) \log \pi_{\boldsymbol{\theta}_A}(a_t|s_t) + \beta H_t$$

update the actor parameters (e.g. using SGD):

$$\boldsymbol{\theta}_{\text{A}} = \boldsymbol{\theta}_{\text{A}} + \alpha \nabla_{\boldsymbol{\theta}_{\text{A}}} L_{\text{pol}}(\boldsymbol{\theta}_{\text{A}})$$

calculate value loss, e.g. using MSE:

$$L_{\text{val}}(\boldsymbol{\theta}_C) = \frac{1}{T}\sum_{t=0}^{T} \Big(v_\pi(s_t) - v_\pi^{\text{tar}}(s_t)\Big)^2$$

update the critic parameters, e.g. using SGD:

$$\boldsymbol{\theta}_C = \boldsymbol{\theta}_C + \alpha_C \nabla_{\boldsymbol{\theta}_C} L_{\text{val}}(\boldsymbol{\theta}_C)$$