MACHINE LEARNING

Part2

Probability Modeling and Reinforcement Learning

PROBABILISTIC METHODS

Machine Learning
Lesson 1 of Part 2 (07/11/23)

Probabilistic Models and Methods

Introduction

- So far, we have considered the machine learning system as a computer with adjustable parameters.
- *Training*, i.e. searching for the best parameters, was performed by minimizing an error function on the training data. In order to encourage generalization, we have introduced *regularization*.

Note that minimising the error function could have been performed either exactly (in linear regression we had to exactly compute the parameters that minimise the loss) or by training, i.e. gradient descent done with Neural Networks.

- All this was done with limited theoretical justification.

With this the professors means that some behaviours of this type of model are not fully understandable or justifiable. So we know that some adjustments work better, but we do not have a full theoretical justification for why they works.

We now consider machine learning based on probabilistic methods. In particular, we model our data by probability distributions, which

⇒ enables novel methods
⇒ naturally gives probabilistic outputs
⇒ helps combatting overfitting
⇒ yields theoretical guarantees
⇒ allows model interpretation.

Even probabilistic models with complex structure (so not just a distribution) can be interpreted (by ansfo a gra to the idea what dep ent on at.

obab stic M ing

- Prob ilistic mode g is ba d on the premise that ta ther in a supe sed setup w targe or in an unsupervised is gene ed by some know ss.

The premise is that data is generated by some unknown random process -> in probabilistic modelling we assume that data is random.

- Therefore, we postulate a *probabilistic model* of data production:

$$\mathcal{X} \sim p(x).$$

- We assume that (training/test) data has been generated by *sampling* from this distribution. In this lecture, we always assume that the samples have been sampled independently from a fixed p, one then says that the data $\mathcal{X}$ is independent and identically distributed (iid).

The professors make the following example to show how complex a distribution can be. Suppose we have an image of 10000 x 100, and 3 channels, then we have a 3mln dimensional space. This means to have a distribution over a space which got 3 mln directions.

indipendnetly means that one sample does not depend on the other, i.e. samples are statistically independent. **identically distributed** means that all samples come from the same probability distribution. This doesn't necessarily mean that the samples are the same, but rather that they are all drawn from the same underlying distribution and therefore follow the same statistical rules, such as having the same mean, variance and so on…
Note that this might not be true any longer, when i deploy a system with works in different situations. E.g. create a model that controls hand prosthesis. You record data in the lab as the training data, by monitoring in a controlled environment (the lab) some action performed by the user. Then the system changes: the person goes out, and starts sweating, or goes shopping and

the weight of the shopping bag changes the way data are recorded. This is a typical example of a **distribution shift** between training and test data, i.e. samples are **no more independent** since the training set and the actual test set are being taken from two different environments/ distributions.

- p is the generative model. It can be used to
  - → make predictions
  - → make inferences about missing inputs
  - → generate predictions/fantasies/imagery.
- In supervised tasks, data *and* targets are comprised in the vector x.

This section (in particular the coin toss example) is heavily based on the UCL slides, lecture 1.

This variable X can potentially incorporate inputs and targets, e.g. in an image classification example, 3mln dimension input and one dimension output (the class).

- All of probabilistic machine learning deals with estimating the distribution p from the training data.
- If we accurately estimate p, we will obtain optimal generalization in the underlying task: the (inference/prediction/generative) system will also work well on new samples.

This means that if we manage to estimate correctly this p distribution, then there won't be any discussion about overfitting or undercutting, since, if we manage to have it, we have the best possible predictor. But this optimal predictor (which is almost impossible to found) can be approximated and, studying the properties of the approximation, getting a **boundary** of the generalisation risk.

- As we know already, this is not always possible, in particular, our model can *overfit* or *underfit*.

We will oundar obabilistic model , and ther methods f Machine arnin are fluent. The p er of probabilisti modeling s based on:

- a p o reaso about uncertainties both i the pred tion and the data elf
- an ensive set mathematical tools.

PREVIEW

What is the criterion to estimate this p?

- For estimating $p(x)$, we need to a) define a suitable criterion, b) describe the space of probability distributions in a way that allows optimization.

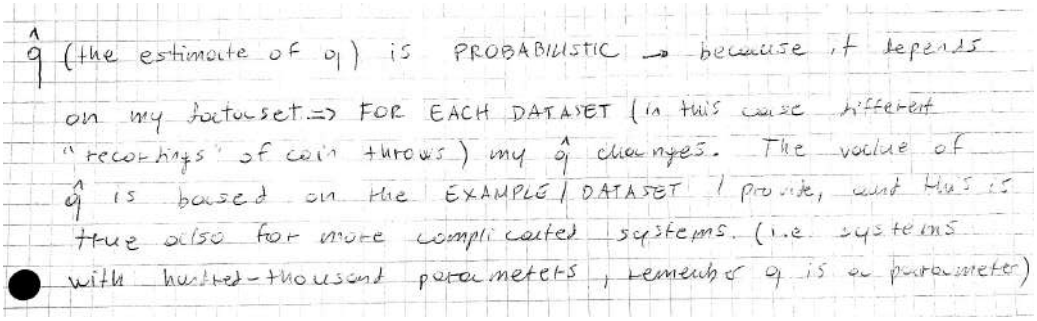
There are many distributions (situations) where I cannot make an estimation in one step, I need to do some kind of iterative process (i.e. training). What professor is saying here is that estimating a distribution often cannot be done in one step and requires an iterative process such as training a model. This iterative process is guided by the criterion and framework established. for complex distributions or large datasets, it is usually not possible to simply calculate the probability distribution in a single calculation. Instead, the process requires iteration.

- In practice, we usually assume that p can be written as a mathematical formula which depends on some parameter vector θ:

$$p(x) = p(x|\theta).$$

- Optimization now amounts to finding the "best" parameter vector θ.
- However, this means that out of all possible probability distributions, we only consider the subset whose elements match the function template $p(x|\theta)$ for some θ.

The core concept is that



Towards a Bayesian Approach - Definitions

- We make the following (generic) definitions:
  - → the parameters θ (can be seen as a vector, or a set)
  - → the Likelihood model or data model $P(\mathcal{X}|\theta)$, where P is a family of probability distributions parametrized by θ
  - → the prior probability of the parameters $P(\theta)$.
- We obtain a *probabilistic* estimate of the parameters θ using Bayes' Theorem:

$$P(\theta|\mathcal{X}) = \frac{P(\mathcal{X}|\theta)P(\theta)}{P(\mathcal{X})}$$

with the evidence

$$P(\mathcal{X}) = \int P(\mathcal{X}, \theta)\,\mathrm{d}\theta = \int P(\mathcal{X}|\theta)P(\theta)\,\mathrm{d}\theta.$$

- $P(\theta|\mathcal{X})$ is the posterior probability of θ, given the data

The **prior probability** of the parameters is a concept derived from Bayes theorem, and means that we start with some idea of what the parameters could be, i.e. we have the probabilistic model (external information) on how the parameters could be.

Bayesian Coin Tossing

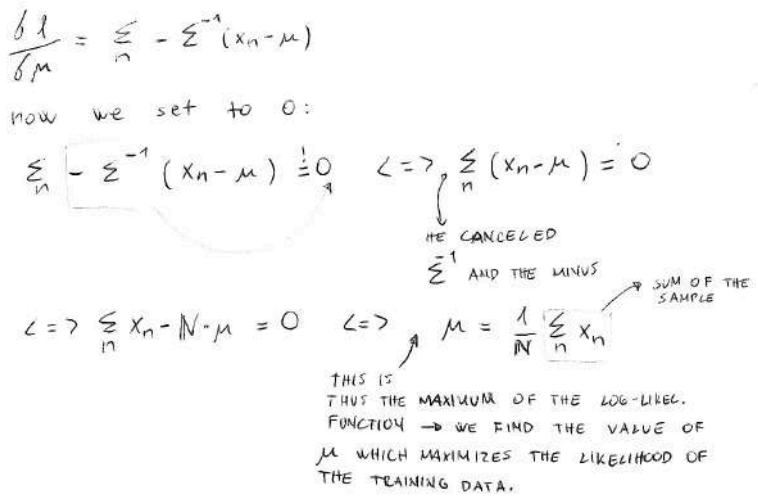
We assume that q is probabilistic, i.e. we do not try to estimate the single value of q, but the whole probability distribution.

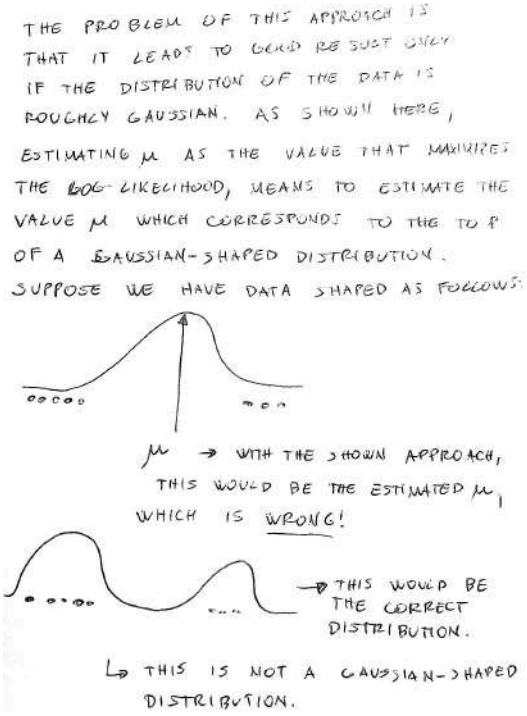
To clarify

What the professor is saying here is that knowing that parameters are probabilistic leads to the need to estimate their full probability distribution because it provides a more complete picture of the uncertainty and variability associated with the parameters, which is crucial for making informed decisions based on the model and to assess how well it generalizes. More in detail, when we say a parameter q is probabilistic, we mean that instead of being a fixed unknown value,
q is treated as a random variable with its own probability distribution. This reflects uncertainty in our knowledge about the true value of q. Instead of trying to estimate a single, best value of q, which assumes that there is a true fixed value that we can determine with enough data, we aim to estimate the entire probability distribution of q. This distribution represents all possible values that q could take, along with the likelihood of each value. *Reasoning in this way tells us not just what the most likely value of q is, but also how confident we can be in that estimate and what other values are reasonably likely.* As we will see, this approach is very powerful: other than a more nuanced understanding of the parameter, with this approach we can incorporate prior knowledge or beliefs about the parameter (Bayesian statistics) and allows to communicate uncertainty in predictions and decisions made based on the model

- We apply Bayesian reasoning to coin tossing!
- Our goal is the same as before: We aim to estimate the probability q that the coin comes up heads.
- Great difference to the ML approach: q is now itself *probabilistic*!
- Thus we estimate not a single value for q, but a *probability distribution* which describes possible values for q.
- This allows to model how uncertain we are about the true value of q.

Let's see closely how we compute the derivative of the log-likelihood w.r.t. *mu*, in order to maximise the likelihood function:





- The solution for $\mathbf{\Sigma}$ is a bit more involved. We limit ourselves to the one-dimensional case, for the full derivation see e.g. `https://stats.stackexchange.com/questions/351549/maximum-likelihood-estimators-multivariate-gaussian`.
- Assume μ is fixed. We have

$$\frac{\mathrm{d}\ell(\mu,\sigma)}{\mathrm{d}\sigma} = \frac{\mathrm{d}}{\mathrm{d}\sigma}\sum_{n=1}^{N}\left[-\log(\sqrt{2\pi}) - \log(\sigma) - \frac{1}{2}\frac{(x_n-\mu)^2}{\sigma^2}\right] = -\frac{N}{\sigma} + \sum_{n=1}^{N}\frac{(x_n-\mu)^2}{\sigma^3}.$$

- From the condition $0 = -\frac{N}{\sigma} + \sum_{n=1}^{N}\frac{(x_n-\mu)^2}{\sigma^3}$, we get by multiplying with σ^3 and dividing by N: $N\sigma^2 = \sum_{n=1}^{N}(x_n-\mu)^2$.
- Thus we get the ML estimate

$$\hat{\sigma}_{\mathrm{ML}}^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n-\mu)^2$$

where we can plug in $\mu = \hat{\mu}_{\mathrm{ML}}$ if μ is not known beforehand.

- In the D-dimensional case, the estimate is

$$\hat{\mathbf{\Sigma}}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(x_n-\boldsymbol{\mu})(x_n-\boldsymbol{\mu})^T$$

where we can likewise set $\boldsymbol{\mu} = \hat{\boldsymbol{\mu}}_{\mathrm{ML}}$.

- Note the similarity of the ML estimates of $\boldsymbol{\mu}$ and $\mathbf{\Sigma}$ with the definitions of the mean and the variance!

i.e. the datapoints are not distributed in a Gaussian-like way

here we are looking for the maximum likelihood of the variance, i.e. the value of *sigma* which maximize the log-likelihood. This estimate tends to underestimate the optimal variance of data, so its a little bit smaller. In general, we tend to underestimate the true variance.

precision (prior belief) with the information from the design matrix. I.e. it is a mixture of prior assumption (*alfa*l) and prior observation.

- The predictive distribution

$$p(t|\phi, \mathcal{X}, \mathcal{T}, \alpha, \beta) = \int p(t|\phi, \mathbf{w}, \beta) p(\mathbf{w}|\mathcal{X}, \mathcal{T}, \alpha, \beta) \mathrm{d}\mathbf{w}$$

can be evaluated similarly. The result is

$$p(t|\phi, \mathcal{X}, \mathcal{T}, \alpha, \beta) = \mathcal{N}(t|m_N\phi, \sigma_N^2(\phi))$$

where the variance $\sigma_N^2(\phi)$ is given by

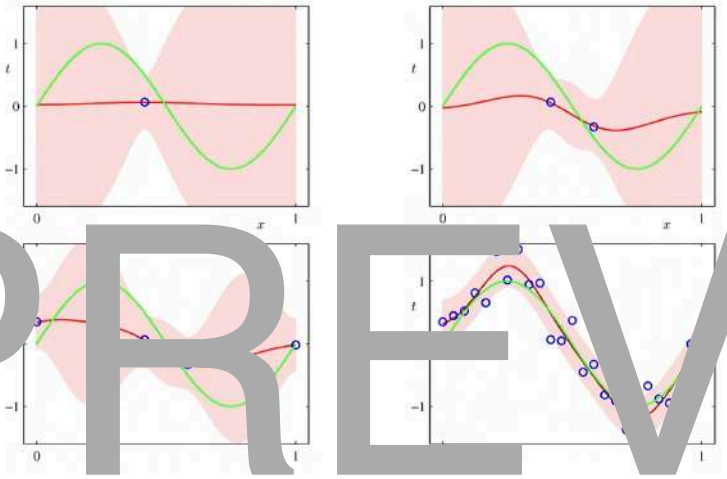
$$\sigma_N^2(\phi) = \boxed{\beta^{-1}} + \boxed{\phi^T S_N \phi.}$$

this is the information of how inexact is our target if we have got a feature.

This is derived from the training data.

- In particular, note that the variance of the prediction is given by two components: The inherent noise of the target (β^{-1}), and a term derived from the actual training data.

The variance of our prediction is given by two components



Examples for the predictive distribution, where the features are polynomial and the data is sampled from a sine wave. Shaded area indicates variance.

Red curve -> estimated, Green Curve-> actual, red-shaded area-> variance. If we got one sample, we could not even predict the samples from a curve, with two samples at least we get a rough tendency and the red shaded area (the variance) is largest the furthest from the training data. With four training data samples the prediction is still not good. With Bayesian approach I do not only get a prediction, but this distribution automatically gives you a prediction of how certain the prediction is given the concrete data we have.

LOGISTIC REGRESSION

- We have gotten to know logistic regression as a simple probabilistic model for *classification* (despite its name).
- Q: Under which conditions might such a simple model (remember that the decision boundary is linear) actually work well?
- Assume there are two classes c_0, c_1, with targets $t = 0, 1$, respectively. We have annotated data points in form of features, i.e. $\phi_1, \ldots, \phi_N$.
- Using Bayes' formula, we get, for any data point ϕ:

$$p(c_0|\phi) = \frac{\boxed{p(\phi|c_0)p(c_0)}}{\boxed{p(\phi|c_0)p(c_0) + p(\phi|c_1)p(c_1)}} = \frac{1}{1 + e^{-a}} = \sigma(a)$$

with the logistic sigmoid σ and

$$a = \ln \frac{p(\phi|c_0)p(c_0)}{p(\phi|c_1)p(c_1)}.$$

(It is not difficult – try this at home.)

bayes formula

probability of *phi*
the probability of *phi* is then the probability of *phi* given c0 + the probability of *phi* given c1. This can be done because c1 and c0 do not overlap and together they make up the entire space.

Now we have to make the following assumptions:

- Assume that the data for classes c_0 and c_1 is Gaussian distributed with the *same* covariance matrix Σ, and means μ_0, μ_1.
- The data model is

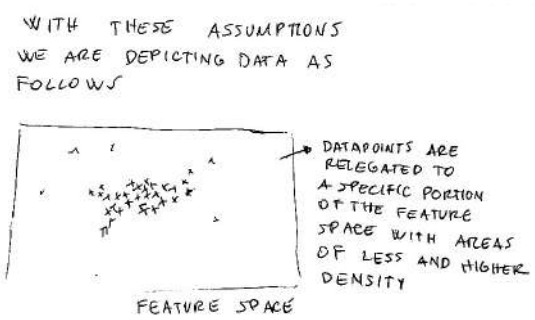
$$p(\phi|c_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\phi - \mu_k)^T \Sigma^{-1} (\phi - \mu_k)\right).$$

- Defining

$$\mathbf{w} = \Sigma^{-1}(\mu_0 - \mu_1)$$
$$w_0 = -\frac{1}{2}\mu_0^T \Sigma^{-1} \mu_0 + \frac{1}{2}\mu_1^T \Sigma^{-1} \mu_1 + \ln \frac{p(c_0)}{p(c_1)}$$

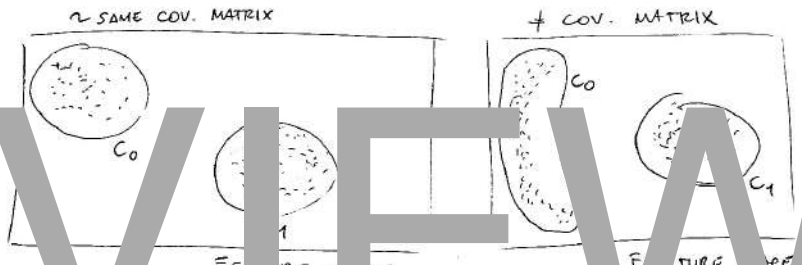
one obtains the surprising result

$$p(c_0|\phi) = \sigma\left(\ln \frac{p(\phi|c_0)p(c_0)}{p(\phi|c_1)p(c_1)}\right) = \sigma(\mathbf{w}^T \phi + w_0).$$



we basically get that p(c0 | *phi*) is the sigmoid of a linear mapping with bias.

The key concept is that when we do not have any idea (or not a clear idea) of the data, we always assume/estimate them to be roughly-gaussian.
Clearly this linear regression model works well if the covariance matrices of the data are roughly the same. What the professor is saying here, is the following: the covariance matrix represents how the features of the data vary together.
If both classes have the same covariance matrix, it means they have the same shape of the data distribution, though they may be centered at different locations (different means). Under the assumption of equal covariance matrices the boundary that separates the classes will be linear (a hyperplane). This makes the problem solvable by finding a linear combination of features that best separates the classes, which is what the vector **w** represents in the formula.



- Thus we have recovered the original definition of Logistic Regression by directly computing the probabilities for classes c_0 and c_1, under the assumption that the data for those classes is Gaussian distributed with identical covariance matrices.
- In this case, the decision boundary is linear.
- Note how the prior probabilities of the classes affect the bias w_0!
- The method does not work if the class covariances are different, or the distributions are not Gaussian (the decision boundaries might not be linear any more).
- How to compute the parameters for Logistic Regression? There is no closed-form solution (not even for the Maximum Likelihood case), one could for example use gradient descent to find an approximate solution, just as we have done for neural networks.

- We have traded the simplicity of an analytic solution for a rather restrictive model for an *iterative* approach to train a far more powerful model.
- As with iterative training of neural networks, we need a criterion to determine when to stop the iteration.
- Usually, one considers the (log-)likelihood of the training or validation data and stops training when it appears to converge.
- The EM algorithm for Gaussian mixtures normally converges quite fast.
- In contrast to NN training, each step (E or M) *always increases* the likelihood of the data!

The only methods that find an optimal solution are single gaussian and linear regression. For all other problem I need to approximate the optimal solution.

- In specific cases, training may fail; in particular, if the data is in a low-dimensional subspace, the variance of the data approaches zero in one more directions, leading to degenerate distributions.

⇒ Apply *dimensionality reduction* to the features.

- This can also occur if only few samples are assigned (with high γ) to a component: The component becomes "sharper" (small variance in many directions), which causes even less samples to be assigned to this component, which in turn increases the problem.

⇒ One usually deals with such situations heuristically (e.g. deleting a Gaussian if its combined assignment probabilities become too small).

⇒ Alternatively, this kind of overfitting can be avoided with a Bayesian approach to the GMM training.

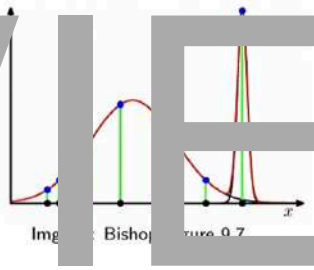
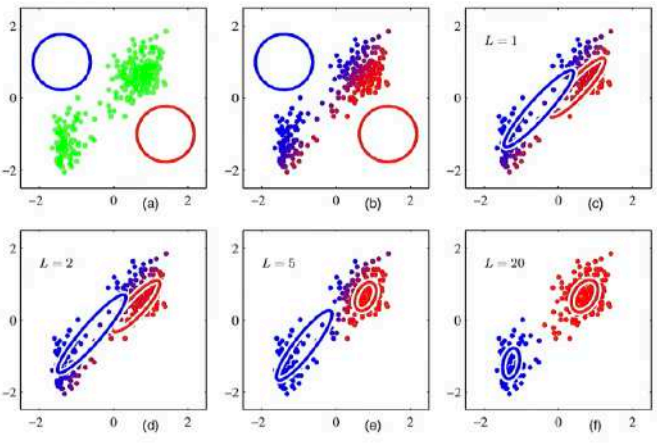


Image: Bishop, figure 9.7



EM for two Gaussians in two dimensions. Note that the Gaussians start to take a reasonable shape after just 5 iterations.

How EM works (2-D case).
The sequence of panels shows how the EM algorithm iteratively updates the parameters of the Gaussian distributions to better fit the data points. Starting from an initial random guess, note that the two ellipses illustrate the confidence intervals of the Gaussian distributions. They typically represent regions where the probability density is above a certain threshold. While the colour of the datapoint likely represent data points that the algorithm currently assigns to one Gaussian distribution or the other.

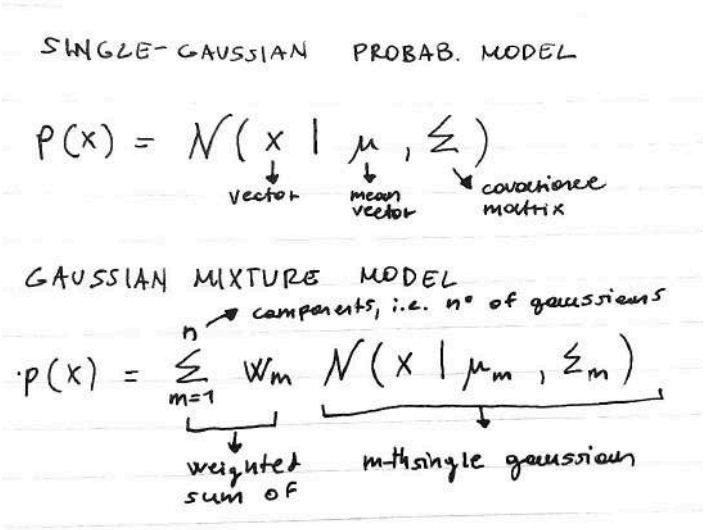
In general, then higher the number of datapoints, the less the model tend to overfit. The higher the number of gaussian, the higher is the probability that it overfits.

Machine Learning - Part 2 Lesson 6
23.11.2023

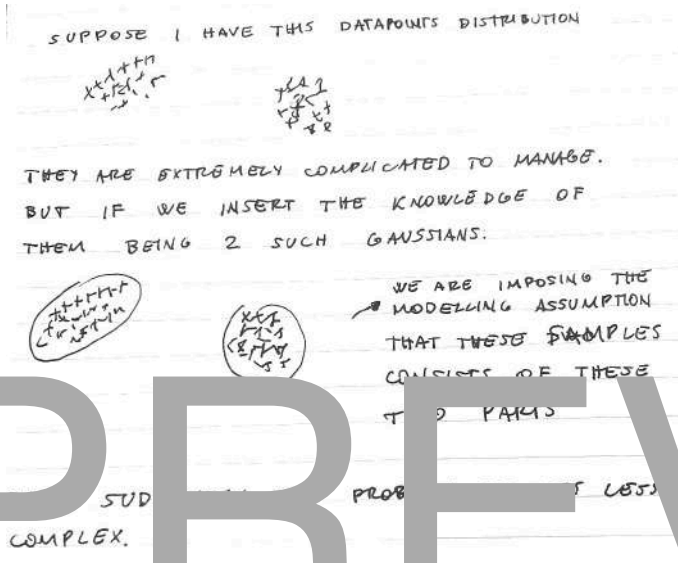
Recap of Gaussian Mixture Models, Latent Variables, and EM Training

The Em Algorithm

The idea of the EM algorithm is to perform approximate-maximum-likelihood of GMMM (Gaussian Mixture Models).



note that we write **x** and not $\mathbf{x}_m$ because the Gaussian Mixture probabilistic model here works on the same training data, we are not working on specific parts of it, but the model takes all the training data x, not a specific part for each gaussian.



Latent Variables

We introduce latent variables to make our probabilistic models more straightforward to understand and therefore easier to compute. In short, latent variables are used to describe complicated probability distribution to make them simpler. Latent variables are used to make the description of our data *simpler,* thus more human-like and more mathematically analytically computable. Drawn example, the latent variable is to assume that each sample is assigned to one of the components gaussians, but we do not know how (and we either know the assignment). Since we don't have the latent variables, we replace them with probabilities, with the posterior probabilities given the previous estimation of our parameters.

To clarify

Latent variables are not directly observed but are inferred from the models built on the observable data. They are introduced into a model to capture certain aspects that are not directly observable or to simplify the structure of the data.

In the context of Gaussian Mixture Models (GMMs), the latent variables represent the component assignment for each data point. Instead of trying to model the entire data distribution directly, which might be very complex due to the mixture of multiple subpopulations within the data, we introduce latent variables to indicate which subpopulation (or Gaussian component) each data point is most likely to come from. The use of latent variables turns an otherwise intractable problem into one that can be solved iteratively using the EM algorithm, providing a powerful way to fit complex models to data. As mentioned, in the context of GMMs, the latent variable for a particular data point would explicitly state which Gaussian component that point is drawn from. However, we don't observe these assignments in our data; they are "hidden" from us. Instead of knowing the latent variables outright, we calculate the posterior probabilities. These probabilities represent our best guess, given the current parameters of the model, of which component a data point is likely to come from.

- **E-step:** In the **E-step** of the EM algorithm, we compute the posterior probabilities for each data point and for each component. For a data point x_n, we calculate the probability that this point was generated by the m-th Gaussian component given the data point and the current model parameters θ^{old}. This is mathematically represented as:

$$p(z_{n,m} = 1 | x_n, \theta^{old}),$$

where $z_{n,m}$ is the latent variable indicating the component assignment, x_n is the observed data point, and θ^{old} represents the current estimates of the model parameters. These posterior probabilities are calculated using the following formula:

$$p(z_{n,m} = 1 | x_n, \theta^{old}) = \frac{p(x_n | z_{n,m} = 1, \theta^{old}) p(z_{n,m} = 1 | \theta^{old})}{\sum_{k=1}^{M} p(x_n | z_{n,k} = 1, \theta^{old}) p(z_{n,k} = 1 | \theta^{old})}.$$

The numerator computes the likelihood of the data point x_n under the assumption that it came from the m-th component, weighted by the prior probability of the m-th component. The denominator normalizes this value by summing over all possible component assignments.

These probabilities effectively "replace" the unknown latent variables during the computation. They are used in the subsequent Maximization step to update the parameters of the model in such a way that the expected likelihood of the data, given these updated parameters, is maximized. The EM algorithm iterates these two steps — first computing the posterior probabilities (E-step) and then updating the parameters based on these probabilities (M-step) — until the changes in the parameters are below a certain threshold and the algorithm has converged.

Gaussian Mixtures: Applications

- We hav… …t the GM… …d t… …escribe the …ruc…e …f the d…a with a s…ll se… …f parameters (μ … Σ_m, w_m
- T…s fal… into the c…gory … Unsupervised L…rning, w…ch we … …ot cov…
- We now …onsider ho… to u… GMMs in supervise… ap…cations, namely …assificatio… …nd re…ession.

Let's see an example of supervised regression.

Gaussian Mixtures: Regression

- Goal: supervised regression from multidimensional inputs $\mathbf{x} \in \mathbb{R}^N$ to multidimensional targets $\mathbf{t} \in \mathbb{R}^K$.
- Train a *joint* GMM on the combined observations

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{t} \end{pmatrix},$$

which yields a joint model with component means and covariances structured like this:

$$\mu = \begin{pmatrix} \mu^{\mathbf{x}} \\ \hline \mu^{\mathbf{t}} \end{pmatrix} \qquad \Sigma = \left(\begin{array}{c|c} \Sigma^{\mathbf{xx}} & \Sigma^{\mathbf{tx}} \\ \hline \Sigma^{\mathbf{xt}} & \Sigma^{\mathbf{tt}} \end{array}\right)$$

(note that $\Sigma^{\mathbf{xt}} = (\Sigma^{\mathbf{tx}})^T$).

- You can easily reduce this model to a model for $\mathbf{x}$ only by considering the relevant parts of the mean vectors and covariance matrices ($\mu^{\mathbf{x}}$, $\Sigma^{\mathbf{xx}}$), for each component Gaussian, respectively.

- Regression for a new value $\mathbf{x}_0$ is performed by
  - computing weights w_m, using *only* the model for $\mathbf{x}$, as

$$w_m = \frac{\mathcal{N}_m^{\mathbf{x}}(\mathbf{x}_0)}{\sum_{\ell=1}^{M} \mathcal{N}_\ell^{\mathbf{x}}(\mathbf{x}_0)}$$

  - computing the w_m-weighted average of the expected values of the *conditional* distributions of $\mathbf{t}$ given $\mathbf{x}$:

$$\hat{\mathbf{t}} = \sum_{m=1}^{M} w_m E[\mathbf{t}|\mathbf{x} = \mathbf{x}_0, \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m]$$

weighted um of conditional expectations for each gaussian

where the conditional probabilities of the components take the form of Gaussians with expected values

$$E[\mathbf{t}|\mathbf{x} = \mathbf{x}_0, \boldsymbol{\mu}, \boldsymbol{\Sigma}] = \boldsymbol{\mu}^{\mathbf{t}} + \boldsymbol{\Sigma}^{\mathbf{xt}}(\boldsymbol{\Sigma}^{\mathbf{tt}})^{-1}(\mathbf{x}_0 - \boldsymbol{\mu}^{\mathbf{x}}).$$

To clarify

Gaussian Mixture Models (GMMs) can be utilized for regression by predicting a continuous response variable t given a new input x_0. The process is as follows:

1. **Computing Weights w_m**: For a new observation x_0, we compute the weights w_m corresponding to each Gaussian component in the mixture. The weights are computed using:

$$w_m = \frac{N'_m(x_0)}{\sum_{\ell} N'_\ell(x_0)}$$

where $N'_m(x_0)$ is the probability density function of the m-th Gaussian evaluated at x_0.

2. **Weighted Average of Expected Values**: The estimated response $\hat{t}$ is calculated as the weighted average of the expected values of the conditional distributions of t given x:

$$\hat{t} = \sum_{m=1}^{M} w_m E[t|x = x_0, \mu_m, \Sigma_m],$$

where $E[t|x = x_0, \mu_m, \Sigma_m]$ is the expectation of t for the m-th Gaussian component conditioned on x_0.

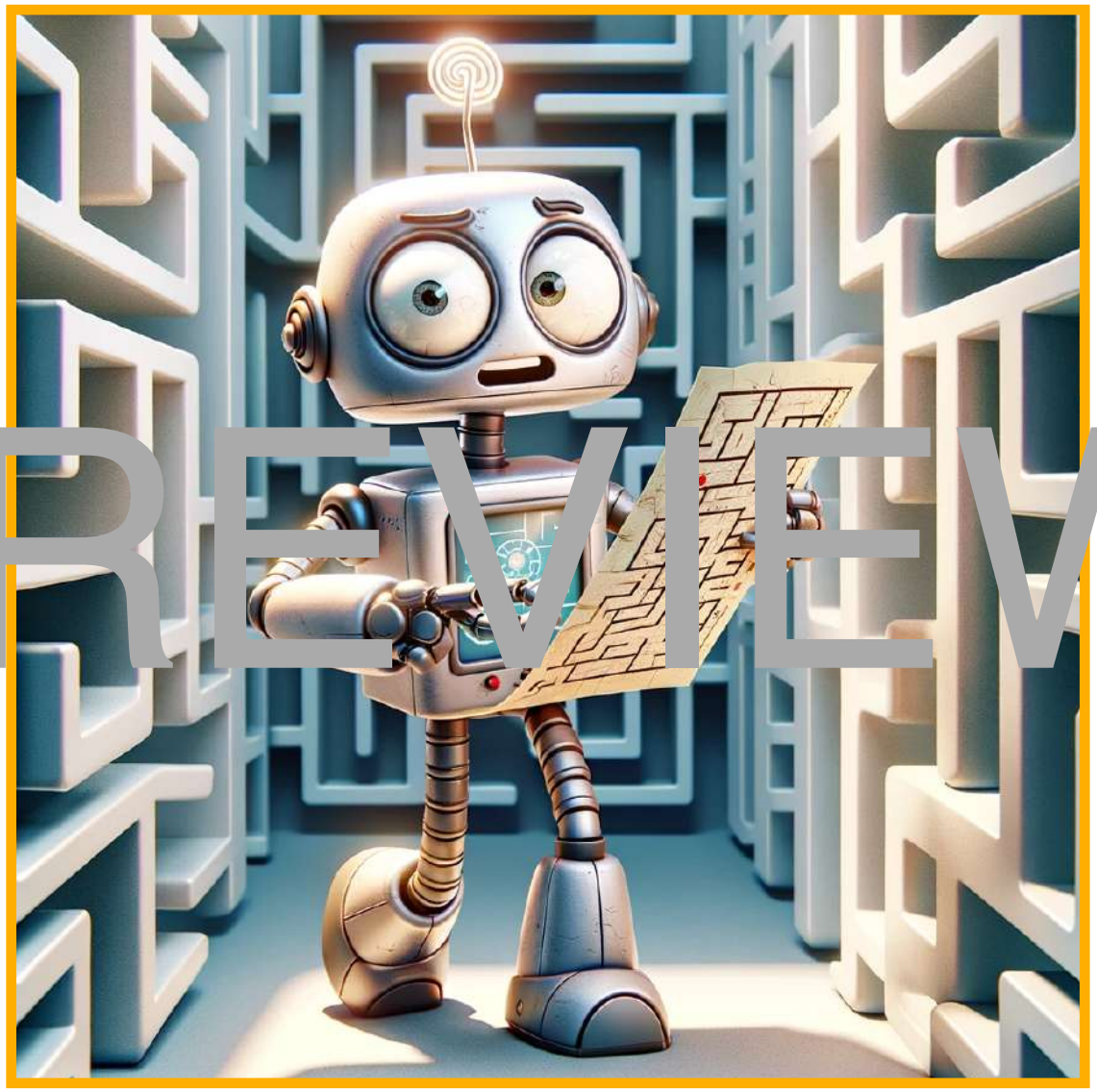
3. **Conditional Expectations**: The conditional expectations for the Gaussian components are calculated as:

$$E[t|x = x_0, \mu, \Sigma] = \mu^t + \Sigma^{xt}(\Sigma^{tt})^{-1}(x_0 - \mu^x),$$

with μ^t and μ^x being the mean vectors for t and x, respectively, and Σ^{xt} is the cross-covariance matrix between x and t.

This method provides a flexible approach to regression that accounts for the multimodal nature of the data distribution and allows for capturing complex relationships between the input x and the output t.

REINFORCEMENT LEARNING



Machine Learning - Lecture 7
Reinforcement Learning 1
28.11.2023

Reinforcement Learning Introduction

Deals with how an *agent* takes *actions* in some kind of *environment*

- learn from experience
  - generate data by interacting with the environment
  - accumulate asynchronous rewards (not known which action led to which reward, no explicit error correction).
- Often only partial information is available.
- Inherently sequential method, focus on online capabilities.

The goal of RL is to learn from experience, not so much on having specific instructions.
The agents receives rewards with are scalar values (i.e. numbers) between -inf to +inf. The goal is to maximise the long term reward (called **return**). The reward is less precise than a target vector (as in regression tasks) -> the reward might give us the information that the agent performed bad, but is far less precise. To clarify, in the target vector each target is the precise information your system/model has to reach, in RL, the reward just tells wether the agent performed well or bad, it contains no informations on what went wrong, it only states how good/ bad the agent performed.
Moreover, rewards are . Let's make an example to clarify. Suppose we are training a chess player. In chess the environment is clearly defined and finite (the board is defined and can handle finite number of possible sequences of actions). So the chess-table is the environment and the moves of the player are the actions. This environment also has some sort of randomness since we do not know how the opponent is going to behave. The main thing to understand is that, in this chess example, we have a series of actions (which is the sequence of actions taken by the agent\player). Having in mind this example, how can we assign reward? The reward should be given at the end of the game, when the player wins or gets a draw (pareggio), i.e. the agent takes a series of moves/action and only at the very end it receives a reward. It doesn't mean how many and what ieces capture ng at matters o w Nov em merges: v nee to define v ch a ons/moves on t way we good nd w ch were bad. In ther s atio other envi nmen s) the agent may e given eward nore equently (and no at the d), b we alwa suppo rds are iven a **ynchro ously** ries o ctio s you ge reward. hen v get a reward it do n't ean that spec c action was goo or ad, but it m ans that so ethin along the way happ ne which w goo or bad (rewards m y e negativ
It is important to notice the following issues:
-environment may be undefined (e.g. is so large and complex it cannot be fully defined)
- the agent/robot might have got partial informations (e.g. the robot maybe cannot look at the back unless he takes some actions to turn around, it may do not recognise everything he sees)

Examples of RL tasks

A few typical sequential decision tasks solved by Reinforcement Learning (RL) ...

- Autonomous robotics
- Controlling chemical processes
- Network routing
- Game playing
- Stock trading

- Game playing: Racing [Koutnik et al. 2013], diverse ATARI games [Mnih et al. 2015], Dota 2 [Brockman et al. 2019], Chess [Silver et al. 2018]

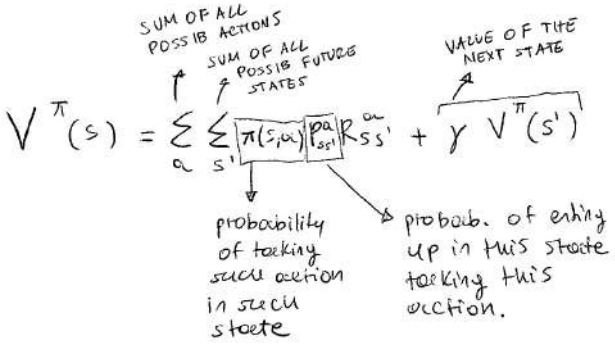
Chess, Racing Games, ATARI Games and Dota 2 were all solved by RL. But also robotic tasks such solving a Rubic's cube.

- Robotics: Rubic's Cube Manipulation [Brockman et al. 2019]

actual value function for action a, I sum over all possible future state. $P^a_{ss'}$ are the transition probabilities to end up in the future state given the action. Just plugging in one into the other we get the Bellman Equations:

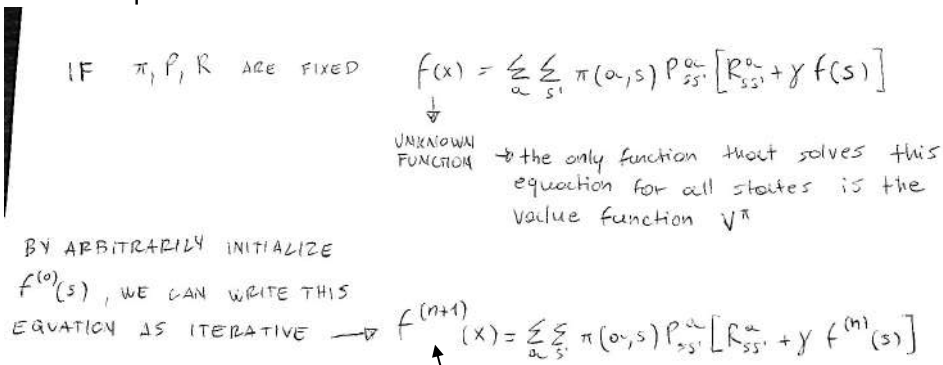
Value functions satisfy the following *important* recursive relation:

$$
\begin{aligned}
V^\pi(s) &= E(G_t|S_t = s) = E\left[\sum_{k=0}^{H} \gamma^k R_{t+k+1} \middle| S_t = s\right] \\
&= \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P^a_{ss'} \left[R^a_{ss'} + \gamma E\left(\sum_{k=0}^{H} \gamma^k R_{t+k+2} \middle| S_{t+1} = s'\right)\right] \\
&= \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right]
\end{aligned}
$$



This is called the *Bellman equation* for the value function.

With the policy fixed, the value function is the unique solution to the Bellman equation.



The fact that the value function is the only solution for this, has the consequence that:

Bas… lm… Equ… ion, the estima… n of th… Valu… Function can b… easte… as a fi… d-point problem

The Bel… an equa… is fun… mental for RL opti… zatio… both fr… a theoret… l and a pra… ical p… spective.

$$V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(s,a) \sum_{s'\in\mathcal{S}} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right]$$

(jus… hat sho… n above)

A similar recurrence can be found for the action-value function:

$$Q^\pi(s,a) = \sum_{s'\in\mathcal{S}} P^a_{ss'} \left[R^a_{ss'} + \gamma \sum_{a\in\mathcal{A}} \pi(s',a) Q^\pi(s',a)\right]$$

We have now a way to quantify the quality of possible states and actions (for fixed policies).

To clarify:

Value Function: The value function $V^{\pi}(s)$ encapsulates the expected total reward an agent can accumulate over the future, starting from state s and thereafter following a policy π. This function is critical because it provides a quantitative measure of the "goodness" or "value" of a state, which helps the agent make decisions that maximize its long-term reward. To understand the value function deeply, consider that it is a prediction of future rewards. These rewards are uncertain and depend on both the policy the agent follows and the stochastic dynamics of the environment. The value function is a sum of the expected rewards, with each future reward discounted by a factor γ raised to the power of the time step at which the reward is received. The discount factor γ represents the preference for immediate rewards over distant ones and ensures that the sum of the future rewards converges to a finite value.
Recall that a policy π is a function from states *S* to actions A. For each state s, the policy π outputs a single action to perform (if we implement a **deterministic** –greedy– policy) or a probability distribution over actions (if we implement a stochastic policy). The policy is what defines the behavior of the agent in the environment, by providing him what the "best" action is (i.e. the action which is estimated to return the highest reward).

Action-Value Function: Similarly, the **action-value function** $Q^{\pi}(s, a)$ tells the agent the expected return of taking action a in state s and thereafter following policy π. In short, it evaluates how "good" it is to perform a particular action *a* in a particular state *s*. Similar to the value function, "good" means the expected total reward the agent can accumulate *from this point on*, but here it's *conditional* not just on the starting state *but also on the first action taken*. While the value function gives you the expected return from a state if you follow a certain policy, the action-value function gives you the expected return from a state taking a certain action and then following a certain policy. It gives a value for every possible state-action pair, representing the expected long-term reward of taking action *a* in state *s* and then following policy π.
It is defined as:

$$Q^{\pi}(s, a) = E[G_t | S_t = s, A_t = a]$$

which i[illegible] ward i[illegible] on a in stat[illegible]

[illegible]lati[illegible]ip betwee[illegible]alue [illegible]nction and Action-V[illegible]ue Fun[illegible]on: The [illegible]lation[illegible]ip between the valu[illegible] function[illegible] ue func[illegible]n is crucial for under[illegible]ndin[illegible]ow to co[illegible]ute o[illegible] from the other. The value fu[illegible]tion can be [illegible]presse[illegible]n terms of the action-v[illegible]e [illegible]nction as f[illegible]ows:

$$[illegible] = \sum_{a \in A} \pi(s[illegible] Q^{\pi}(s, a)$$

where $\pi(s, a)$ is the probability of taking action a in state s under policy π, and A is the set of all possible actions.
Conversely, the action-value function can be computed from the value function by looking one step ahead into the future. This is expressed by the following equation:

$$\begin{aligned} Q^{\pi}(s, a) &= E\left[\sum_{k=0}^{H} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \\ &= \sum_{s' \in S} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma E\left[\sum_{k=0}^{H} \gamma^k R_{t+k+2} | S_{t+1} = s'\right]\right] \\ &= \sum_{s' \in S} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s')\right] \end{aligned}$$

where $P^{a}_{ss'}$ is the probability of transitioning to state s' from state s after taking action a, and $R^{a}_{ss'}$ is the immediate reward received after transitioning from s to s' due to action a. The term $\gamma V^{\pi}(s')$ represents the discounted value of the next state s', and the summation over all possible next states S accounts for the stochastic nature of the environment. What it matters here is to learn that:
$Q^{\pi}(s, a) = \sum_{s' \in S} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s')\right]$.

Recursive value function: the main thing to understand here is that, using Bellman equation, we can define the Value Function recursively (so as a recursive function). What you need to learn here is that the recursive formula for value function that we get by using Bellman, is the following:
$V^{\pi}(s) = \sum_{a} \sum_{s'} \pi(s,a) P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V^{\pi}(s') \right]$, where:

- s represents the current state of the agent.
- a represents an action that the agent can take.
- s' represents a subsequent state that the agent can transition into after taking action a.
- $\pi(s,a)$ is the probability of taking action a in state s under policy π.
- $P^{a}_{ss'}$ is the probability of transitioning from state s to state s' after taking action a.
- $R^{a}_{ss'}$ is the immediate reward received after taking action a in state s and moving to state s'.
- γ is the discount factor, which is used to reduce the value of future rewards compared to immediate rewards.
- $V^{\pi}(s')$ is the value of the next state s' under policy π.
- The outer summation over a takes into account all possible actions the agent can take in state s.
- The inner summation over s' takes into account all possible subsequent states that could result from each action.

Advantages of this recursive formulation: recall that the Bellman equation expresses the value of a state as a combination of the immediate reward from an action taken in that state plus the future rewards that follow, discounted by a factor γ. We typically do not know the values of any states upfront. However, we can make an initial guess (often starting with zeros or random values) and then use the Bellman equation to improve these guesses. Through iteration, we use our current estimates of the future states' values to update the current state's value. This process is repeated, with each iteration refining the values based on the latest estimates. As this iterative process continues, the values of the states begin to stabilize. This means that the difference between the value estimates from one iteration to the next becomes very small, signaling that we are approaching the true values: after several iterations, the iterative value function will reach a point where subsequent applications of the Bellman equation produce no changes — this is the **fixed point**. Lastly, it is important to understand also that, given a specific policy π, there is a unique value function $V^{\pi}(s)$ that satisfies the Bellman equation for all states. At the fixed point, the value function accurately reflects the expected rewards of following the policy from every state. If policy π is optimal, then the fixed point corresponds to the optimal value function $V^{*}(s)$. Basically, at the fixed point, we can be confident that the values accurately represent the expected rewards for the policy being evaluated.

Sampling-Based Approximations

- We have gotten to know two “exact” DP methods: *Value Iteration, Policy Iteration*
  - They are exact in the sense that we aim at computing the true value function / optimal value function from the model.
  - Note that knowing the model, we never needed to actually “run” the agent (i.e. generate actual experience)!
- If we do not have access to the model, we need to
  - a) generate actual experience of the environmental dynamics
  - b) adapt our methods such that they work with samples from the environment.

with model-based RL, it is all on planning, the agent has never to actually be active, we never actually run the simulation.

- Bellman equation:

$$V^{\pi}(s) = \sum_{a} \sum_{s'} \pi(s,a) P^{a}_{ss'} [R^{a}_{ss'} + \gamma V^{\pi}(s')]$$

- Policy iteration:

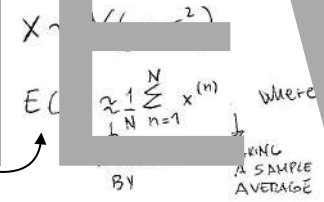
$$V_k(s) \leftarrow \sum_{a} \sum_{s'} \pi(s,a) P^{a}_{ss'} [R^{a}_{ss'} + \gamma V_{k-1}(s')]$$

- Replace weighted sum by expectation:

$$V_k(s) \leftarrow E_{a,s'} [R^{a}_{ss'} + \gamma V_{k-1}(s')]$$

here the expectation is taken over the probability distribution of actions and states derived from policy and environment.

always if you see a weighted sum with probability distribution, that is an expectation. What can I do with expectations now? How to approximate an expected value from samples? The prototypical example where this was given I am going to estimate a mean vector from a gaussian distribution. The expectation of a Gaussian random variable is its mean. Similarly, in RL, we can approximate the expected value of the return by taking an average of the returns we observe from sampled transitions (actions and resulting states and rewards), rather than relying on known probabilities and rewards.



- We know that expectations can be replaced by sampling!
- For sampling-based Policy Iteration, let us assume that we generate a series of sampled returns $G_{\text{smp}}(s)$ from some state s (will soon see how).
  - Requires that all states are actually reached.
- Then

$$\hat{V}(s) = \frac{1}{N} \sum_{n=1}^{N} G^{(n)}_{\text{smp}}(s)$$

- or, written as a running average:

$$\hat{V}(s) \leftarrow (1-\alpha)\hat{V}_{\text{old}}(s) + \alpha G_{\text{smp}}(s)$$

where the estimate gets updated whenever a new sample arrives. α can be considered the learning rate.

I update the estimate of the value of state s, using some kind of learning rate/update rate *alpha* multiplied by the old estimate and sum to the observed return. With the correct choice of *alpha,* this should be pretty the same as this.

source: Sutton/Barto

So, after having performed action a_t, I have reached state s_{t-1}. I have received the reward, and now, for my state s_{t-1} , I look at my estimate of all possible future actions, and I maximise over this.

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal

To clarify:

Q-learning update rule. Q-learning uses the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

In this formula:
- $Q(S_t, A_t)$ is the current estimate of the action-value function for state-action pair (S_t, A_t).
- α is the learning rate.
- R_{t+1} is the reward received after taking action A_t in state S_t.
- γ is the discount factor, which balances the importance of immediate and future rewards.
- $\max_a Q(S_{t+1}, a)$ is the maximum estimated value over all possible actions in the next state S_{t+1}, which represents the best expected future reward attainable from the next state.

Q-learning can learn the optimal policy even if the agent is not following it. This is because it always considers the best possible action to take in the next state, regardless of the policy's actual choice. During each update, Q-learning considers the immediate reward received after taking an action A_t in state S_t and the estimated value of the best possible action in the subsequent state S_{t+1}. This is the $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ part of the Q-learning update rule. The term $\max_a Q(S_{t+1}, a)$ represents the agent's current estimate of the best possible future reward from the next state S_{t+1}, looking one step ahead. This estimate implicitly includes the expected future rewards from all subsequent steps. That's because each Q-value is an estimate of the total expected return from taking an action in a given state and following the optimal policy thereafter.

In summary, while Q-learning updates are based on immediate next-step rewards and the estimated value of subsequent states, the action-value function it learns represents the expected cumulative future rewards when following the optimal policy.

Behavior Policy. This is the policy that the agent actually follows while interacting with the environment. It dictates the agent's actions and is used to generate behavior for learning. The behavior policy is typically designed to ensure sufficient exploration of the state-action space. For example, an ε-greedy policy, which mostly exploits the best-known action but sometimes explores randomly, can be used as a behavior policy.
Target Policy. This is the policy that the agent is learning about and trying to improve. The target policy is the one that the agent aspires to optimize – it is the policy that the agent believes will yield the highest long-term rewards. The key point in off-policy methods is that the target policy can be different from the behavior policy.

Variant of SARSA: expected SARSA
(professor just mentioned it, didn't explain it in depth).

- Another variation of the TD update rule consists in taking the *expectation* over possible next actions.

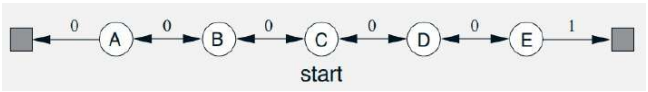
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\right]$$

$$\leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\right],$$

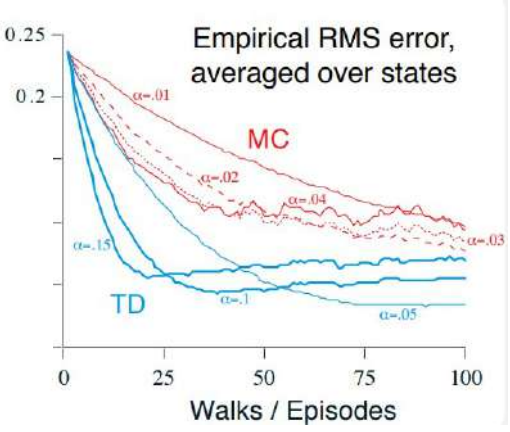
- The resulting algorithm (*Expected SARSA*) is slightly more complex than SARSA, but in general converges better.

Further Remarks and Conclusions on RL part

- TD Advantage: easy and natural to implement online.
  - This also works well when the value function is only approximated (e.g. with a neural network, next lesson).
- Convergence guarantees are possible (notably, requires to choose the learning rate well)
  - ...but best of all, it works well in practice!
- Usually faster convergence than MC methods.
- Example (Sutton/Barto) for a *Markov Reward process* (no actions, just learn environment).



- Start at position C, go left or right with equal probability, two terminal states
- Only reward: +1 in right terminal state
- No discounting: value of state is probability of ending up in the right terminal state
- TD faster and usually better than MC
- TD fluctuates less



Why are TD estimates typically better than MC?

- Consider the paths for which state values are estimated.
- The TD algorithm *averages* values over possible paths, thus giving improved estimates.
- Right panel: The two light blue paths (up to the crossing) are identical in their outcome, but MC (second panel from left) will not detect this.

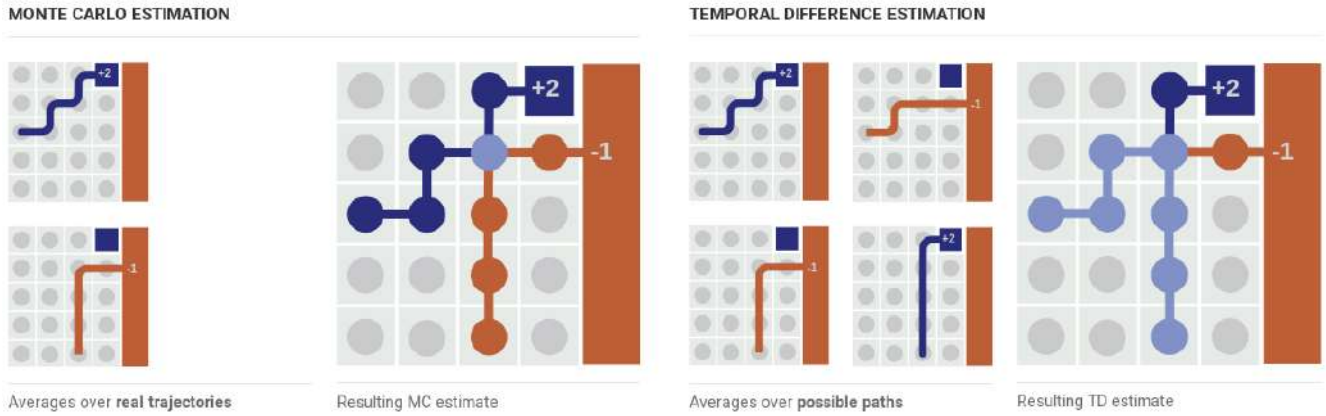


Image. The task here is to move across the great. If u reach the orange pit you get a -1 reward, if u get to the blue square you get a +2 reward. We can see two trajectories, one ending up in the pit, the other on the blue squares. The main thing to understand is that with TD I get an average over many different returns fundamentally way faster. The light blue are the averaged-return paths.

Monte Carlo Estimation. MC methods estimate the value of a state by averaging the returns following real trajectories (episodes) that start from that state and go until the end of the episode. The value of a state is the average of these returns. However, MC methods can only update the value of a state at the end of an episode, which means they don't leverage the information gained during [illegible] until it's [illegible]

Temporal Difference Estimation. TD methods, on the other hand, update value estimates based on observed rewards and the estimated values of subsequent states partway through the episode. This is done every [illegible] not at the end of the episode. [illegible] even if two paths diverge after a certain point, the value of the states up to the divergence point is informed by both paths. The learning uses the existing value estimates of subsequent states to update the current state's value. This allows for more frequent and incremental updates, which can lead to faster learning.

MC vs TD. With MC, only the final outcomes (reaching a reward or a pit) are considered, and since the paths are part of separate episodes, the value estimate for the crossing point doesn't benefit from the shared trajectory. With TD, instead, the shared part of the trajectories informs the value estimate of the crossing point, as the value estimate is updated every step, not just at the end of the episode.This is beneficial because it means that TD learning can quickly adjust to changes and incorporate new information as it becomes available, rather than waiting until the end of an episode as MC does. It is particularly useful in environments where certain states are visited through various paths, allowing TD methods to average out the experiences from different trajectories to converge on a stable estimate for the state's value.