The Document

João Morais

October 25, 2019

Contents

1 Resources & Hand Written Notes

check?

3 Artificial Intelligence / Machine Learning

3.1 Artificial Intelligence

There are some goal states and one initial state. The objective is to find the goal state that is closer (with the shortest path to the root/ with the least search cost). Is given as the solution the steps necessary to perform that path.

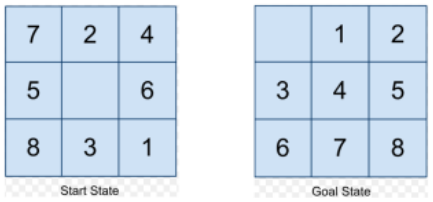
To keep the formulation as general as possible, abstractions are required, keeping in mind that they will need a correspondence when applied to a real problem.

Example of the application to a problem: a vacuum cleaner that needs to clean every square. In this case, only 2 squares are presented, no localization sensors are present, only "rubbish" sensors that tell if the current square is dirty or not.

Therefore:

- States : $< r, d_1, d_2 >$ where r is the robot position, d_1 and d_2 are binary, representing the existence of dirt in each of the rooms.
- Operators/Actions : L (go left), R (go right) or S (suck dirt)
- Goal Test : d_1 = False and d_2 = False. $((d_1 \text{ nor } d_2) == 1)$
- Initial State : $< r, d_1, d_2 >=< 1, T, T >$ is at square 1, and squares 1 and 2 are dirty
- Step Cost : the description of each action cost. In this case, 1 for each one.

8-puzzle



States: specify in a 3x3 matrix the location of the 8 numbered tiles plus the blank one

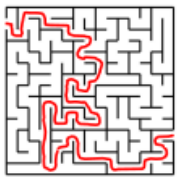
Initial state: any puzzle configuration

Operators: move "blank" to the left, right, up and down

Goal test: checks whether the state matches the goal configuration

Path cost: each step costs 1 (so path cost = number of steps)

Mazes



States: maze configuration plus current location

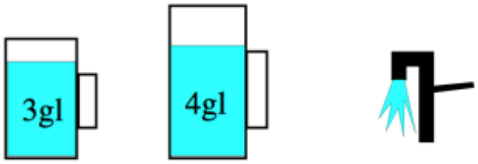
Initial state: any maze location

Operators: move to an adjacent and linked position

Goal test: current location = maze exit

Path cost: number of steps

Water jugs



States: amount of water in each jug (e.g., <1,4>)

Initial state: <0,0>

Operators: fill J4; fill J3;
empty J4; empty J3;
pour water from J4 into J3 until J3 is full or J4 is empty;
pour water from J3 into J4 until J4 is full or J3 is empty;

Goal test: <0,2>

Path cost: amount of water used

Hanoi towers

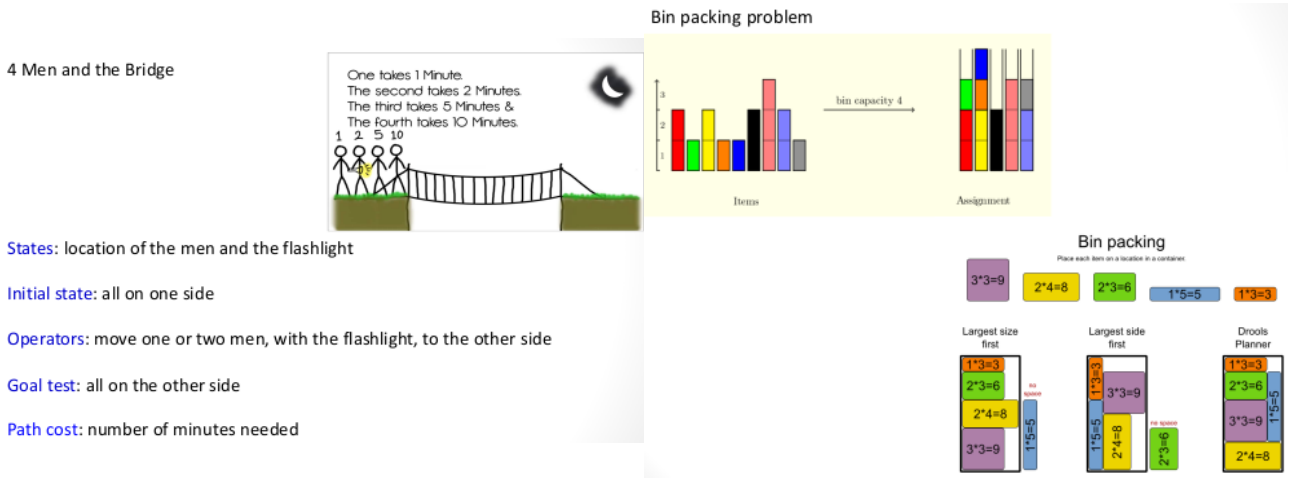
States: location of the disks in the three poles

Initial state: all disks correctly arrange in the left pole

Operators: move a free disk to another pole, which is empty or all disks in it are bigger

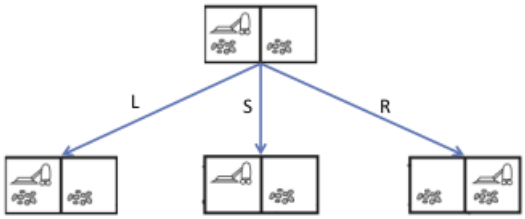
Goal test: all disks correctly arrange in the right pole

Path cost: number of disk moves

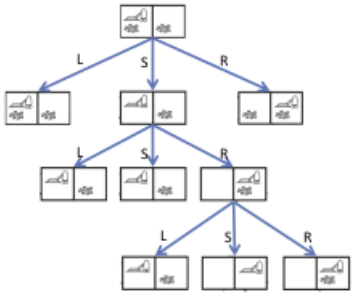


As you can see, every problem is very well defined in terms of the start, the end, the possible moves and what makes a solution better than other. Therefore, the conditions to put the computer thinking how to get from the possible steps to the best solution are assembled. One way the computer should be able to get to the solution is by exploring every move combination and check the state it ended up with.

Given the initial state, use the operators to generate successor states.



Then a choice of which is more "profitable" or more likely to lead to a goal state needs to be made and this cycle continues until the arrival at a goal state.



In this case, the solution would be $\{S, R, S\}$.

However, before diving directly in algorithms, is important to note the characteristics of the Environment and of the Agent. Based on these we'll be able to perform much better choosing the search algorithm that is best for us.

3.1.1 Environment

Observability: how much can the agent know about the environment.

An environment is fully observable if the agent can see everything. Or partial observable: - part of the state is occulted and you simply can't

Deterministic: the effect of the actions are predictable If I move one queen from one place to the other, I can predict the effects, what is attacking what, etc... Instead of Deterministic, it can be stochastic, where the outcomes are functions of probability functions. Therefore, you can't be 100% sure of the outcome. Therefore, you can use probability to make choices...

Neither the environment nor the agent performance change while the agent is deliberating. –¿ Static.

In a dynamic environment, the agent performance can change with time.

Semi-dynamic means that the world is static, but the performance is changing. ex: turn game where the game doesn't change while you are thinking but the more you take, the less point you get.

Continuous or discrete

Sequential or episodic. Episodic means that one episode doesn't influence the next one. It's very related with causality. In episodic environments, there's no influence in consequent problems.

E.g. one game of chess is sequential, but different games are episodic.

The outcomes for all actions are given. –¿ Known environment.

In a case where the less time you spend thinking before answering,

Knowing these is very important because it allows us to best choose the shelf of methods from which we take our algorithms from. Some are best from some things and some can only be used in certain situations as well.

Internally to the agent, you have a way of representing of the world. (Internal representation)

Example when you don't have an internal representation: random vacuum cleaning robots don't know anything about the environment, they just rotate randomly when they see an object.

- Fully observable vs. partially observable
  (partially observable because of noise, inaccurate or faulty sensors, or hidden parts)
- Single agent vs. multiagent (cooperative vs. competitive)
  (in multiagent environments, communication may be a key issue)
- Deterministic vs. stochastic
  (an environment is uncertain if it is not fully observable or not deterministic)
  (nondeterministic environment is when actions have different possible outcomes but no probabilities associated)
- Episodic vs. sequential
- Static vs. dynamic
  (if the environment doesn't change with the time but agent's performance does, it is **semidynamic**)
- Discrete vs. continuous (applies to states, time, percepts, and actions)
- Know vs. unknown
  (depends on the agent's knowledge about how the world evolves – the "laws of physics" of the environment)

	observable	deterministic	episodic	static	discrete
Chess	Y	Y	N	Y	Y
Poker	N	N	N	Y	Y
Taxi	N	N	N	N	N
Image analysis	Y	Y	Y	Semi	N

3.1.2 Agent

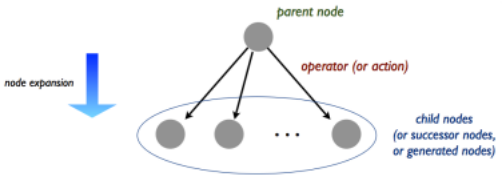
Model-based: typically, when you have partial observability.

Goal-based: when you aim for a goal. You have states and actions and can search for the goal.

Utility-based: are very similar to goal-based but are a bit more, not all goals are the same. There a preference between goals. Utility Theory handles how to translate preferences to numbers.

3.1.3 Search Problems

In essence, is necessary some **Search Terminology** to refer to certain things:



Successor function: given a node, returns the set of child nodes

Open list (or frontier or fringe): set of nodes not yet expanded

Closed (explored) list: set of nodes already expanded

Leaf node: a node without successors

State space can be:

- Tree-based – no repeated nodes in search
- Graph-based – directed cycle graph

General algorithm shape, starting with $open\_list = \{initial_node\}$, iterate over:

1. Select a node from the open_list;
2. Check if it's a goal node (in case it satisfies the goal test). If yes, return solution by backing up to the root;
3. If not, remote it from the open_list, expand it with the successor function and insert the nodes that come from there to the open_list.

Different selection criteria leads to a variety of search methods.

If it's tree based, then no nodes will be repeated and it's not necessary to have a list of the already visited nodes. In a graph however, is needed to have a list of these, or else a cycle is possible.

To evaluate the algorithm, many parameters may be enumerated:

Completeness: guarantee that a solution is found if there is one

Optimality: the solution found minimizes path cost over all possible solutions

Time complexity: how long does it take to find a solution (usually measured in terms of the number of nodes generated)

Space complexity: how much memory is needed to find a solution (usually measured in terms of the maximum number of nodes stored in memory)

Branching factor (b): maximum number of successors of any node

Depth (d) of the shallowest goal node

m = maximum length of any path in state space (may be infinite)

g(n): the cost of going from the root node to node n (path cost function)

Mentioning types of search strategies:

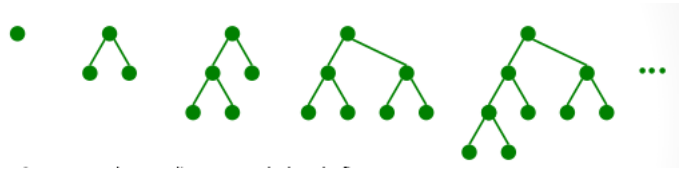
Uninformed (or blind) search: does not use additional (domain-dependent) information about states beyond that provided in the problem definition

Informed (or heuristic) search: uses problem-specific knowledge about the domain to "guide" the search towards more promising paths

3.1.4 Uniformed Search Strategies

A list of **Uniformed search strategies** we'll have a deeper look to:

- **Breadth-first Search** - Select earliest expanded node first - uses a FIFO queue (First In, First out). This leads to opening every node at the safe depth first before moving the deeper nodes.



Because order of depth is followed and every node checked, this search strategy is Complete and Optimal (if the path cost increases - or at least doesn't decreases - with depth). Being **d** the depth of the solution and **b** the branching factor(max number of successors of a node.) then in the worst case, the total number of nodes generated is: $1 + b + b^2 + b^3 + \cdots + b^d$

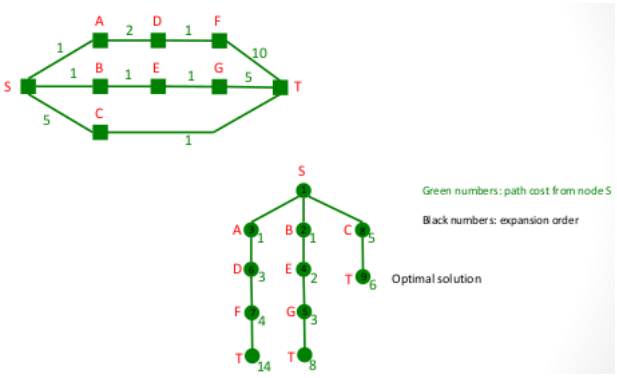
Time Complexity - O(b^d) Space Complexity - O(sum no of nodes) = O(b^d)

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Note that it makes a difference if you test the node before or after expanding it. If it's tested before, there's no need of expanding the node. If the test is made only after the expansion, the complexities grows to O(b^{d+1}).

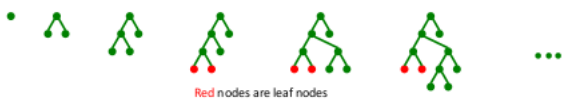
- **Uniform-cost Search** - expands the node that has the smallest cost from the root to it.



Note that each action generates one possible state from the state where the robot was previously.

This search strategy is only complete and optimal if the step costs are strictly positive. Else, it can give several steps that cost nothing to places far away from the solution.

- **Depth-first Search** - Exactly as the name suggests, goes until the deepest node first, and only then looks at the other nodes at the first level. Therefore, it can be very inefficient on a large or infinite tree. Open the last node added to the list (LIFO- Last In, First Out).



- **Backtrack Search** - a variation of depth-first, but expands one node at a time, only stores in memory that only node and the expansion is made by modifying the node, while backtrack is nothing more than undoing the modification..

  It's not complete nor it is optimal... but saves a lot of memory.

- **Depth-limited search** - another variation of depth-first where limiting the search tree to a depth L, is possible to contain the inefficiency. It's complete if the depth of the solution if smaller than L but still not optimal.

  Time complexity: O(b^L) Space complexity: O($b \times L$)

- **Iterative deepening depth-first search** - A variation of the previous one. In this one, the idea will be to run depth-limited search for an increasing L. Run for L=1, L=2, ...

This way, it is complete and it's optimal (if the path cost is a non-decreasing function of depth).

Time complexity: $O(b^d)$ Space complexity: $O(b \times d)$

Node expansion:

Breadth-first

$$1+b+b^2+b^3+\cdots+b^{d-1}+b^d$$

Iterative deepening

$$(d+1)1+(d)b+(d-1)b^2+(d-2)b^3+\cdots+(2)b^{d-1}+(1)b^d$$

Example: b = 10 and d = 5

breadth-first = 111 111

iterative deepening = 123 456 (+ 11%)

- **Bidirectional Search** - Search both from initial node and from the gold node. Note however that can only be used when a goal node is known and when the parent nodes can be computed given its child (through the sets of available actions). It's complete (if breath-first in both directions) and Optimal, if the step costs are equal.

  Time and space complexity: $O(b^{d/2})$

Example: $b = 10$ and $d = 5$

$b^d = 1\,111\,111$

$2\,b^{d/2} = 2\,222$

A summary of the above analysis:

Criterion	Breadth-first	Uniform-cost	Depth-first	Depth-limited	Iterative deepening	bidirectional
Complete?	✓	✓[1]	×	×	✓	✓[3]
Optimal?	✓[2]	✓	×	×	✓[2]	✓[2,3]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^L)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b.m)$	$O(b.L)$	$O(b.d)$	$O(b^{d/2})$

[1] for strictly positive step costs
[2] for path costs a non-decreasing function of depth
[3] for breadth-first in both directions

A **General Search Algorithm** can be formulated in the following way:

function General-search (***problem***, ***strategy***) **returns** a solution or failure
insert the root node into the open list
(the root node contains the initial state of problem)
loop do
if there are no candidate nodes for expansion **then return** failure
choose a node for expansion according to strategy (using strategy function)
if the node contains a goal state (using goal checking function) **then**
return the corresponding solution
else
for each operator in the list of operators (or successor function)
create a child node (for the new child state)
update child node path cost (using g-function)
add the resulting node to the open list [*unless… see graph search versions*]
end

domain (problem) independent

problem argument should include at least:

- initial state (using a specific state representation)
- successor function: new state = succ (current state, operator)
- path cost function (g-function)

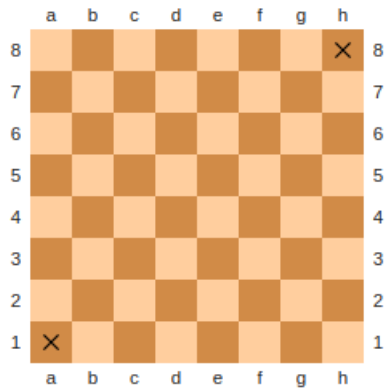
domain (problem) dependent

strategy argument (e.g., FIFO, LIFO, priority queue (by path cost), etc.)

algorithm selection

Finally, there are problem dependent and problem independent details. The search strategies are problem independent, but how we define the actions, states, ect… is problem dependent. And the way we see and think about the problem can be highly related to the way we represent it.

An incredibly well formulated example is the Mutilated Chess Board problem. If we take just the squares of the corners of a chess board, can we still fill the whole board with dominos (each domino takes 2 squares). It becomes slightly harder! If you keep removing squares it gets increasingly harder to figure out by head.



However, if you represent the chess board as the remaining black and white pieces and notice that a domino piece must always cover one black and one white squares, then by taking those two corners then 30 black and 32 white squares will be remaining making it impossible to fill with dominos. As a matter of fact, accordingly with Gomory's Theorem is also possible to say that if 2 square of opposite colours are removed then is always possible to fill the board with dominos! More in: Mutilated chessboard problem

The bottom line is that the representation (problem dependent details) matters.

With this is mind, consider the following example:

Initial state with: bedroom(3), living room(2), kitchen(3), hall(2), truck(0)

a) State: $< pos, n_{bedroom}, n_{living}, n_{kitchen}, n_{hall}, n_{truck} >$

b) Initial State: $< truck, 3, 2, 3, 2, 0 >$

c) m for move, p for push $< m_N, m_S, m_E, m_W, p_N, p_W, p_E, p_W >$. However, is only possible to push in directions where there are rooms (as for walking) and when there are boxes in the current room we are located in.

d) A goal condition could be $n_{truck} = 10$. Another could be the sum of all rooms to be 0. But the first one is more elegant and also seems more general... We don't want to throw boxes out of the window.

Sometimes, just finding the solution is enough. Some times, there's a best solution.

The heuristic (the only difference between informed and uninformed search) is a function of a state that gives us the appreciation we have for that state - how good that state is. If we want the best solution, we must have the heuristic function is key. If you just want a solution, that function can be much simpler or even nonexistent.

3.1.5 Informed Search Strategies

A problem-solving agent is a goal-based agent that acts on the environment, leading him to go through a series of states in order to achieve the desired goal

In this course, we only study the single-state problems

3.2 Supervised Learning

Supervised learning concerns the problems where the objective is to predict something based on previous data. The counterpart Unsupervised Learning tries to find patterns in unlabelled data.

More generally, the dataset for supervised learning problems consists on a feature vector **x** and a output vector **y**. Regarding unsupervised learning, everything is features / data.

3.2.1 Regression Problems - Least Squares

These are the two most commons types of problems. Probably every supervised learning problem can be *classified* as one of these.

Regression is when the output should be continuous, classification when the output should be in discrete classes.

About the first, a measure to minimize is the difference between our prediction to the value we want to achieve. The Sum of the Square Errors (SSE) is very standard cost function to minimize.

The function that is required to minimize is loss/cost/risk function. Nomenclature wise is a problem... Therefore, the following letters/terms can be will be used interchangibly: L (Loss Function) or J (more used when the weights or coefficients are θ) or R (Risk Function):

$$R = \frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)} \right)^2$$

Where $\hat{y}$ is our prediction or estimation of the true value of y and **m** is the number of training samples we have. Therefore $y^(i)$ constitutes the outcome of sample i.

One prediction can be made with:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$$

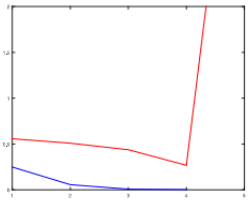
Where **n** is the number of features we are using. Note that features and data points are different things. We can have data regarding only one measure but take the square and the cube of the measure multiplied to different coefficients in order to try to better estimate the function.

β's are usually used in regressions while in neural networks letters like θ or w are more common.

One intuition that is important to have is that **the more we increase the features, the more likely it is that we end up overfitting our training set and loosing generalization capabilities for the actual test data** . This is why it is important to divide all the available data in sets, the model will be trained to guess the training samples, not samples that it never saw before.

- One set for training and one set for testing the prediction capabilities;
- One training set, one validation set and one test set.

The last option is meant to validate the model. The model is trained with the training set, then some parameters are tuned with the validation set, namely the number of polynomial features, regularization term and step size related parameters (like momentum or adaptive step size), and the actual performance is testing in the test set. This way, we avoid optimistic measures of performance by not testing in data used for training.



Average loss in the training set (blue) and in an independent set (red), as a function of polynomial degree.

The result wouldn't be very different if done with the number of iterations or number of features. In particular, it is called doing an "early stop" when the iteration that minimises the loss in the test set is early in the minimization process. It is useful when the model starts overfitting the data.

3.2.2 So, how do we calculate the coefficientes

After having the coefficients, given any other set of data points we can already give predictions on the output.

Note that we are trying to minimize the cost/risk/loss function. The actual cost function would have to be some sort of prediction because it's impossible to know exactly how much the actual outcome will be, despite knowing exactly what the outcome of the model will be to a certain feature vector. As opposed to the empirical risk function where the training outcome and the training predicted result are used, therefore being able to calculate the difference between each estimation and the supposed outcome. To compute the real risk, the expected value of the SSE is necessary. Also, there are continuous results so an integral is required:

$$R = E[y - \hat{y}] = \int_{-\infty}^{+\infty} L(y, x)\phi(y)dxdy$$

Because a potential function to give us a measure on how frequent certain values are is not known, the only way is to approximate the actual error empirically, using the model with some test data. This will degenerate in the actual cost function presented before. L here is meant to denote the loss of one sample which is nothing more than the squared error.

One thing that won't happen in all problems is having an analytical and optimum solution for them. Actually, minimizing the SSE is a kind of problem is called the **Least Squares** . This kind of problem is very usually used in optimisation and often a closed solution is possible.

In this case, since we are searching for the function's minimum, the functions partial derivatives need to be zero in order to have a critical point (maximum, minimum or saddle point).

In this case,

$$\frac{\delta R}{\delta \beta_0} = -2\sum_{i=1}^{n}(y^{(i)} - \beta_0 - \beta_1 x^{(i)}) = 0$$
$$\frac{\delta R}{\delta \beta_1} = -2\sum_{i=1}^{n}(y^{(i)} - \beta_0 - \beta_1 x^{(i)})x^{(i)} = 0$$

Is possible to simplify further these equations, putting the β's in evidence and separating sums, arriving at:

$$\begin{bmatrix} \sum_{i=1}^{n} 1 & \sum_{i=1}^{n} x^{(i)} \\ \sum_{i=1}^{n} x^{(i)} & \sum_{i=1}^{n} x^{(i)2} \end{bmatrix} \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y^{(i)} \\ \sum_{i=1}^{n} y^{(i)} x^{(i)} \end{bmatrix}$$

Is possible to invert these equations and get the expressions for the coefficients. However there's one important factor to have in mind. Are we aiming at a minimum, maximum or something different like a saddle point? The Hessian Matrix will tell us.

3.2.3 Extrema Conditions and Hessian Matrix

Videos: 87 - Warm up to the second partial derivative test to... 89 - Second partial derivative test intuition

$$H = \begin{bmatrix} \frac{\partial^2 SSE}{\partial \beta_0^2} & \frac{\partial^2 SSE}{\partial \beta_0 \partial \beta_1} \\ \frac{\partial^2 SSE}{\partial \beta_1 \partial \beta_0} & \frac{\partial^2 SSE}{\partial \beta_1^2} \end{bmatrix} = 2 \begin{bmatrix} \sum_{i=1}^{n} 1 & \sum_{i=1}^{n} x^{(i)} \\ \sum_{i=1}^{n} x^{(i)} & \sum_{i=1}^{n} x^{(i)2} \end{bmatrix}$$

In one dimension, to find an extrema is necessary to equalise the first derivative to zero, and the second derivative must be positive - in case of a minimum - or negative - in case of a maximum. If the second derivative is zero at the critical point, then there's an inflexion point. A similar analysis must be done in $(n+1)$-D where n is the number of features used in the regression.

Guaranteeing that the first derivative is zero, which in N dimensions is correspondent to guaranteeing that the gradient is zero in that point, is the first step. Setting $\nabla f = 0$ means that, in that point, the function is not increasing or decreasing in any of the N directions. So the first derivative makes sense.

However, when we go to the second derivative, the meanings get a bit more complicated.

In 2D, if the second derivative was 0, it was certainly a saddle point, if it was > 0 or < 0 it was, respectively, a local minimum or maximum.

The conditions we should impose in 3D is to have a positive (for finding a minimum) or negative (for finding a maximum) definite Hessian Matrix. While it is possible to attribute a meaning to second derivatives in order to just one variable, being nothing more than the concavity in those 2 directions, why do the cross derivatives play a role as well? And, why do they mean really?

Well, first the explanation on why it is needed: there are functions that across multiple dimensions still show that it is an extrema but then there's an inflexion along directions that are not along the axis. So, checking the axis is not enough. Why checking the cross partial derivatives makes it enough?

The Second Derivative Test

$$f_{xx}(x_o, y_o) f_{yy}(x_o, y_o) - f_{xy}(x_o, y_o)^2 \gtrless 0$$

If it is greater than 0, we have a maximum or a minimum and have to check the value of f_{xx} or f_{yy} to be sure. If it is less than 0, we have a saddle point. If it equals 0, then we don't know if it is a saddle point, but it is not a min or max therefore, at least for now, we certainly don't care.

Cross or Mixed partial derivatives can be switched? Yes if the function is C^2. (Boring to prove theorem called: Schwarz' Theorem)

Therefore, we just need to compute one of the cross derivatives.

Also this works because the second derivative test is nothing more than the determinant of the Hessian matrix. The determinant is the product of every eigenvalue of that matrix, therefore it can only be positive if they are both positive or both negative, in which cases there is, respectively, a minimum or a maximum.

But why do eigenvalues tell us this? Because they tell us how the eigenvectors are scaled! And the eigenvectors of such matrix will be the greatest and the least curvatures. Therefore, they either have the same signal / are scaled the same amount, or

Some other links that helped with this:

- Differential Geometry
- Criterior for critical points - Maximum, Minimum or Saddle?
- David Butler - Facts about Eigenvalues

The two main properties of eigenvalues that allow us to quickly calculate them from the Hessian matrix (specially if it is 2x2) are:

$$\mathrm{tr}(A) = \sum_{i=1}^{n} \lambda_i$$

$$\det(A) = \prod_{i=1}^{n} \lambda_i$$

Because there are only 2 variables, there can't be very big changes across more than 2 main directions, so it is possible to quantify the main directions which will be the eigenvectors directions.

The eigenvalues of the Hessian Matrix are also called principal curvatures and the eigenvectors the principal directions.

3.2.4 Analytical Expression for the Coefficientes

From the equation presented in the end of 3.2.2, we can re-write the SSE and the normal equations in the following way.

cost function: $SSE(\beta) = \|y - X\beta\|^2$

normal equations: $(X^T X)\hat{\beta} = X^T y$

And arrive at the analytical expression through the simple inversion of the normal equations. Another way of reaching the analytical expression is deriving the cost function.

parameter estimates: $\hat{\beta} = (X^T X)^{-1} X^T y$

Cost function

$$\begin{aligned} SSE &= \|y - X\beta\|^2 = (y - X\beta)^T (y - X\beta) \\ &= y^T y - 2y^T X\beta + \beta^T X^T X\beta. \end{aligned}$$

Computing the gradient and making it equal to zero

$$\nabla_\beta SSE = -2X^T y + 2X^T X\beta = 0,$$

Note however that:

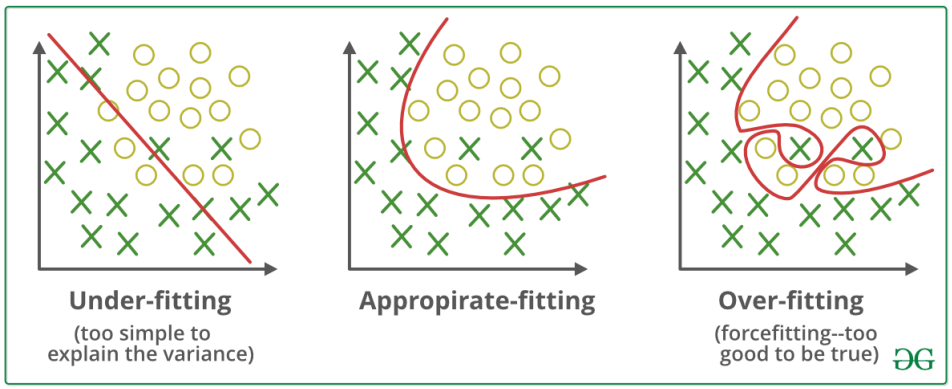
The inverse of matrix $X^T X$ may not exist due to two main reasons:

- small amount of data *e.g.*, number of data points smaller than the number of features.
- redundant features (linearly dependent) *e.g.*, duplicated features.

A final remark on multiple outputs: in case our feature vector serves to estimate more than one output, we can simply use it separately for each output!

3.2.5 Regularization

This is the method of taking importance away from the minimization of the errors between the training set supposed outcomes and the actual model outcomes for those samples. If we don't take importance away, the model may become too good at predicting training examples and may forget that it should predict a tendency and generalize well for the test data.



Performing a regularization consists on nothing more than adding a new parameter to the cost function, in order to shift away the focus of minimizing the SSE.

There are generally two terms that can be added. One with the **norm of the coefficients squared** and the other is the with the module of the coefficients squared.

For the norm squared, if the regularization is applied to a regression - **which is not a necessity since regularization can even be applied to Neural Networks** - it's called Ridge Regression:

$$\hat{\beta}_{\text{ridge}} = \arg\min_{\beta} \|y - X\beta\|^2 + \lambda\|\beta\|^2$$

$$\hat{\beta}_{\text{ridge}} = (X^T X + \lambda I)^{-1} X^T y$$

Two key things to note:

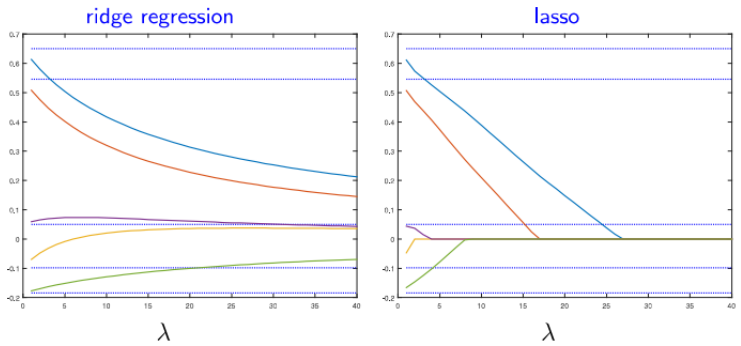
- Note that β_0 is usually not included as the data should be normalised already for much better results. Normalised data means data that has zero mean in every feature and outcome.
- If $\left(X^T X\right)$ is singular, the least squares estimate is not unique. Regularization will help finding an estimate even then because $\left(X^T X + \lambda I\right)$ is always non-singular.

For the simple norm of the coefficients, when applied to a regression problem it is called the Lasso Regression:

$$\beta_{\text{lasso}} = \arg\min_{\beta} \|y - X\beta\|_2^2 + \lambda\|\beta\|_1$$

$$||\beta||_1 = \sum_{j=1}^{n} |\beta_j|$$

The key difference between the two is that Ridge aims to minimize the norm of all of them while Lasso aims to minimize the sum of module of each of them. Therefore, Ridge it is much likely to pull closer to zero the biggest ones as those are the ones that matter the most for the Euclidean norm, while Lasso will try to pull each of them to 0, there's a direct dependency between a coefficients and the cost function. This is also why the Lasso Regression is called to do feature selection: because if the SSE doesn't depend on the coefficients, the regularization term with the sum of the norms will put that coefficients to zero very quickly.



Again, recall that the data should be centered - have zero mean - and that after calculating the model we need to de-centre it to obtain the real predictions!

How should we proceed if the training data $\mathcal{T} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ are not centered?

1. pre-processing: $x'^{(i)} = x^{(i)} - \bar{x}$, $y'^{(i)} = y^{(i)} - \bar{y}$ ($\bar{x}, \bar{y}$ average values computed in the training set);
2. estimate linear model without intercept: estimate model $y' = x'^T \beta'$, with $\beta' \in \mathbb{R}^p$, using the pre-processed data $\mathcal{T} = \{(x'^{(1)}, y'^{(1)}), \ldots, (x'^{(n)}, y'^{(n)})\}$ and regularization;
3. invert pre-processing: $\hat{\beta} = [\hat{\beta}_0 \ \hat{\beta}'^T]^T$ where $\hat{\beta}_0 = \bar{y} - \bar{x}^T \hat{\beta}'$;

3.2.6 Optimization problems - Gradient Descent and Newton's Method

The gradient descent is probably the most know method to approximate a functions minimum. By changing the direction of the step we have the gradient ascent.

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

The above expression works because the gradient points the direction of the maximum growth of the function. Therefore, taking a step in the opposite direction will lead to the minimum.

Momentum and Adaptive StepSize

Momentum $0 \leq \alpha \leq 1$:

$$\begin{aligned} \Delta \mathbf{x}^{(n+1)} &= \alpha \Delta \mathbf{x}^{(n)} - \eta \nabla f[\mathbf{x}^{(n)}] \\ \mathbf{x}^{(n+1)} &= \mathbf{x}^{(n)} + \Delta \mathbf{x}^{(n+1)} \end{aligned}$$

or, alternatively, by

$$\begin{aligned} \Delta \mathbf{x}^{(n+1)} &= \alpha \Delta \mathbf{x}^{(n)} - (1-\alpha)\eta \nabla f[\mathbf{x}^{(n)}] \\ \mathbf{x}^{(n+1)} &= \mathbf{x}^{(n)} + \Delta \mathbf{x}^{(n+1)} \end{aligned}.$$

We'll use this second version.

Note that we want to pick the $\mathbf{x}$ that brings the function to a minimum. Therefore, the "exponent" will simply refer to the iteration number, not the sample like in the previous section. If α is closer to 1 the memory of the previous increment is more taken into account, meaning that the increment will change only slightly. The closer the parameter gets to 0, the closer we get to the normal situation. This is called the momentum term because it gives the convergence some inercia, the behaviour of momentum. By changing the increment slowly, it may converge faster and have less abrupt changes.

Adaptive Step size, with typical values: $u = 1.2, d = 0.8$:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta_i^{(t)} \frac{\partial J}{\partial \theta_i}\left(\theta^{(t)}\right).$$

Step size update

$$\eta_i^{(t)} = \begin{cases} u\eta_i^{(t-1)} & \text{if } \frac{\partial J}{\partial \theta_i}\left(\theta^{(t)}\right) \cdot \frac{\partial J}{\partial \theta_i}\left(\theta^{(t-1)}\right) > 0 \\ d\eta_i^{(t-1)} & \text{otherwise} \end{cases}.$$

Let f be the function where the objective minimum lies, the Divergence criterium is given by:

$$f(x^{(n+1)}) > f(x^{(n)})$$

Where the threshold is found in the equality.

Newton's method

Given by:

$$\theta^{(t+1)} = \theta^{(t)} - \left[H(\theta^{(t)})\right]^{-1} \nabla J_\theta(\theta^{(t)})$$

Where the gradient and the Hessian Matrix are given by:

$$\nabla_\theta J = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} \qquad H = \begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1^2} & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_d} \\ \frac{\partial^2 J}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 J}{\partial \theta_2^2} & \cdots & \frac{\partial^2 J}{\partial \theta_2 \partial \theta_d} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial^2 J}{\partial \theta_d \partial \theta_1} & \cdots & \frac{\partial^2 J}{\partial \theta_2 \partial \theta_2} & \frac{\partial^2 J}{\partial \theta_d^2} \end{bmatrix}$$

The Newton's method converges insanely fast! But requires the inversion of the Hessian matrix which can be a serious problem...

3.2.7 Newton's Method - Intuition and Demonstration

This is a very very good proof!

3.2.8 How to optimise hyperparameters

There will always be parameters to optimise in order to obtain the best model possible. It was said before that 3 sets should be selected: training, validation and testing. And that it was in the validation set that all hyperparameter tunning should be done. The simplest way is calculate the model with all combinations of the parameters possible and see which one performs better in the validation set.

There can be one other problem: little data. If the data is too little, dividing it into sets can start to give biased results to the accuracy.

One way of calculating the accuracy with more ... accuracy ... is splitting the data in k folds and rotate:

Data: k folds $\mathcal{T}_k$.
for k=1, ..., K **do**
  $f = \text{train}(\mathcal{T} \setminus \mathcal{T}_k)$;
  $P_k = \text{perform}(f, \mathcal{T}_k)$;
end
$P = \bar{P}_k$

Finally, if both things need to be done at the same time, then: 1- k folds need to be made. 2- for each fold, all values of the hyperparameters need to be used for training and tested in the test set. Note: it will be used for training T except (T_i and T_j) that are, respectively, the test set and the validation set. So the hyperparameters testing will be done with T_i. When the all combinations are done, the hyperparameters are selected for the bet one and the actual model is trained with T except T_i depending on the fold considered. 3- use the performances of each fold to get the best average of performance.

(Note that this is not very used...)

Data: k folds $\mathcal{T}_k$.
for $i = 1, \ldots, K$ **do**
  for *all values of* ξ **do**
    for $j \neq i$ **do**
      $f = \text{train}(\mathcal{T} \setminus (\mathcal{T}_i \cup \mathcal{T}_j), \xi)$;
      $P(\xi)_j = \text{perform}(f, \mathcal{T}_j)$;
    end
    $P(\xi) = P(\xi)_j$
  end
  $\hat{\xi}_i = \arg\min_\xi P(\xi)$;
  $f = \text{train}(\mathcal{T} \setminus \mathcal{T}_i, \hat{\xi}_i)$;
  $P_i = \text{perform}(f, \mathcal{T}_i)$;
end
$P = \bar{P}_i$

3.2.9 Neural Networks

Some history

Data: k folds $\mathcal{T}_k$.
for $k=1, \ldots, K$ **do**
  $f = \text{train}(\mathcal{T} \setminus \mathcal{T}_k)$;
  $P_k = \text{perform}(f, \mathcal{T}_k)$;
end
$P = \bar{P}_k$

Pros
It can be proved that the Rosenblatt algorithm solves any binary problem in a finite number of iterations, provided the training data can be separated by a hyperplane in feature space.

However, there's a big problem with only being able to distinguish data that can be separate with an hyperplane: very often the data doesn't behave that way.

How should we choose the number of layers?
Cybenko (1989) proved that a multilayer perceptron with 1 hidden layer is an universal approximator of any continuous function defined on a compact subset of $\mathbb{R}^p$. This is a useful theorem but it does not explain how many units are needed nor how should the weights be chosen.

- Common practice shows that it is often better to use more layers since the network can synthesize a wider variety of nonlinear functions with less units.
- It also shows that deeper networks (with more layers) are more difficult to train.

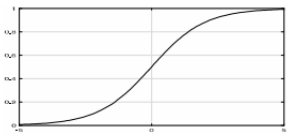
One can make a more complex analysis of the situation...

Activation Functions and Architecture

Some activation functions:

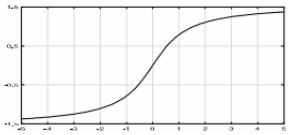
sigmoid: logistic function

$$g(s) = \frac{1}{1 + e^{-s}}$$



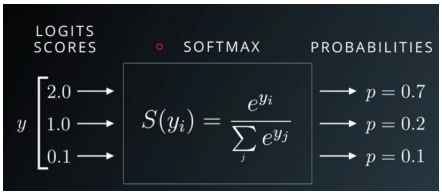
sigmoid: arctangent function

$$g(s) = \arctan s$$



Two other are Rectifier Linear Unit and the softmax:

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ ax & \text{otherwise.} \end{cases}$$



In the all layers with the exception of the last one, ReLu or the logistic function work very well. The last layer must have a function that returns results between 0 and 1, therefore only SoftMax and sigmoid functions like the logistic function would work properly.

Training methods:

The gradient vector includes the contribution of all the training patterns. The weight update using all the training patterns in each iteration is called the batch mode.

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{R}}{\partial w_{ij}} = -\frac{\eta}{n} \sum_k \frac{\partial L(y^{(k)}, \hat{y}^{(k)})}{\partial w_{ij}} = -\frac{\eta}{n} \sum_k \frac{\partial L^k}{\partial w_{ij}} .$$

Another alternative consists of using one training pattern k, only, and updating the weights with that information. This is called the on-line mode

$$\Delta w_{ij} = -\eta \frac{\partial L^k}{\partial w_{ij}} .$$

A third hypothesis consists of updating the NN weights using a small subset of training patterns. This is known as mini-batch mode.

One very interesting fact is that the image features are not selected by anyone. The network itself crafts its features throught backpropagating the changes required to get the images right. This is true for all Neural Networks.

Also, in image recognition for instance, but in deep neural networks in general, the last layers usually are fully connected.

3.2.10 Neural Networks - BackPropagation

Some of the most useful websites to check while trying to demonstrate the backpropagation algorithm:

- all backpropagation derivatives
- Error (deltas) derivation for backpropagation in neural networks
- 3Blue1Brown Neural Networks - Specially the last one, on backpropagation.

The fastest way is the following:

Note that multiple training modes are possible. The normal one is on-line, where the increment to the weight is nothing more than the step multiplied by the respective partial derivative - Equation (1). Then one can do the batch-mode where all the samples are considered for the weights update - Equation (2). Finally, there's the mini-batch mode that doesn't use all the training samples.

$$w_{ij}^{(l)} = w_{ij}^{(l)} + \triangle w_{ij}^{(l)}$$

$$\triangle w_{ij}^{(l)} = -\eta \frac{\delta C^k}{\delta w_{ij}^{(l)}} \tag{1}$$

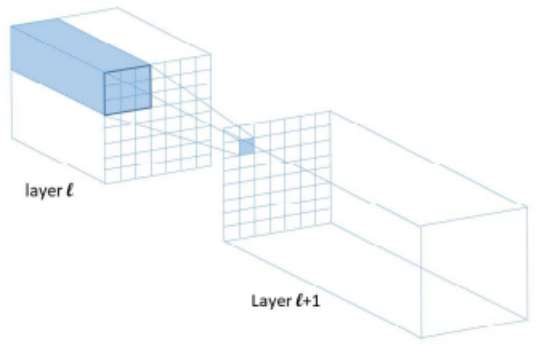
$$\triangle w_{ij}^{(l)} = -\frac{\eta}{m} \sum_{k=1}^{m} \frac{\delta C^k}{\delta w_{ij}^{(l)}} \tag{2}$$

3.2.11 Neural Networks - Convolutional

Convolutional Neural Networks are extremely useful for applications like image recognition. They take a width x height x image depth 3D array and convolve it with a kernel, generating an 2D array.

Because many elements in an image are translation invariant, considering patches of the image is one of the best ways of acquiring features.

Convolutional neural networks allow us to use less inputs to our neural networks. We map a region of a picture to just one pixel/input. We tend to use many different kernels - a set of weights to multiply to each of pixel in the set we chose - to perform this step.



At the end of the convolution, an activation function (like ReLu) is applied.

Note that this convolutional layer can be applied to a 3D array to reduce it to 2D. Moreover, many different kernels can be convolved with the section of the 3D array creating several layers. If there are 20 kernels, we'll have 20 2D arrays, therefore a new 3D array that is all the pixels in the several vicinities, weighted with different kernels.

3D input: $z_{ijk}^{\ell-1}$ $\ell - 1$ - number of input layer

3D kernel: h_{ijk}^{ℓ}

2D output:

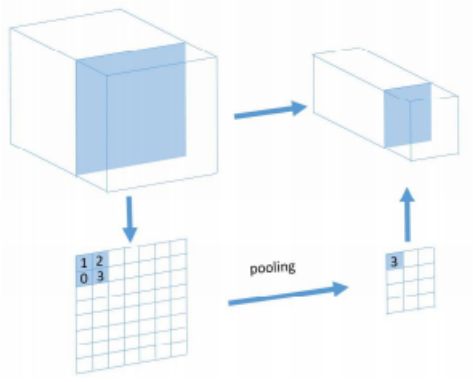
$$s_{ij}^{\ell} = \sum_{p}\sum_{q}\sum_{r} h_{pqr}^{\ell} z_{i+p,j+q,0+r}^{\ell-1}$$

$$z_{ij}^{\ell} = g(s_{ij}^{\ell})$$

Each filter produces a 2D output known as a feature map. Stacking the feature maps produced by multiple filters leads to a 3D array.

Pooling

2D to 2D but with smaller dimensions. Downsize the feature array by choosing what we consider to be the most important values. For instance, for each 4x4 square of pixels, we choose the maximum of them (the highest value in grey scale, for instance). Therefore, the end result will have 16x less pixels/inputs.



Overall, the tendency is the kernel to be smaller, but the network to have many many layers.

3.2.12 Kernels

Why kernels are important? They facilitate a lot the mapping of features to higher dimensions!

Why kernels?

3.3 Unsupervised Learning

3.3.1 Lagrange Multipliers

The Lagrange Multipliers is a mathematical optimization method, allows the finding of local maxima or minima of a function, subject to equality constraints.

Perfect Explanation why it works:

Quora - Intuition on Langrange Multipliers

And a video showing exactly how it's done:

Khan Academy - Lagrange Multipliers Example

3.4 Reinforcement Learning and Decision Making

An agent to make the wisest decision in its situation has to have knowledge on what state he is in, how the environment will evolve, how it will be like if he performs a certain action (taking into account stochastic environments) and a utility function to know how much that state contributes to its happiness.