Informatics Engineering

Intelligent Systems

Secta Final work

Ducha Cásedas, Diego
Neva Carod, Marcos
Pinedo Olivan, Alejandro

3º Informatics Engineering

23/01/2020

First Part (Practical part)

Reinforcement learning:

How Does Reinforcement Learning Work?

Imagine an agent is trying to pick up a pen, and it fails. It tries again, fails. After repeating this process 1000 times, it finally succeeds. The agent has now learned how to pick up a pen. This is Reinforcement Learning in a nutshell, it's a lot like how living creatures learn.

There are 3 key terms in Reinforcement Learning.

State: Describes the current situation.

Action: What the agent can do in a situation.

Reward: Feedback for whether a particular action in each state was good or bad.

What is q-learning?

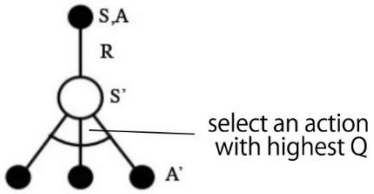
Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

Q-learning

Q-learning learns the action-value function Q(s, a): how good to take an action at a particular state. For example, for the board position below, how good to move the pawn two steps forward. Literally, we assign a scalar value over the benefit of making such a move.

In Q-learning, we build a memory table Q[s, a] to store Q-values for all possible combinations of s and a. If you are a chess player, it is the cheat sheet for the best move. In the example above, we may realize that moving the pawn 2 steps ahead has the highest Q values over all others. (The memory consumption will be too high for the chess game. But let's stick with this approach a little bit longer.)

Technical speaking, we sample an action from the current state. We find out the reward R (if any) and the new states (the new board position). From the memory table, we determine the next action a' to take which has the maximum Q(s', a').



In a Pacman, we score points (rewards) by eating white pieces trying to avoid also enemies.
We can take a single move a and see what reward R we can get. This creates a one-step look ahead. R + Q(s', a') becomes the target that we want Q(s, a) to be. For example, say all Q values are equal to one now. If we move Pacman to the right and score 2 points, we want to move Q(s, a) closer to 3 (i.e. 2 + 1).

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

As we keep playing, we maintain a running average for Q. The values will get better and with some tricks, the Q values will converge.

Q-learning algorithm

The following is the algorithm to fit Q with the sampled rewards. If γ (discount factor) is smaller than one, there is a good chance that Q converges.

Algorithm:
- Start with $Q_0(s, a)$ for all s, a.
- Get initial state s
- For k = 1, 2, ... till convergence
  - Sample action a, get next state s'
  - If s' is terminal:
    - $\text{target} = R(s, a, s')$
    - Sample new initial state s'
  - else:
    - $\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$
  - $Q_{k+1}(s, a) \leftarrow (1 - \alpha) Q_k(s, a) + \alpha \, [\text{target}]$
  - $s \leftarrow s'$

However, if the combinations of states and actions are too large, the memory and the computation requirement for Q will be too high. To address that, we switch to a deep network Q (DQN) to approximate Q(s, a). The learning algorithm is called Deep Q-learning. With the new approach, we generalize the approximation of the Q-value function rather than remembering the solutions.

Conclusions

Q learning is a powerful technique believed by many researchers as the most promising lead towards artificial intelligence. RL can solve many toy problems, but the Q-Table is unable to scale to more complex real-world problems. The solution is to learn the Q-Table using a deep neural network. However, training deep neural networks on RL is highly unstable due to sample correlation and non-stationarity of the target Q-Network. So, this algorithm although sounds great at first sight still has some open problems to be solved.

Double Q-Learning (DDQN)

In DQN, the target Q-Network selects and evaluates every action resulting in an overestimation of Q value. To resolve this issue, DDQN proposes to use the Q-Network to choose the action and use the target Q-Network to evaluate the action.

In DQN the estimate of the Q value is:

$$Q_{\max} = \begin{cases} r_{j+1} & if\ episode\ terminates\ at\ j+1 \\ r_{j+1} + \gamma \max_{a_{j+1}} Q_{target}\left(s_{j+1}, a_{j+1}; \theta^{-}\right) & otherwise \end{cases}$$

Improvements to DQN

Many improvements are made to DQN. In this section, we will show some methods that demonstrate significant improvements.

Double DQN

In Q-learning, we use the following formula for the target value for Q.

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)\,.$$

However, the max operation creates a positive bias towards the Q estimations.

Consider the Q estimates are

$Q(s, a_1) = 1.05, Q(s, a_2) = 0.95$ at time t_0,

$Q(s, a_1) = 0.95, Q(s, a_2) = 1.05$ at time t_1.

$max_a Q(s, a) = 1.05$ for t_0 and t_1.

$Q(s, a_1) = 1.00, Q(s, a_2) = 1.00$ at time t_{1000}

From the example above, we see max Q(s, a) is higher than the converged value 1.0. i.e. the max operation overestimates Q. By theory and experiments, DQN performance improves if we use the online network θ to greedy select the action and the target network θ- to estimate the Q value.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\ D}\left(r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta_i^-) - Q(s, a; \theta_i)\right)^2$$

Deep Reinforcement Learning

Deep reinforcement learning is about taking the best actions from what we see and hear. Unfortunately, reinforcement learning RL has a high barrier in learning the concepts.

In most AI topics, we create mathematical frameworks to tackle problems. Q-learning is unfortunately not very stable with deep learning.

DQN is similar for Q-learning, but using a deep network to approximate Q. We use supervised learning to fit the Q-value function. We want to duplicate the success of supervised learning but RL is different. In deep learning, we randomize the input samples, so the input class is quite balanced and pretty stable across training batches. In RL, we search better as we explore more. So, the input space and actions we searched are constantly changing. In addition, as we know better, we update the target value of Q. That is bad news. Both the input and output are under frequent changes.

$$\underbrace{label}_{\text{changing}} = Q_\phi(\underbrace{\mathbf{s}_i}_{\text{changing}}, \mathbf{a}_i)$$

This makes it very hard to learn the Q-value approximator. DQN introduces experience replay and target network to slow down the changes so we can learn Q gradually. Experience replay stores the last millions of state-action-rewards in a replay buffer. We train Q with batches of random samples from this buffer. Therefore, the training samples are randomized and behave closer to the supervised learning in Deep Learning.

In addition, we have two networks for storing the values of Q. One is constantly updated while the second one, the target network, is synchronized from the first network once a while. We use the target network to retrieve the Q value such that the changes for the target value are less volatile. Here is the objective for those interested. D is the replay buffer and θ- is the target network.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

DQN allows us to use value learning to solve RL methods in a more stable training environment.

Q LEARNING

ADVANTAGES:

- Can be used to solve very complex problems that cannot be solved by conventional techniques
- Learning model is very similar to the learning of human beings
- The model can correct errors occurred during the training process
- Robots can implement q learning algorithms to learn how to walk.
- Can be useful when the only way to collect information about the environment is to interact with it.
- Maintain a balance between exploration and exploitation.
- Can create the perfect model to solve a particular problem.
- In the absence of a training dataset, it is bound to learn from it is experience.

DISADVANTAGES:

- Q learning as a framework is wrong in many different ways, but it is precisely this quality that makes it useful.
- Too much Q learning can lead to an overload of states which can diminish the results
- Needs a lot of data and a lot of computation.
- Assumes the world is Markovian, which it is not.
- Curse of real-world samples. The robot hardware is usually very expensive, suffers from wear and tear, and requires careful maintenance. Repairing a robot system costs a lot.
- We can use a combination of Q learning with other techniques rather than leaving it all together. One popular combination is Q learning with Deep Q Learning.

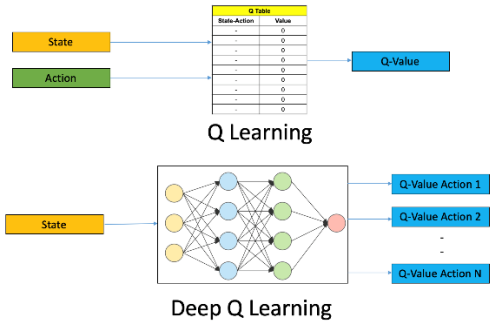
DEEP Q LEARNING

ADVANTAGES:

- Simple yet quite powerful algorithm to create a cheat sheet for our agent.
- Helps the agent figure out exactly which action to perform.
- Non-stationary or unstable target
- Experience Replay
- Simplify the more standard actor-critic style algorithms
- Preserves the benefits of nonlinear value function approximation.

DISADVANTAGES:

- Overestimates action values.
- Anything that requires reasoning like programming or applying the scientific method is out of reach for deep Q learning models.
- Does not have any understanding of their input, at least not in any human sense.
- Limited in what it can represent
- Cannot be expressed as continuous geometric morphing of data manifold



Q-Learning vs Deep Q-Learning 1

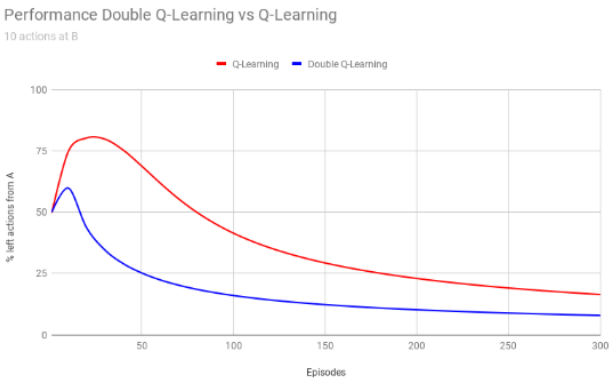
DOUBLE Q LEARNING

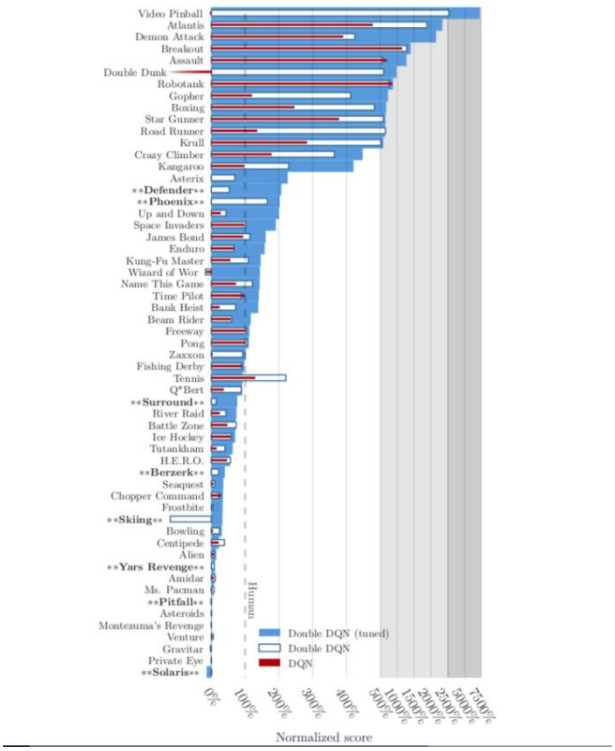
ADVANTAGES:

- Solves the problem of large overestimations of action value.
- Reduces the excessive variance for selecting the max estimated Q-value.
- Results may even slightly underestimate the Q-value.
- Reduces the estimation error.
- Performs well on a variety of MDP problems.

DISADVANTAGES:

- Same disadvantages of other Q-Learning methods but it reduce the time to train our programs.





Double Q learning vs Deep Q Learning

Secta PacMan Final Project

This practice aim is to create different algorithms in order to learn how teach and ai to play pacman, trying to reach the maximum scores available.
We have implemented several methods considering the explanations of above, by the way we have had several issues with this project that we will comment later.

First attempt to Reinforcement learning:

The website where we have found the archives and we have been following the tutorial of Reinforcement Learning is the following: https://www.cse.huji.ac.il/~ai/reinforcement/reinforcement.html

We have modified the following .py archives:

valueIterationAgents.py (A value iteration agent for solving known MDPs.)
qlearningAgents.py (Q-learning agents for Gridworld, Crawler and Pac-Man)

To make Pacman Learn in the Pacman original grid game you should run the following .py with the commands below:

python3 pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l originalClassic

Second attempt to Reinforcement learning:

We tried to train the algorithm, but we found some mistakes and errors during our training.

Second attempt has 4 different algorithms:

The first one has reach almost 1.600.000 steps training and we found that it was bugged.

The second one was a fixed algorithm with the same bases as the previous one, but with less batch size and less steps.

The third one was an implementation of the algorithm using the ram of our Pc, instead of using the GPU to train our models.

The fourth one was an implementation of the vanilla dqn method, that was trained with 10m of steps, and we obtain the best scores.

Aditional features to our Reinforcement Learning work:

Apart of the second option we also implemented two other reinforcement learning methods: Deep Q-Learning and Double Q-Learning.
The deep Q-Learning is implemented and explained on the folder: SectaPacmanAditionalFeatures with our working code and explanation.
The double Q-Learning is not implemented yet because of some errors at the implementation of the code, as the code is not implemented for the Pacman.
Also, we gathered all the colab implementations in the SectaPacmanColab folder. There is an environment to make Pacman OpenAI work.

GitHub link to see the whole work done: https://github.com/Pinedo-cell/SectaFinalProjectPacman

Second Part

Questions to resolve:

Describe the TABU, GENETIC and A Search algorithm.*

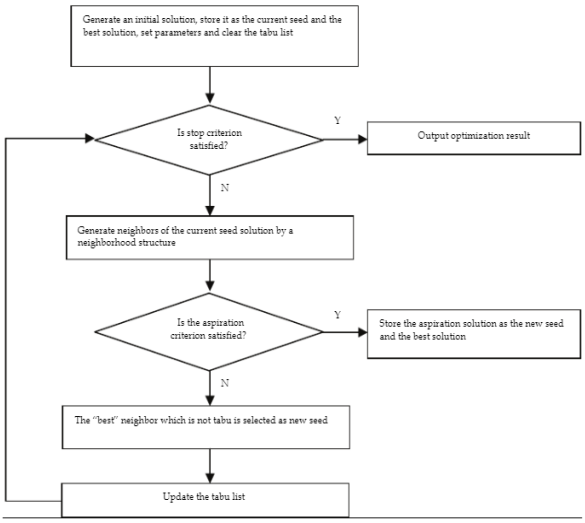
In which category do you classify it. Indicates advantages and disadvantages with the algorithms that have been studied in the classes.

Which advantages and disadvantages do you discover in this algorithm to solve the traveling salesman problem?

Tabu Search Algorithm

- Taxonomy: Tabu Search is a Global Optimization algorithm and a Metaheuristic or Meta-strategy for controlling an embedded heuristic technique.

- Strategy: The objective for the Tabu Search algorithm is to constrain an embedded heuristic from returning to recently visited areas of the search space, referred to as cycling. The strategy of the approach is to maintain a short-term memory of the specific changes of recent moves within the search space and preventing future moves from undoing those changes.



- Procedure: Algorithm (below) provides a pseudocode listing of the Tabu Search algorithm for minimizing a cost function. The listing shows the simple Tabu Search algorithm with short term memory, without intermediate and long-term memory management.

```
Input: TabuList_size
Output: S_best
S_best ← ConstructInitialSolution()
TabuList ← ∅
While (¬ StopCondition())
  CandidateList ← ∅
  For (S_candidate ∈ Sbest_neighborhood)
    If (¬ ContainsAnyFeatures(S_candidate, TabuList))
      CandidateList ← S_candidate
    End
  End
  S_candidate ← LocateBestCandidate(CandidateList)
  If (Cost(S_candidate) ≤ Cost(S_best))
    S_best ← S_candidate
    TabuList ← FeatureDifferences(S_candidate, S_best)
    While (TabuList > TabuList_size)
      DeleteFeature(TabuList)
    End
  End
End
Return (S_best)
```

Pseudocode for Tabu Search.

- Heuristics:

    1. Designed to manage an embedded hill climbing heuristic, although may be adapted to manage any neighbourhood exploration heuristic.
    2. Designed for and has predominantly been applied to discrete domains such as combinatorial optimization problems.
    3. Candidates for neighbouring moves can be generated deterministically for the entire neighbourhood or the neighbourhood can be stochastically sampled to a fixed size, trading off efficiency for accuracy.
    4. Intermediate-term memory structures can be introduced (complementing the short-term memory) to focus the search on promising areas of the search space (intensification), called aspiration criteria.
    5. Long-term memory structures can be introduced to encourage useful exploration of the broader search space, called diversification. Strategies may include generating solutions with rarely used components and biasing the generation away from the most commonly used solution components.

- Applications: Scheduling / Computer Channel Balancing / Cluster Analysis / Space Planning / Assignment / etc
  Technical problems like:
    1. **Travelling salesman**
    2. Graph Colouring
    3. Character Recognition

Advantages

- Random selection of a neighbouring solution
- Probabilistic acceptance of non-improving solutions
- Generate generally good solutions for optimisation problems compared with other AI methods
- The best solutions are recorded

Disadvantages

- Possibility of being trapped at a local optimum
- Is allowing the designer to select the maximum number of cells as well as machines in cell
- Not allowed to re-visit exact the same state that we have been before

- Exhaustive usage of memory resources
- Collecting more data than could be handled
- No guarantee of global optimal solutions

Normal Search Algorithm VS TABU Search Algorithm

	SA	TS
No. of neighbourhoods considered at each move	1	n
Accept worse moves? How?	Yes P = exp^(*c/t)	Yes The best neighbourhood if it is not tabu-ed.
Accept better moves?	Always	Always (aspiration)
Stopping conditions	T=0, or at a low temperature, or NO improvement after certain number of iterations.	Certain number of iterations or NO improvement after certain number of iterations.

TABU Search Algorithm in Travelling Salesman Problem

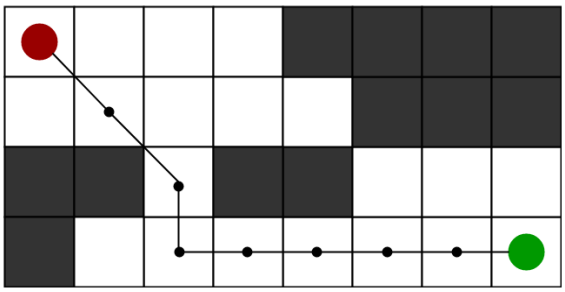
This problem poses a straightforward question given a list of cities: What is the shortest route that visits every city?

- Advantage
  The value of exploiting problem structure is a recurring theme in metaheuristic methods and tabu search is well-suited to this. A class of strategies associated with tabu search called ejection chain methods has made it possible to obtain high-quality TSP solutions efficiently.
- Disadvantage
  TABU Search Algorithm will not guarantee a global optimal solution but it will create another good solution to this problem and return to us the best solution found during its execution.

A* Search Algorithm

- Taxonomy: This is one of the most popular algorithms used in path-finding and other types of graphs. This concrete algorithm is particularly smart if we compare it with others of the same type.

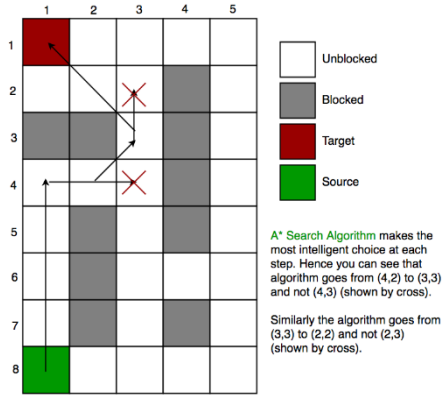
- Strategy: Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible, this is a typical problem for this algorithm.



What A* Search Algorithm does is that at each step it picks the node according to a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell and defining 'g' and 'h' as simply as possible below.

- g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
- h = the estimated movement cost to move from that given square on the grid to the destination.

- Procedure: It is created two lists: Open List and Closed List. First, we initialize the open list and the closed list and put the starting node on the open list.
  While the open list is not empty, we tried to find the node with the least f on the open list, call it "q" and pop q off the open list. After that is needed to generate q's 8 successors and set their parents to q. And now for each successor we make sure that the successor is successor.g = q.g + distance between successor and q and the successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics-Manhattan, Diagonal and Euclidean Heuristics) and last if successor.f = successor.g + successor. If a node with the same position as successor is in the open list which has a lower f than successor, we should skip this successor. If a node with the same position as successor is in the closed list which has a lower f than successor, skip this successor otherwise, add the node to the open list, then finished the loop. Finally push q on the closed list till the end of loop.



So, suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristic.

- Heuristics: To calculate h there are two methods:
  - Either calculate the exact value of h (which is certainly time consuming).
  - Approximate the value of h using some heuristics (less time consuming).

  For the exact Heuristics We can find exact values of h, but that is generally very time consuming. By pre-compute the distance between each pair of cells before running the A* Search Algorithm, the other way would be when here are no blocked cells/obstacles, then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance.
  For the approximation heuristics the methods to calculate the h are Manhattan Distance which is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively. The Diagonal Distance that calculate the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively. And the Euclidean Distance where the distance between the current cell and the goal cell is calculated using the distance formula.

- Application: Tower defence is a type of strategy video game where the goal is to defend a player's territories or possessions by obstructing enemy attackers, usually achieved by placing defensive structures on or along their path of attack. A* Search Algorithm is often used to find the shortest path from one point to another point. You can use this for each enemy to find a path to the goal. One example of this is the very popular game- Warcraft III.

Advantages

- It is complete and optimal.
- It is the best one from other techniques. It is used to solve very complex problems.
- It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

Disadvantages

- Although being the best pathfinding algorithm around, A* Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h
- This algorithm is only complete if the branching factor is finite and every action has fixed cost.
- The speed execution of A* search is highly dependent on the accuracy of the heuristic algorithm that is used to compute h (n).
- It has several complexity problems.

Normal Search vs A*

- A* search is better in every way if you can do it. The only time you wouldn't do it is if you did not have the global graph data and needed to explore to find out the shape of the graph.
- If you did have global graph data but did not have an available admissible heuristic for the destination, you'd still want to at least try Dijkstra's and not fall all the way back to normal search. Normal search is extremely prone to wandering down dead ends and having to backtrack a lot. That's inevitable if you're in a maze you know nothing about, but if you know where every edge is going to lead and can pick the shortest path just by considering all currently available options to generate the shortest path, you should do it.

Traveling salesman problem advantage and disadvantages with a*

- A* is a derivative of Dijkstra, which I don't think can be used in this fashion. First, the TSP generally starts from any node. More importantly though, these algorithms seek to find the shortest path between two points, irrespective of the number of nodes visited. In fact, they depend on the fact that the shortest path from S to T via some node A, the path from S to A is irrelevant if it's the same cost.

- The only way for this functioning is if you generated a new graph representing nodes visited. So is not that much beneficial as the other algorithms could be.

Genetic Search Algorithm

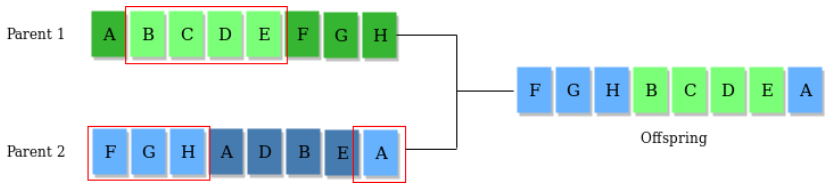
- Taxonomy: Is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms.

- Strategy: In a genetic algorithm, a population of candidate solutions to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties which can be mutated and altered. Traditionally solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.

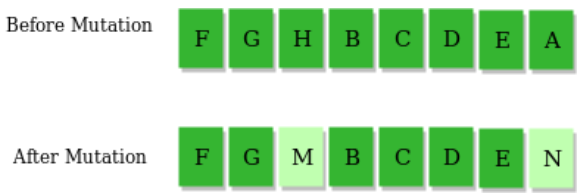
- Procedure: 3 Steps:

    1. Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to the successive generations.

    2. Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged creating a completely new individual.



    3. Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence.



```
public void find() {
    // Initialization
    List<T> population = Stream.generate(supplier)
                               .limit(populationSize)
                               .collect(toList());
    // Iteration
    while (!termination.test(population)) {
        // Selection
        population = selection(population);
        // Crossover
        crossover(population);
        // Mutation
        mutation(population);
    }
}
```

- Heuristics:

    1. The capability of GA to be implemented as a 'universal optimizer' that could be used for optimizing any type of problem belonging to different fields.
    2. Simplicity and ease of implementation.
    3. Proper balance between exploration and exploitation could be achieved by setting parameter properly.
    4. Logical reasoning behind the use of operators like selection, crossover and mutation.
    5. Mathematical or theoretical analysis in terms of schema theory or Markov chain models for the success of GA.
    6. One of the pioneer evolutionary algorithms.

- Applications: Chemical Kinetics / Code-breaking / Calculation of bound states and local-density approximations / Computer Architecture / Configuration Physics Applications / Data Server Farm / Electronic Circuit Design / Feynman-Kac models / Finding Hardware Bugs / Game Theory Equilibrium Resolution / Power Electronics Design / **Travelling Salesman Problem** and its applications / etc

Advantages:

- The concept is easy to understand
- Search from a population of points, not a single point.
- Use payoff information, not derivatives.
- Supports multi-objective optimization.
- Use probabilistic transition rules, not deterministic rules.
- Good for "noisy" environments.
- Robust w.r.t. to local minima/maxima.
- Easily parallelize.
- Can operate on various representations.
- Stochastic.
- Works well on mixed discrete/continuous problem.

Disadvantages:

- Implementation is still an art.
- Requires less information about the problem but designing an objective function and getting the representation and operators right can be difficult.
- Computationally expensive time-consuming.

- Advantage: Classics crossover and mutation operators can be used without the necessity of designing new operators. MPX operator is that it only destroys a limited number of edges. Allows hyper-plane analysis. Classical crossover operator can be used.
- Disadvantage: Classical operators do not necessarily result in legal offspring's tours; repair algorithms would be necessary. For larger problems instances the binary strings which represents the tours become unmanageably large.

What is overfitting training problem and how to solve it in neural networks and specially in Keras.

Overfitting refers to a model that models the training data too well. Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data.
Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. As such, many nonparametric machine learning algorithms also include parameters or techniques to limit and constrain how much detail the model learns.

EXAMPLE:

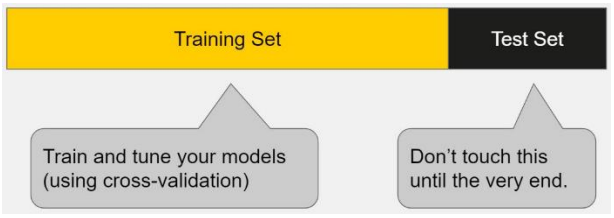
Let's say we want to predict if a student will land a job interview based on her resume.
Now, assume we train a model from a dataset of 10,000 resumes and their outcomes.
Next, we try the model out on the original dataset, and it predicts outcomes with 99% accuracy…
But now comes the bad news.
When we run the model on a new ("unseen") dataset of resumes, we only get 50% accuracy…

Our model does not generalize well from our training data to unseen data.

In fact, overfitting occurs in the real world all the time. You only need to turn on the news channel to hear examples.

HOW TO SOLVE OVERFITTING

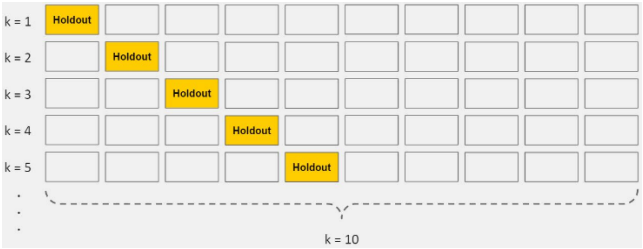
1. Detect it: A key challenge with overfitting and with machine learning in general, is that we cannot know how well our model will perform on new data until we test it.
   To address this, we can split our initial dataset into separate training and test subsets.



   This method can approximate of how well our model will perform on new data.
   If our model does much better on the training set than on the test set, then we are likely overfitting.
   Another tip is to start with a very simple model to serve as a benchmark.

2. Prevent it: Detecting overfitting is useful, but it does not solve the problem. Fortunately, you have several options to try.

    - Cross validation: Is a powerful preventive measure against overfitting.
      The idea is to use your initial training data to generate multiple mini train-test splits, use these splits to tune your model.
      In standard k-fold cross-validation, we partition the data into k subsets, called folds. Then, we iteratively train the algorithm on k-1 folds while using the remaining fold as the test set.

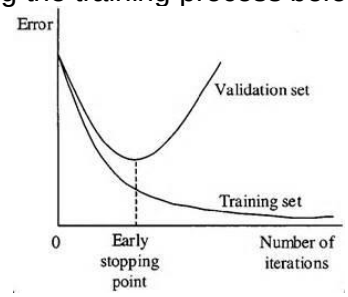


Cross-validation allows you to tune hyperparameters with only your original training set. This allows you to keep your test set as a truly unseen dataset for selecting your final model.

- Train with more data: It will not work every time, but training with more data can help algorithms to detect the signal better.

- Remove features: Some algorithms have built-in feature selection. For those that do not, you can manually improve their generalizability by removing irrelevant input features.

- Early stopping: When you are training a learning algorithm iteratively, you can measure how well each iteration of the models performs. Up until a certain number of iterations, new iterations improve the model. After that point, however, the model's ability to generalize can be weaken as it begins to overfit the training data.
- Early stopping refers stopping the training process before the learner passes that point.



- Regularization: Refers to a broad range of techniques for artificially forcing your model to be simpler.

- Ensembling: Ensembles are machine learning methods for combining predictions from multiple separate models.

OVERFITTING IN KERAS - HOW TO SOLVE IT

1. Download the IMDB dataset:

```
num_words <- 10000
imdb <- dataset_imdb(num_words = num_words)

c(train_data, train_labels) %<-% imdb$train
c(test_data, test_labels) %<-% imdb$test
```

2. Demonstrate overfitting: Reduce the size of the model. In deep learning, the number of learnable parameters in a model is often referred to as the model's "capacity".
   On the other hand, if the network has limited memorization resources, it will not be able to learn the mapping as easily. It will have to learn compressed representations that have more predictive power.
   Unfortunately, there is no magical formula to determine the right size of your model, you will have to experiment using a series of different architectures.

3. Create a baseline model:

```
baseline_model <-
  keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = 10000) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

baseline_model %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = list("accuracy")
)

baseline_model %>% summary()
```

4. Create a smaller model: Create a model with less hidden units to compare against the baseline model that we just created and train the model using the same data.

```
smaller_model <-
  keras_model_sequential() %>%
  layer_dense(units = 4, activation = "relu", input_shape = 10000) %>%
  layer_dense(units = 4, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

smaller_model %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = list("accuracy")
)

smaller_model %>% summary()
```

5. Create a bigger model: Add to this benchmark a network that has much more capacity, far more than the problem would warrant and train the model using the same data.

```
bigger_model <-
  keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = 10000) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

bigger_model %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = list("accuracy")
)

bigger_model %>% summary()
```

6. Plot the training and validation loss: See the results.

```
compare_cx <- data.frame(
  baseline_train = baseline_history$metrics$loss,
  baseline_val = baseline_history$metrics$val_loss,
  smaller_train = smaller_history$metrics$loss,
  smaller_val = smaller_history$metrics$val_loss,
  bigger_train = bigger_history$metrics$loss,
  bigger_val = bigger_history$metrics$val_loss
) %>%
  rownames_to_column() %>%
  mutate(rowname = as.integer(rowname)) %>%
  gather(key = "type", value = "value", -rowname)

ggplot(compare_cx, aes(x = rowname, y = value, color = type)) +
  geom_line() +
  xlab("epoch") +
  ylab("loss")
```

Give an original example of reasoning based on backward chaining.

Backward chaining starts directly with the conclusion (hypothesis) and validates it by backtracking through a sequence of facts being the exact opposite method as forward Chaining.
When comparing forward chaining and backward chaining, the first one can be described as “data-driven” (data as input), while the latter one can be described as “event (or goal)-driven” (goals as inputs).
We are going to create an example and try to validate a simple hypothesis – if the Great Wall of China is on Planet Earth.
To do this task we are going to create a project in Drools. Drools is a business rule management system (BRMS) with a forward and backward chaining inference-based rules engine, more correctly known as a production rule system, using an enhanced implementation of the Rete algorithm. Drools is open source software, released under the Apache Software License.

We should start by creating a simple fact base describing things and its location:

1. Planet Earth
2. Asia, Planet Earth
3. China, Asia
4. Great Wall of China, China

Starting by creating some query that will do the reasoning like:

```
query belongsTo(String x, String y)
        Fact(x, y)
        or
        (Fact(z, y;) and belongsTo (x,z;))
```

With this function the program examines and reason, but it needs the rules for the fully application. The rules are going to be:

```
rule “Great Wall of China BELONGS TO Planet Earth”
when
        belongsTo(“Great Wall of China”, “Planet Earth”;)
then
        result.setValue(“Decision one taken: Great Wall of China BELONGS TO Planet Earth”);

rule “print all facts”
when
        belongTo(element, place;)
then
        result.addFact(element + “IS ELEMENT OF “ + place);
```

And now for creating the application to set the parameters, we would set the class Fact:

```
public class Fact {
private String Element;
private String Place;
}
```

Also creating another class to get the value anda list of facts we create the class Value:

```
public class Value{
private String value;
private List<String> facts = new ArrayList<>();
}
```

And run the example in another class:

```
public class BackwardChainingTest {

 @Before
  public void before() {
       result = new Result();
       ksession = new DroolsBeanFactory().getKieSession();
   }

   @Test
   public void whenWallOfChinaIsGiven_ThenItBelongsToPlanetEarth() {
```

```
        ksession.setGlobal("result", result);
        ksession.insert(new Fact("Asia", "Planet Earth"));
        ksession.insert(new Fact("China", "Asia"));
        ksession.insert(new Fact("Great Wall of China", "China"));

        ksession.fireAllRules();

        assertEquals(
          result.getValue(),
          "Decision one taken: Great Wall of China BELONGS TO Planet Earth");
    }
```

When the test cases are executed, they add the given facts (“Asia belongs to Planet Earth“, “China belongs to Asia”, “Great Wall of China belongs to China”).

After that, the facts are processed with the rules described in BackwardChaining.drl, which provides a recursive query belongsTo(String x, String y).

This query is invoked by the rules which use backward chaining to find if the hypothesis (“Great Wall of China BELONGS TO Planet Earth”), is true or false.

We want to solve the problem of travelling Sales problem (https://en.wikipedia.org/wiki/Travelling_salesman_problem), how would you solve it? What algorithm would you use and why? How should the problem be coded to solve it with the "tabu search" algorithm?

What is the best algorithm? GENETIC

Why? This problem is known to be NP-Hard. In order to increase the efficiency of the genetic algorithm, the initial population of feasible solutions is carefully generated so for many problems instances this algorithm can generate solutions with same value as the best-known solutions.

The population of a genetic algorithm evolves by using genetic operators, so the selection provides a reproduction of the strongest and more robust individuals, while the reproduction is a phase in which the evolution run.

Compared to the classical optimization algorithm, the genetic algorithm has several advantages in the TSP as:

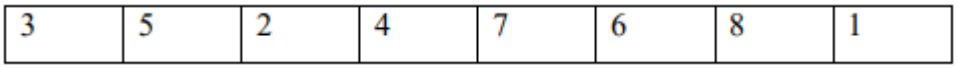
- It does not require any special property of the function to be optimized (Less time taken to find a solution in the TSP).
- Generation has a parallel form by working on several points at once (We can investigate and analyse more than one way at the same time in the TSP)
- The use of probabilistic transition rules as opposed to deterministic algorithms where the transition between two individuals is required by the structure and nature of the algorithm (Better than other algorithms).

This are the three main reasons to use GENETIC algorithm instead of other algorithms.

Solving TSP using TABU Search Algorithm

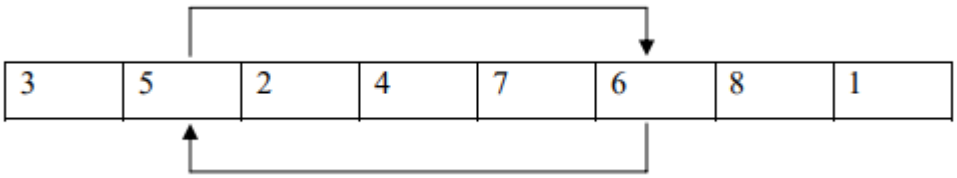
Tabu Search Algorithm is heuristic that, if used effectively, can promise an efficient near-optimal solution to the TSP. Steps to follow:

1. Solution Representation:



2. Initial Solution: Starting with the first node in the tour, find the nearest node. Each time find the nearest unvisited node from the current node until all the nodes are visited.

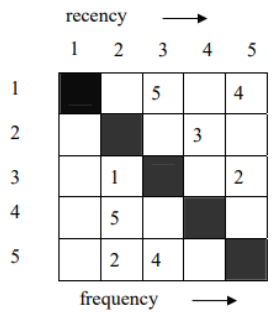
3. Neighbourhood: If we fix node 1 as the start and the end node, for a problem of N nodes, there are ${}^{N-1}C_2$ such neighbourhoods to a given solution. At each iteration, the neighbourhood with the best objective value is selected.



4. Tabu List: Prevents the process from cycling in a small set of solutions for a limited period. The attribute used is a pair of nodes that have been exchanged recently. This structure stores the number of iterations for which a given pair of nodes is prohibited from exchange.

5. Aspiration criterion: Tabus may sometimes be too powerful. The criterion used for this to happen in the present problem of TSP is to allow a move, even if it is tabu, if it results in a solution with an objective value better than that of the current best-known solution.

6. Diversification: To allow the process to search other parts of the solution space, it is required to diversify the search process. This is implemented in the current problem using “frequency-based memory”. This diversifying influence can operate only on occasions when no improving moves exist.



7. The algorithm terminates if a pre-specified number of iterations is reached.

Bibliography used for the first work part:

(Reinforcement Learning)
https://towardsdatascience.com/reinforcement-learning-tutorial-with-open-ai-gym-9b11f4e3c204
https://towardsdatascience.com/deep-reinforcement-learning-tutorial-with-open-ai-gym-c0de4471f368
https://towardsdatascience.com/summary-of-tabular-methods-in-reinforcement-learning-39d653e904af
(Deep learning)
https://medium.com/@jonathan_hui/rl-deep-reinforcement-learning-series-833319a95530
https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4
https://medium.com/@jonathan_hui/rl-introduction-to-deep-reinforcement-learning-35c25e04c199
https://towardsdatascience.com/applications-of-reinforcement-learning-in-real-world-1a94955bcd12
https://blog.dominodatalab.com/deep-reinforcement-learning/

(Double Q Learning)
https://papers.nips.cc/paper/3964-double-q-learning
https://dl.acm.org/doi/10.5555/2997046.2997187

Bibliography used for the second work part:

(Overfitting)
https://keras.rstudio.com/articles/tutorial_overfit_underfit.html
https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/
https://elitedatascience.com/overfitting-in-machine-learning#examples

(Genetic Algorithm)
https://electricalvoice.com/genetic-algorithm-advantages-disadvantages/
https://www.researchgate.net/post/Why_genetic_algorithms_is_popular_than_other_heuristic_algorithms
https://www.toptal.com/algorithms/genetic-algorithms
https://www.geeksforgeeks.org/genetic-algorithms/
https://www.sciencedirect.com/topics/engineering/genetic-algorithm

(TABU Algorithm)
https://www.researchgate.net/publication/329917885_Traveling-Salesman-Problem_Algorithm_Based_on_Simulated_Annealing_and_Gene-Expression_Programming
https://www.researchgate.net/post/Comparison_of_advantages_and_disadvantages_meta-heuristics_algorithm
https://www.hindawi.com/journals/mpe/2015/947021/

(A* Algorithm)

https://ieeexplore.ieee.org/abstract/document/8790018
https://www.geeksforgeeks.org/a-search-algorithm/
http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK2/NODE49.HTM
https://www.sciencedirect.com/science/article/abs/pii/0305054878900333
https://www.brainkart.com/article/A--Search--Concept,-Algorithm,-Implementation,-Advantages,-Disadvantages_8883/

(Drools project for backward chaining example)
https://www.drools.org/