Introduction to Artificial Intelligence

Hisar CS

hisar computer science

İçindekiler

Intro

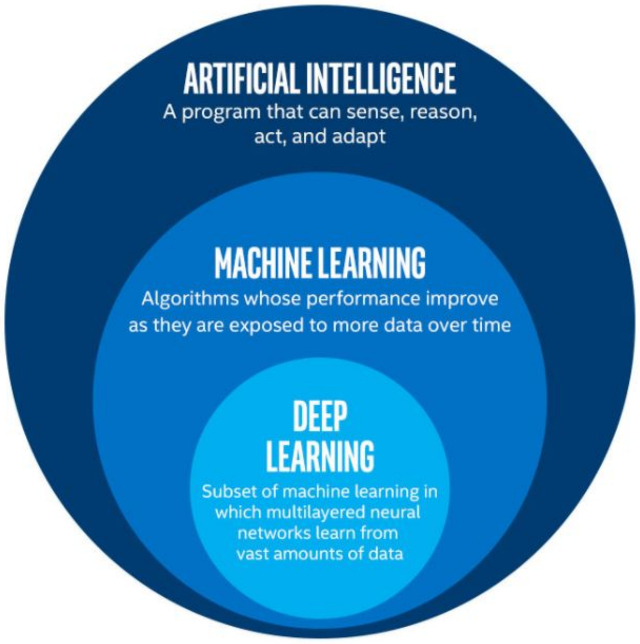
Artificial intelligence is a topic that we see almost everyday in our daily lives. From movies, to videogames and other sources of media. Even though we can’t create machines that control entire armies of robots to conquer planets, or create new gadgets and designs on command, we have our own version of artificial intelligence, even if it is a bit more trivial compared to all those listed

above. They may not be autonomous like we see in the movies, but in the state they are in right now, they act as great companions to humans in their works, especially when trying come up models to make future predictions.
Right now computer science professionals are working on new methods and algorithms to increase the capabilities of the already existing technology. Even though the world seems to be obsessed with artificial intelligence, we still don't know the limitations of the technology, if there is any, or if we can come up with a way to completely transfer a human brain to a digital database.

Artificial Intelligence

What is artificial intelligence

Artificial intelligence as defined means: "the theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages." Basically meaning as the word implies, an intelligence or an intelligent being that is able to think and rationalize. Artificial intelligence is the pursuit of creating a digital intelligence that can possibly help us improve as a species overall, in places where human emotion or limitations such as requiring sleep and rest may hinder us. The inspiration for artificial intelligence comes from the nature as all other sciences once were. Scientists and researchers are trying to replicate phenomenon in nature that allows even the smallest critters to make seemingly intelligent decisions to at least keep them alive. Even though we may underestimate these animals, the furthest we've gotten is digitize the brain of a worm, that took years of development and research, which only wobbles around for now.(open worm) [1]
Artificial intelligence also has some categories underneath itself. These are: "Machine learning", and "Deep learning". These will be covered in detail later on, but for now we won't go too in depth.


ARTIFICIAL INTELLIGENCE
A program that can sense, reason, act, and adapt
MACHINE LEARNING
Algorithms whose performance improve as they are exposed to more data over time
DEEP LEARNING
Subset of machine learning in which multilayered neural networks learn from vast amounts of data

Machine learning

Sometimes the term “machine learning” is confused with “artificial intelligence”. While artificial intelligence is the overall arching research topic, machine learning is almost a subdivision of it. Machine learning is based on the research and development of new algorithms that can be “trained” on large datasets that are structured as tables or in cases images to create models that can later be used to make future predictions or analyzing data.

Artificial Intelligence

What we call artificial intelligence is the overall arching theme as we mentioned earlier that aims to create a digital intelligent being.
“Yapay zeka” dediğimiz alan ise bu makine öğreniminin özel bir alt bölümde kullanılarak zeki ve daha özelleştirilmiş bir program yaratmak amacıdır. Bu programlar tıpkı insanlar gibi mantık yürütme yetisine sahip şekilde geliştirilmekte, ama kim bilir gelecekte neler olabilir.

qDeep Learning

“Deep learning” on the other hand is a subdivision of machine learning. Deep learning focuses on the analysis, segmentation and classification of images. This is where the well known “neural networks” are used. These neural nets are the foundation of image classification and identification.

Artificial Intelligence Applications

Without knowing it, we use at least one up to a several different machine learning or artificial intelligence applications. Apps such as Youtube and Facebook that customize your feed based on the type of content you are most likely to click on. Netflix that suggests movies or series you'd most likely enjoy, based on your past tastes and on similar customers. Spotify that creates custom playlists from out of the blue that usually matches with your tastes. Or simply the ads that pop up next to the screen that magically seems to suggest products that are similar to what you recently searched for. The applications of artificial intelligence is still a growing market, and as it seems it will continue like this for a long time. As we speak companies such as Autodesk are working on ways that allows computers to create structures given certain constraints.[2] (https://www.autodesk.com/solutions/generative-design/manufacturing) As artificial intelligence increases in complexity, the limits of what we imagined possible will be challenged, and inevitably changed.

History of artificial intelligence

Towards the end of the second world war, the nazi war machine created a contraption that baffled even the top level researchers in the world, Enigma. The enigma machine was a state of the art encryption device that used a "simple" mechanism to encrypt the messages sent. This machine was ultimately deciphered by "Bombe" created by Alan Turing. Both of these machines altered the way people saw computers and where the world is heading. The concepts of machine learning can be said to have started with these two machines. The creator of Bombe Alan Turing later went on to say that "a machine that could converse with humans without the humans knowing that it is a machine would win the "imitation game" and could be said to be "intelligent"."

In 1956, John McCarthy organized the Dartmouth conference. Many of the leading experts in the field of computer science was invited. This conference was where the term "artificial intelligence" was originally coined. After this conference the surge of artificial intelligence began with research centers popping up all over the US. Everyone was trying to get a glimpse into the world of artificial intelligence and see how far it could be pushed. Allen Newell and Herbert Simon were crucial in promoting the field and research.

In 1951 the "Ferranti Mark 1" was built. This machine learned how to play checkers using an algorithm devised by hand. Along the same time, Newell and Simon managed to create an algorithm that solved general mathematical problems. Also in the 50s McCarthy created a new language called LISP that became important in the world of machine learning.

In the 1960s researchers focused on algorithms that would solve the geometric theories and mathematical problems. Towards the end of the 60s they worked on "computer vision" and learning machines. In 1972."WABOT-1" was the first "intelligent" humanoid robot created in Japan.

Despite their best efforts researchers couldn't succeed in creating intelligent machines. Mostly due to the inefficient computing capabilities of the computers of the time. For applications such as machine vision to work effectively, there had to be a large amount of data that had to be processed quickly. Due to the slowing advances, both the public and government opinion of artificial intelligence fell greatly. Because of this, the years 1970-1990 were known as the "AI winter" where close to none new advancements were made due to a lack of funding and researchers. Despite this, the efforts of these researches helped advance general computing technologies greatly.

Towards the end of the 1990s the interest in artificial intelligence began to rise again. The Japanese government announced that they would be working on generation 5 computers to further advance in the field of computer science. Hearing this, the fanatics of artificial intelligence were thrilled at the idea of machines that would translate languages, interact and possibly reason just as we would. In 1997 IBM's "Deep Blue" computer managed to beat the world champion Garry Kasparov in a game of chess.

In our current times, we can see how far the world of artificial intelligence has come. In the last 15 years, companies such as Google, Amazon and Microsoft have made great investments to develop algorithms that can cater to their users better and advance the field of artificial intelligence. As this technology is being opened up to the public, more and more students can finally get a taste of artificial intelligence easier than at any time in history. These students use artificial intelligence and machine learning concepts to work on project, join competitions and even help advance the research efforts. Who knows with the huge amount of eager people working on this new field, where it will go in the future.

Machine Learning Concepts

Intro/Explanation

The term machine learning is a part of artificial intelligence as mentioned before. We will go more in depth into what machine learning is in this section and look at a few math concepts behind this field. This field is based on maths, statistics and data analysis. In a way machine learning is the field of analyzing data using computers to find trends, statistics and underlying connections. Machine learning helps us analyze data more effectively. It also allows us to make computers find algorithms that fit the best for these types of applications. To put into simple terms: Machine learning is the act of trying to teach a baby colors shapes and sounds by exposing them to such material thousands and thousands of times.

How do machines learn?

As we mentioned before, machine learning is a statistics game. Programmers use algorithms to process thousand of data points to find trends, similarities and general connections. For the most part, how machines learn is a mystery, even to those who supplied the data and wrote the algorithm. This is because the computer follows the algorithm provided and tweaks hundreds or sometimes thousands of values in tiny increments until it comes to a point where it's able to discern between a lion and a penguin for example. This is actually good thing, as it would be simply impossible for a human to tweak thousands of values and run the calculations each time and identify what works and what doesn't. Even if it was done, it wouldn't be as effective and would be purely luck. Of course this process has its own terminology, and this is what we'll look at in the upcoming sections.

Training and Testing

The name is mostly self explanatory. Algorithms have to learn or be taught how to make predictions from future date. Basically, we have to teach our computer what a penguin and a lion looks like before expecting it to tell which is which. "Training" is how we accomplish this. Training is the process of using many data points to allow the algorithm to figure out trends in the data provided and accomplish what we want it to. For example, we'd give an algorithm penguin pictures for it to then make predictions on if a future picture presented is a penguin or not. During training, as more and more data is processed, the weights and values of our algorithms, which we will get into later, can be tweaked and changed to fit the data provided and make a proper estimate. This process takes lots and lots of data and computing power, but luckily a laptop of today is at least 100 times stronger than the old computers used back in the days we mentioned earlier. After we tweak all of these values, we need a way to actually evaluate our model, or perhaps test it in some ways which iis where "Testing" comes in.

"Testing" as you can tell from the name is the process of putting our algorithm to the test and see how reliable it is on a dataset that it has never seen before and compare the results with the actual performance. We need a dataset that already has the correct values for a the desired data however. We can't simply throw in a new piece of data and expect our algorithm to magically see if it got something wrong or not. This can be thought of as us giving a test to our model, which we then control using the answer key on the back. For testing we need to reserve some of our data to be used as testing data, if we have a finite amount of data of course. After testing, rearranging data or our algorithm to get a better result, we can start using our model to make predictions or use them in other applications.

.

Supervised Learning

Up till now we always talked about supervised learning but we never got a name for it. "Supervised Learning" is the most commonly used technique to train models, in which the data given has a desired outcome. For example, evaluating the risk of a cancer tumor or identifying pictures of penguins. Meaning the output of the input data training data is also part of the solution. It's almost like trying to learn a subject purely from doing questions on a test book by looking at the answer key at the back. The model tweaks it's values to reach the given answer. This method is mostly employed in applications that involve image classification or making future prediction from table data.

Unsupervised learning

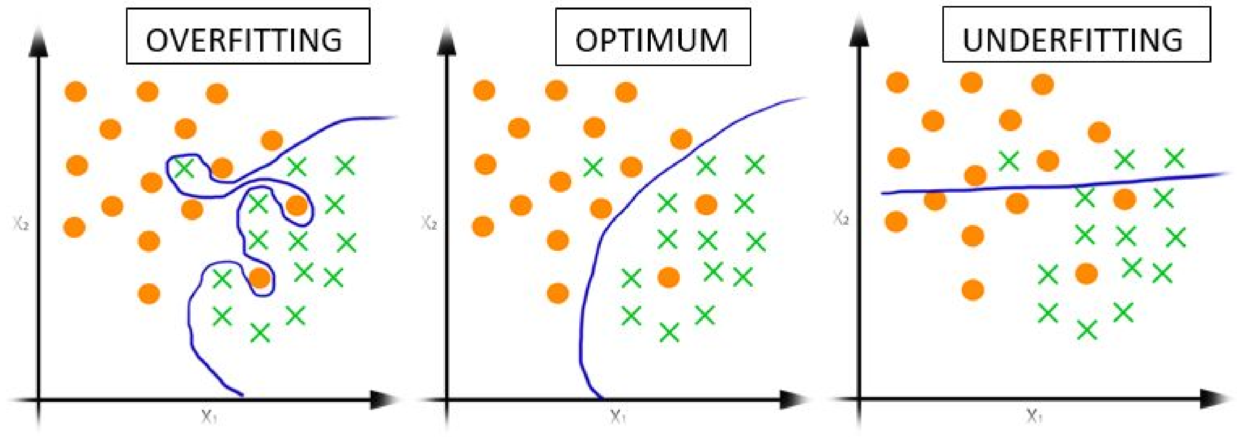
This training system is a bit different in it's application. We supply our model with data set just as we would in supervised learning, but this time we don't have the answer key. This model is simply expected to come up with a solution to the dataset in some way. These types of algorithms are mostly used for clustering, and helping humans in categorizing data and finding underlying connections within a dataset. Clustering algorithms help identify hotspots in data where data points commonly cluster to find trends. An example of this type of learning would be the Facebook wall feed that users receive based on their past activity and interests. Facebook finds the demographic of people that cluster around certain points to see what these users usually click on or what they like to see on their feed. These algorithms also help with research that involve large demographics by supplying researchers with large points of interests and trends within these datasets. Even though it seems like a completely random system, it still has many applications.

Over Fitting,

"Over Fitting" is a phenomenon that occurs when the model is too strictly defined on a limited set. This happens mostly, when there is a lack of data to make a coherent model for future prediction. Overfitted models may perform extremely well on the training set, however they are limited in terms of making estimations from newly given data. Most of the time this occurs because the model is forced into covering every turn and crevice of a graph or dataset, and ends up being so strict that a single digit may flip the entire prediction.

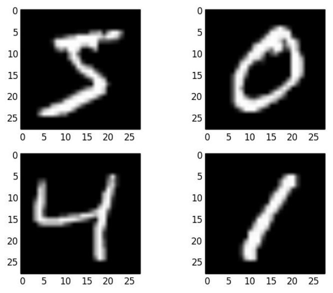
Underfitting

"Underfitting" is the polar opposite of the problem discussed above. This occurs when our model can't get the underlying connections, or when it doesn't encompass the data well enough, or gives an extremely vague thesis for future predictions. Specifically, underfitting occurs if the model or algorithm shows low variance but high bias. Underfitting usually stems from an overly simple model.



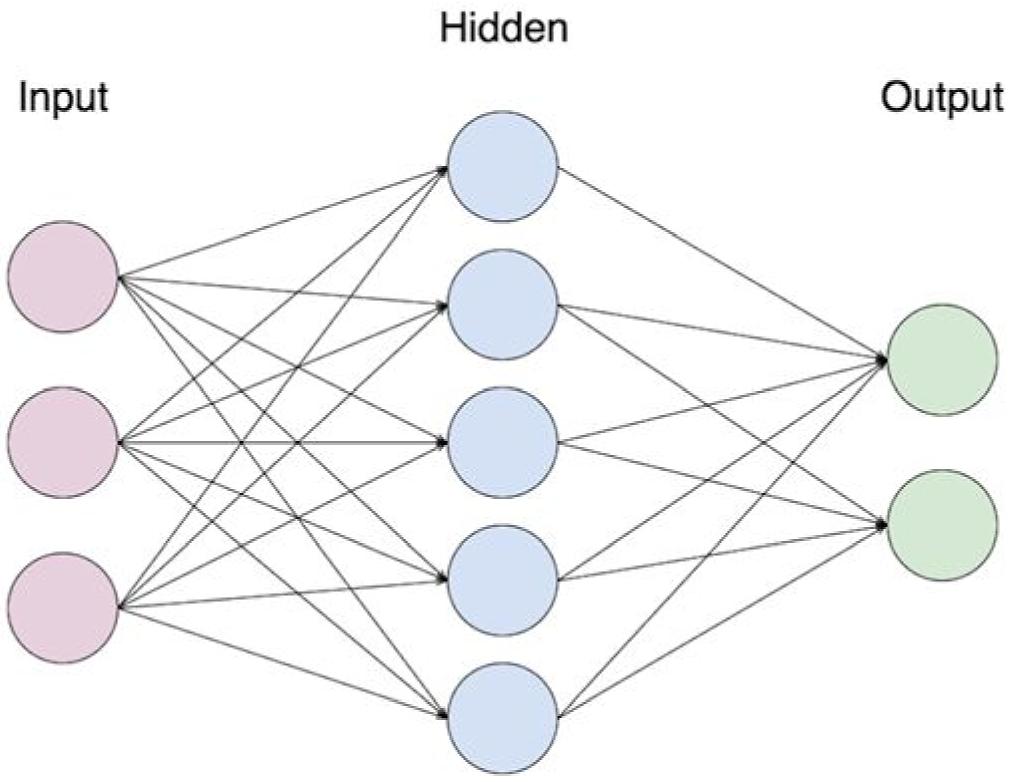
Algorithms of machine learning

After talking about algorithms so much, let's learn a bit about what these algorithms actually are. Algorithms utilize mathematical and statistical models, usually developed for probability and statistical analysis, to make a valid thesis to make future predictions. These algorithms are run many times over large datasets to develop a model that is capable of predicting from new data.
These models then extract certain abstract features and tweaks these values to make predictions.



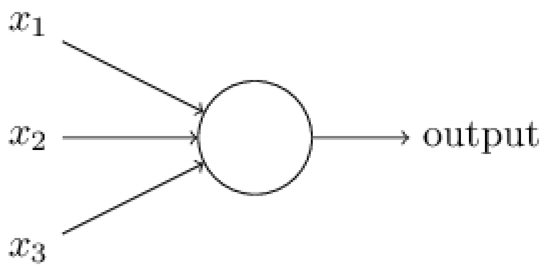
Neural Networks

Probably the most used buzz word when it comes to machine learning and artificial intelligence, but what are they? "Neural Networks" are the widely used algorithm type in Deep learning that works with image processing and computer vision. The word "deep" in deep learning actually comes from the neural nets used, reffering to the depth of the neural networks and their layers.
For example, let's take these four pictures. It's very easy for us to tell which is a 5, 4 or 0, but how do we make a machine understand that? Maybe we could say if there is a circle with a line drawn from the bottom it's a nine. When we calibrate it to identify one number however, what about the other nine? What about all the other types of nines people write? This is where the idea of a deep convolutional network comes in. These networks train on images that have been labelled with their respective answer. They then extract certain features that we can visualize later on, but is completely up to the machine and what kind of data was used. These allow us to bypass the wall computer scientists had for a very long time of hand coding certain features into computers. With this method we use an algorithm to almost write a new algorithm to do a task we want from it. Isn't that the definition of artificial intelligence to some extent?



Perceptron

Perceptrons can be thought of as the main building blocks of a neural network. Perceptrons are basically one layer neural networks, so in order to understand how a neural net operates, we should learn about the perceptrons. Perceptrons were first developed between 1950s and 1960s by Frank Rosenblatt. Basically speaking, perceptrons take in a number of inputs and output a certain value based on their activation function.



For example we have a simple perceptron with 3 inputs and one output. What we do is, we take the weighted sum of all the input values from x1 to x3 and give an output value of one or zero. Let’s give an example that’ll help you understand the general concept. We’ll get to more realistic examples soon. Let’s say there is this festival on the weekend and you’re considering whether you should go or not. Let’s consider our circumstances.

1. Are any of my friends going?
2. Is there a show on Netflix that just came out?
3. Is it easy to get to and leave?

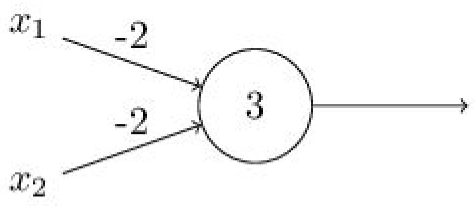
Let’s say we give a 1 value to those that are true, and 0 to those that are false. Let’s suppose all of your friends are going and it’s fairly easy to get to, but a new series comes out just on that day and you have snacks already prepared for an entire day of binge watching. This is where the “weights” come in. We said that these inputs are either 0 or 1, but the weights change things slightly. In our example, the choice involving binge watching would be higher, and this may change your decision to go, even though the other ones would give you the result to go to the festival. The algorithms we use learn these weights from training on thousands of data points and fine tuning each one to extract features. Put into maths terms, we have an equation like this that basically tells us to take the weighted sum of all the values and given a certain threshold value, give either a 1 or 0.

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \qquad (1)$$

Now to take things a bit further. We’ll write the above equation in dot product form. Meaning we’ll be treating weights and inputs as vector quantities. $W \cdot x \equiv \sum_j w_j x_j$: the W and X have the components weights and inputs respectively. Then we move the threshold to the other side and replace it with a “bias” value of the perceptron. So we finally get the equation:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \qquad (2)$$

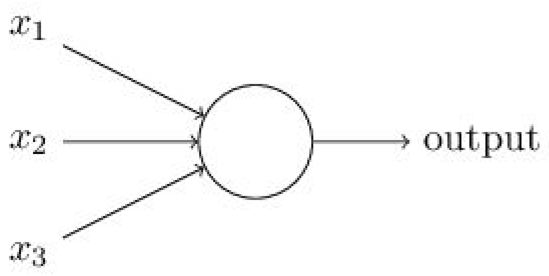
The bias value here determines how easy it is to get this particular node to fire a 1. Or in biological terms, how easy it is for this neuron to fire.



Let’s see a simple example. The perceptron above has -2 values for both of it’s inputs. Let’s suppose x1 = 1 and x2 = 0. We do the simple computation of (1) * -2 + (0) * -2 + 3 = 1. So we fire a value 1 to the next perceptron.

Sigmoid neurons:

Now we’ll take a look at a slightly different version of a neuron. Let’s suppose we have a neural net that we’d like to adjust our weight values to train. This is just how learning works, the constant update and tweaking of variables in a huge network. So we tweak and tweak on a perceptron, but the problem arises from the simplicity of the perceptron. Changing the bias or weights of a perceptron could possibly switch the entire outcome due to the nature of the threshold or the bias function used allowing a small change to make large differences. Sigmoid neurons handle the output value slightly different.



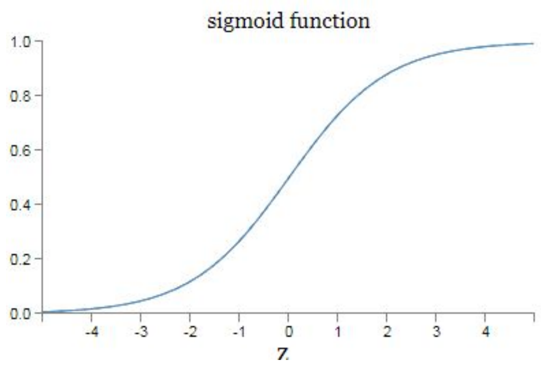
They are structured exactly the same way as perceptrons, but instead of a strict 1 or 0 threshold, the output is determined on a sigmoid graph. For example the output or the input of this type of neuron can be 0.784, as long as it's within the 0 < x < 1 boundary. These neurons also have a bias and weight value however the formula is written as σ(w · x+b). The new σ sign added is "sigma" and it's formula is as given:

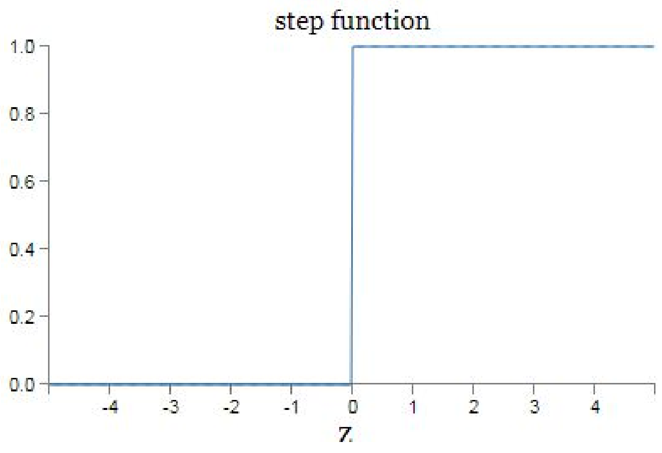
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \qquad (3)$$

Extend the formula and see the entire product:

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}. \qquad (4)$$

At first glance they may seem completely different, but they work on the same concept and formulas, with a few differences to the way it handles inputs and outputs. Let's suppose the z≡w · x+bz≡w · x+b formula used in the definition of the σ is a large positive number. Then our *e* value would look a little like this: e^{-z}≈0e−z≈0. Using this to continue our calculations we get: σ(z)≈1σ(z)≈1. So our neuron would give us a 1 output regardless if it was a sigmoid or a simple perceptron. The exact opposite with a large negative value would also give us a 0. We can understand it more clearly from looking at the two graphs:



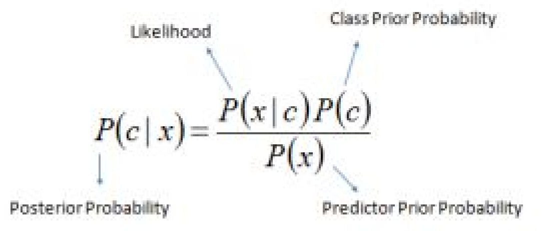


Logistic Regression

Instead of predicting *exactly* 0 or 1, logistic regression generates a probability—a value *between* 0 and 1, exclusive. For example, consider a logistic regression model for spam detection. If the model infers a value of 0.932 on a particular email message, it implies a 93.2% probability that the email message is spam. More precisely, it means that in the limit of *infinite* training examples, the set of examples for which the model predicts 0.932 will actually be spam 93.2% of the time and the remaining 6.8% will not.

Naive Bayes

This classification model utilizes the Naive Bayes theorem. The theory works on the basis that the factors given with the question aren't affected by each other and are completely independant in contributing to the final prediction probability. For example saying if a fruit is about 10 cm in circumference, is red and round, it can be an apple. The theorem works on the premise that the fruit being red and it being round don't affect each other and that these each add to the final probability on their own.



$$P(c \mid x) = \frac{P(x \mid c)\,P(c)}{P(x)}$$

$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

Pros:

- It is easy and fast to predict class of test data set. It also perform well in multi class prediction
- It is very helpful if we are using a multiple classification model,
- If the features given are actually independent of one another, the naive bayes outperforms other algorithms such as linear regression and requires less data to converge.
- Works well when using categorical and numerical inputs.

Cons:

- If the test set has features not present in the training set, these categories are simply given a 0 probability, hurting the final reliability of the predictions. This is called a “Zero Frequency”. You can adopt alternate methods such as “Laplace Estimation” to combat these problems.
-
- Naive Bayes aynı zamanda kötü bir tahmin algoritması olarak bilinmektedir. Bu sebeple olasılık çıktısında “predict_proba” genelde umursanmaz.
- It’s hard to find data that is actually completely separated from each other in the real world, which hurts the reliability.

Reinforcement Learning Notes

What is Reinforcement Learning?

Most of you have probably heard of AI learning to play computer games on their own, a very popular example being Deepmind, also known with their program called AlphaGo that defeats the Korean Go world champion in 2016. There had been many developments about AI, intenting to play games like Breakout, Pong, Space Invaders and Doom.

Each of these programs are developed by using a paradigm of machine learning known as Reinforcement Learning. If you have never heard something like Reinforcement Learning before, basically it is an algorithm that learns what should it do from scratch by experimenting.

Reinforcement Learning Analogy

Consider the scenario of teaching a dog new tricks. The dog doesn't understand our language, so we can't tell him what to do. Instead, we follow a different strategy. We emulate a situation, and the dog tries to respond in many different ways. If the dog's response is the correct one, we reward them with snacks. Now guess what, the next time the dog is encounter to the same situation, the dog executes a similar action with even more enthusiasm in expectation of more food. That's like learning "what to do" from positive experiences. Similarly, dogs will tend to learn what not to do when face with negative experiences.

That's exactly how Reinforcement Learning works in a broader sense:

Your dog is an "agent" that is exposed to the **environment**. The environment could in your house, with you.

The situations they encounter are analogous to a **state**. An example of a state could be your dog standing and you use a specific word in a certain tone in your living room

Our agents react by performing an **action** to transition from one "state" to another "state," your dog goes from standing to sitting, for example.

After the transition, they may receive a **reward** or **penalty** in return. You give them a treat! Or a "No" as a penalty.

The **policy** is the strategy of choosing an action given a state in expectation of better outcomes.

So basically we can conclude that Reinforcement Learning is the science of making optimal decisions using experiences. Breaking it down, the process of Reinforcement Learning involves these simple steps:

1. Observation of the environment
2. Deciding how to act using some strategy
3. Acting accordingly
4. Receiving a reward or penalty
5. Learning from the experiences and refining our strategy
6. Iterate until an optimal strategy is found

Let's now understand Reinforcement Learning by actually developing an agent to learn to play a game automatically on its own.

Example Design: Self Driving Cab

Let's design a simulation of a self-driving cab. The major goal is to demonstrate, in a simplified environment, how you can use RL techniques to develop an efficient and safe approach for tackling this problem. The cabs job will be to pick up a passenger at one location and to drop them into another. In this quest, the cab should take care of few things:

1) Drop the passenger at the right location.
2) Save passengers time by taking the minimum amount of road.
3) Take care of the rules.

There are different aspects that need to be considered here while modeling an reinforcement learning solution to this problem: rewards, states, and actions.

1) Rewards:

Since the driver is reward-motivated, it is going to learn how to control the cab by trial experiences. In other words we need to decide the rewards and/or penalties due to it’s actions.

2) State

In Reinforcement Learning, the agent encounters a state, and then takes action according to the state it's in. The State Space is the set of all possible situations our taxi could inhabit. The state should contain useful information the agent needs to make the right action.

3) Action Space

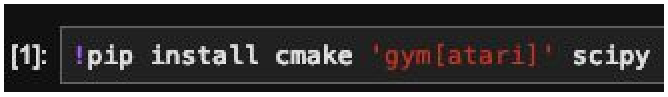
In our example, the cap will have 6 possible moves. South, north, east, west, pickup and dropoff.

Implementation With Python

While creating this program we will use an environment called OpenAI Gym which will help us to build a game environment and test the agent. The library takes care of API for providing all the information that our agent would require, like possible actions, score, and current state. We just need to focus just on the algorithm part for our agent. We'll be using the Gym environment called Taxi-V2, which all of the details explained above were pulled from. The objectives, rewards, and actions are all the same. In this tutuorial we will use Jupyter Notebook in order to run our example.

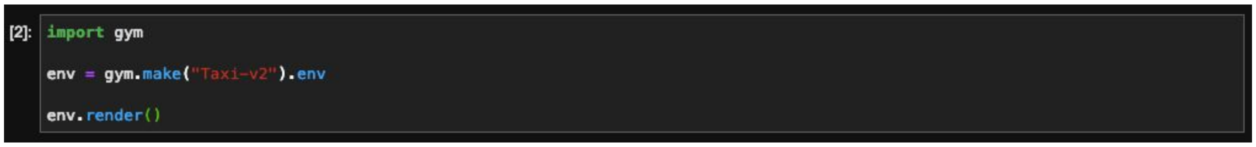
Gym's Interface

We need to install gym first. Executing the following in a Jupyter notebook should work.

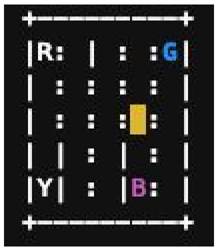


```
[1]: !pip install cmake 'gym[atari]' scipy
```

Once installed, we can load the game environment and render what it looks like:



```
[2]: import gym

     env = gym.make("Taxi-v2").env

     env.render()
```

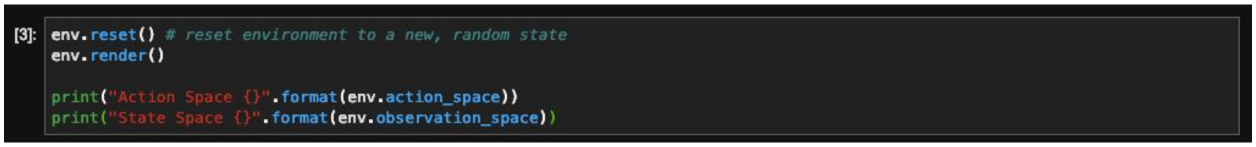


```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

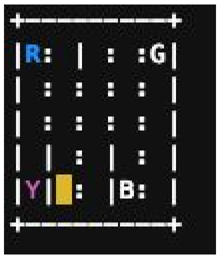
The most important interface of gym is env, which is an environment interface. The following are some of the important methods of env which will be quite helpfull to us:

1) env.reset → Resets the environment and returns a random initial state.
2) env.step(action) → Step the environment by one timestep. Returns
    a) observation: Observations of the environment
    b) reward: If your action was beneficial or not
    c) done: Indicates if we have successfully picked up and dropped off a passenger, also called one *episode*
    d) info: Additional info such as performance and latency for debugging purposes
3) env.render → Renders one frame of the environment (helpful in visualizing the environment)

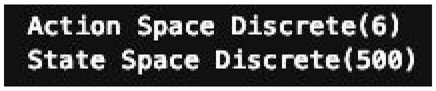
Lets dive more to the environment.



```
[3]: env.reset() # reset environment to a new, random state
     env.render()

     print("Action Space {}".format(env.action_space))
     print("State Space {}".format(env.observation_space))
```



```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```



```
Action Space Discrete(6)
State Space Discrete(500)
```

The filled square represents the taxi which is yellow without a passenger and green with a passenger. The pipe (|) represents a wall which the taxi cannot pass. R, G, Y and B are the possible locations for the taxi to drop or pickup. Finally while the blue letter represents the passengers pickup location, the purple letter represents the current location.As verified by the prints, we have an Action Space of size 6 and a State Space of size 500. As you'll see, our RL algorithm won't need any more information than these two things. All we need is a way to identify a state uniquely by assigning a unique number to every possible state, and RL learns to choose an action number from 0-5 where:

0 = south

1 = north

2 = east

3 = west

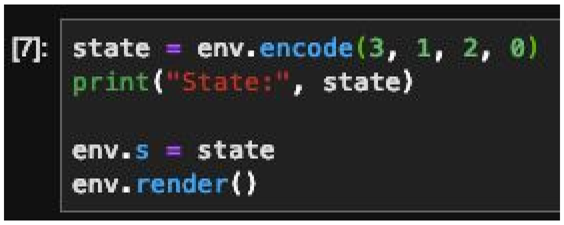
4 = pickup

5 = dropoff

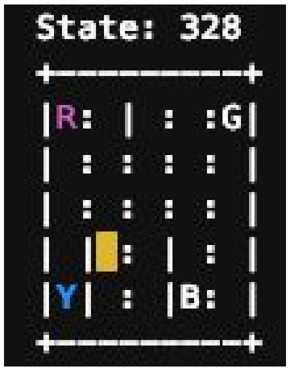
Recall that the 500 states correspond to a encoding of the taxi's location, the passenger's location, and the destination location. Reinforcement Learning will learn a mapping of states to the optimal action to perform in that state by *exploration*, i.e. the agent explores the environment and takes actions based off rewards defined in the environment. The optimal action for each state is the action that has the highest cumulative long-term reward.

Back To Our Illustration

We can actually take our illustration and encode its state. With the help of this we will relocate the taxi in our environment. Using the Taxi-v2 state encoding method, we can do the following.



```
[7]: state = env.encode(3, 1, 2, 0)
     print("State:", state)

     env.s = state
     env.render()
```

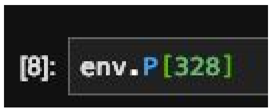


```
State: 328
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

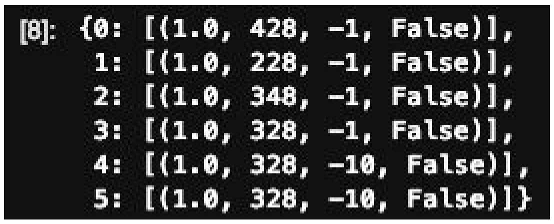
In this particular code we are using our illustrations coordinates in order to generate a number corresponding to a state between 0 and 499 which is 328 for our illustration.

The Reward Table

When the Taxi environment is created, a reward table called ‘P’ is also created. This reward table is like a matrix that has the number of states as raws and numbers of actions as columns. Since every state is in this matrix, we can see the default values of the rewards by using this code:



```
[8]: env.P[328]
```



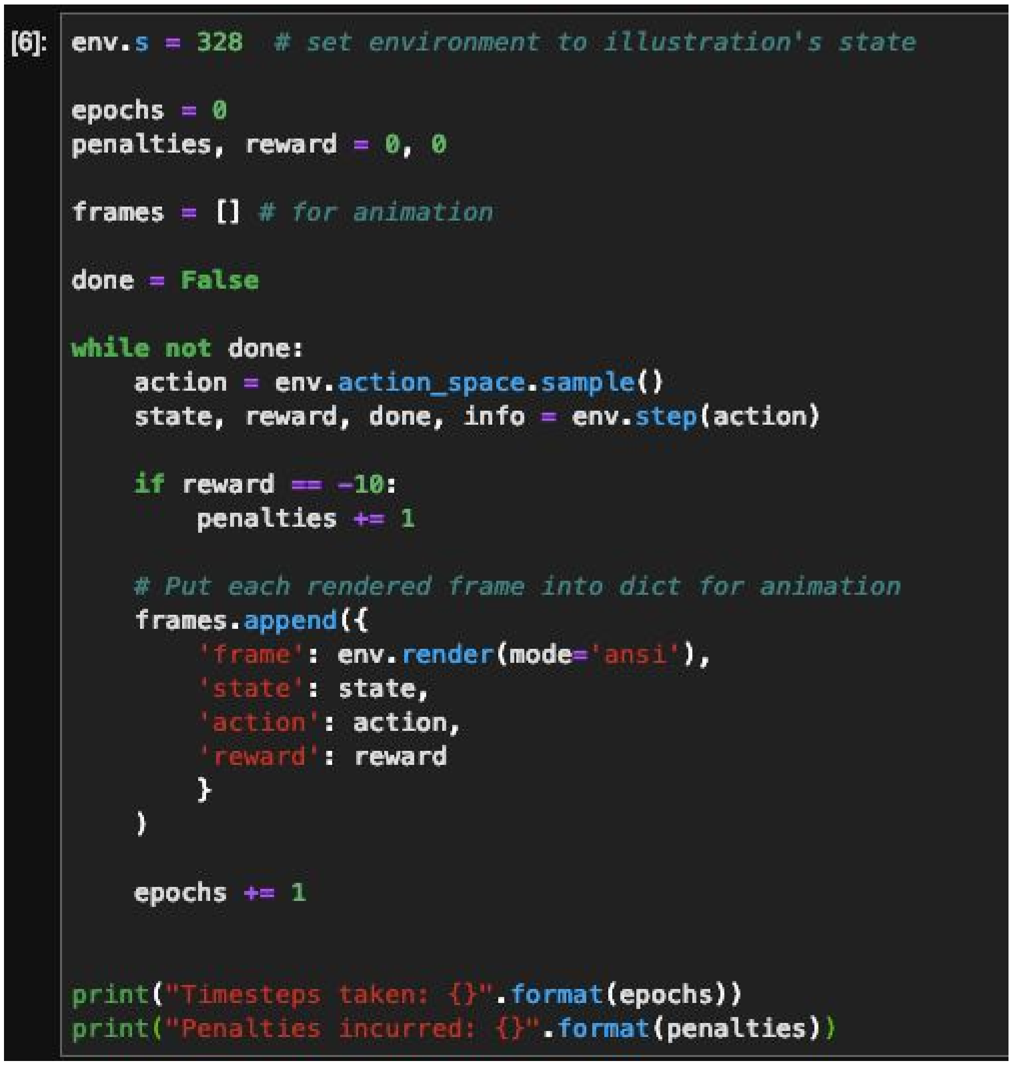
```
[8]: {0: [(1.0, 428, -1, False)],
     1: [(1.0, 228, -1, False)],
     2: [(1.0, 348, -1, False)],
     3: [(1.0, 328, -1, False)],
     4: [(1.0, 328, -10, False)],
     5: [(1.0, 328, -10, False)]}
```

The dictionary has the structure {action: [probability, nextstate, reward, done)]}. A few things to note:

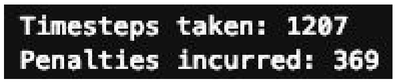
- The 0-5 corresponds to actions (south, north, east, west, pickup, dropoff) the taxi can perform at our current state in the illustrations.
- In this env, probability is always 1.0.
- The nextstate is the state we would be in if we take the action at this index of the dict
- All the movement actions have a -1 reward and the pickup/dropoff actions have -10 reward in this particular state. If we are in a state where the taxi has a passenger and is on top of the right destination, we would see a reward of 20 at the dropoff action (5)
- done is used to tell us when we have successfully dropped off a passenger in the right location. Each successfull dropoff is the end of an **episode.**

Solving The Problem Without Reinforcement Learning

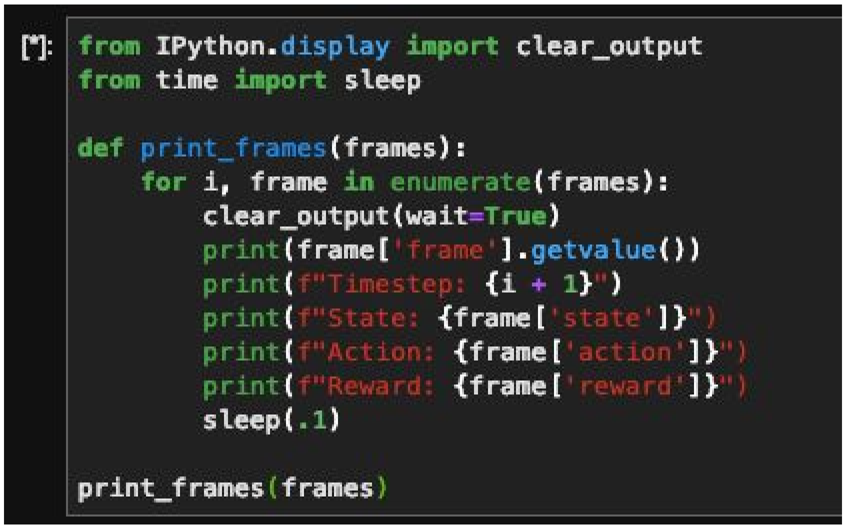
Let’s see what would happen if we try to brute-force our way to solving problem without Reinforcement Learning.Since we have our p table for default rewards in each state, we can try to have our taxi navigate just using that. We will create an infinite loop which runs until one passenger reaches one destination, or in other words, when it receive a reward of 20 points. The env.action_space.sample() method automatically selects one random action from set of all possible actions. Let’s see what happens:



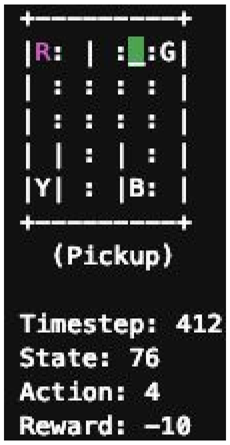
```
[6]: env.s = 328  # set environment to illustration's state

     epochs = 0
     penalties, reward = 0, 0

     frames = [] # for animation

     done = False

     while not done:
         action = env.action_space.sample()
         state, reward, done, info = env.step(action)

         if reward == -10:
             penalties += 1

         # Put each rendered frame into dict for animation
         frames.append({
             'frame': env.render(mode='ansi'),
             'state': state,
             'action': action,
             'reward': reward
             }
         )

         epochs += 1


     print("Timesteps taken: {}".format(epochs))
     print("Penalties incurred: {}".format(penalties))
```



```
Timesteps taken: 1207
Penalties incurred: 369
```



```
[*]: from IPython.display import clear_output
     from time import sleep

     def print_frames(frames):
         for i, frame in enumerate(frames):
             clear_output(wait=True)
             print(frame['frame'].getvalue())
             print(f"Timestep: {i + 1}")
             print(f"State: {frame['state']}")
             print(f"Action: {frame['action']}")
             print(f"Reward: {frame['reward']}")
             sleep(.1)

     print_frames(frames)
```



```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)

Timestep: 412
State: 76
Action: 4
Reward: -10
```

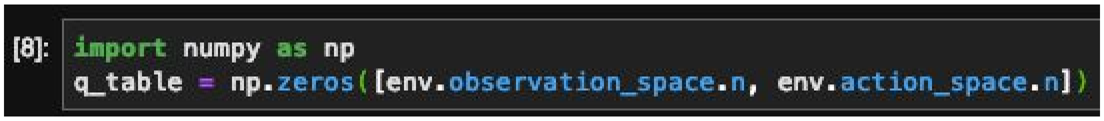
Not good. Our agent takes thousands of timesteps and makes lots of wrong drop offs to deliver just one passenger to the right destination. This is because we aren't *learning* from past experience. We can run this over and over, and it will never optimize. The agent has no memory of which action was best for each state, which is exactly what Reinforcement Learning will do for us.

Enter Reinforcement Learning

We are going to use a simple RL algorithm called Q-learning which will give our agent some memory. Basically, Q-learning lets the agent use the environment's rewarsa to learn over time, the best action to take in a given state. In our Taxi environment, we have the reward table, P, that the agent will learn from. It does thing by looking receiving a reward for taking an action in the current state, then updating a Q-value to remember if that action was beneficial.

Implementing Q-learning In Python

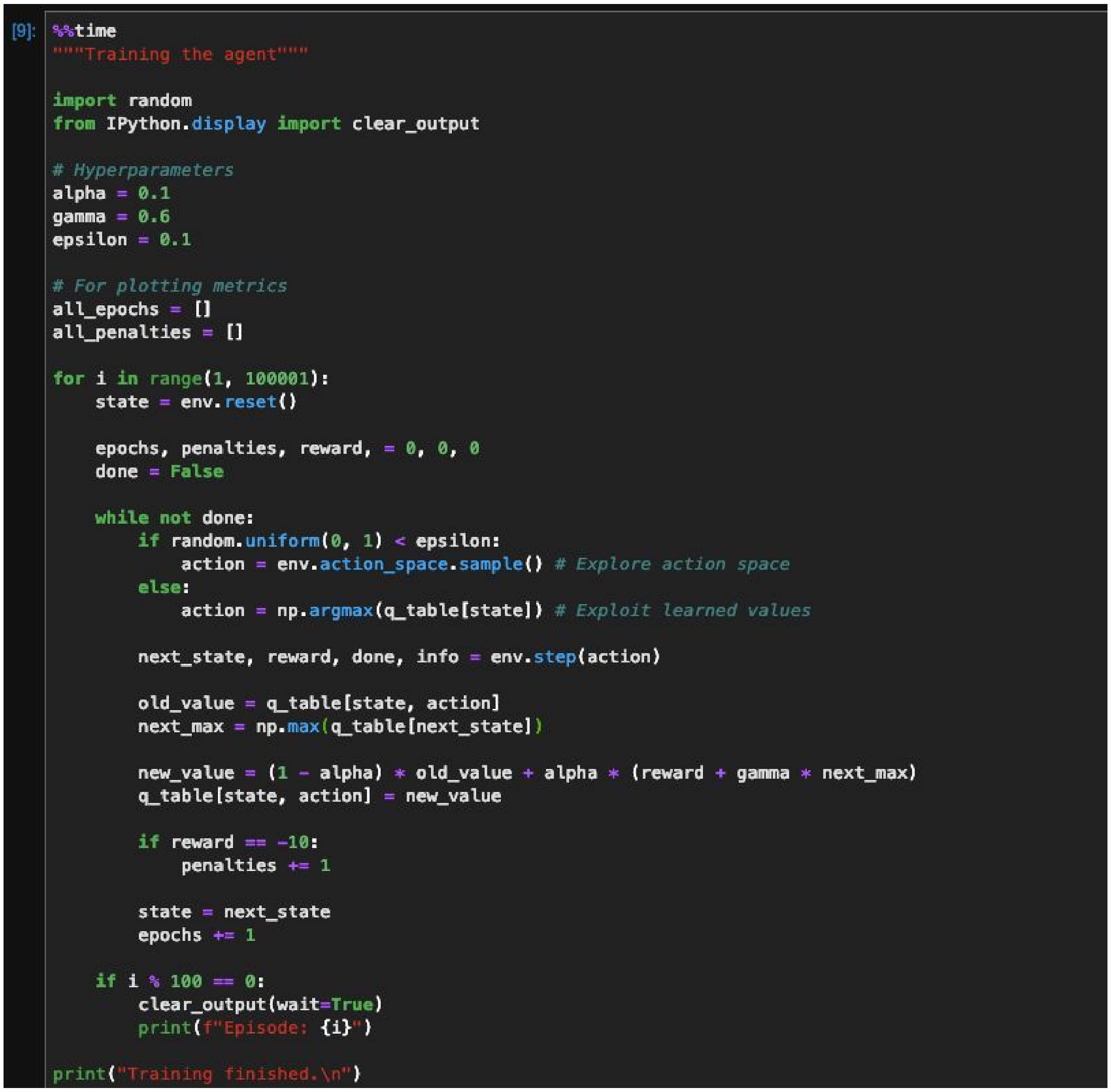
First we will initiliaze the Q-table to a 500x6 matrix of zeros.



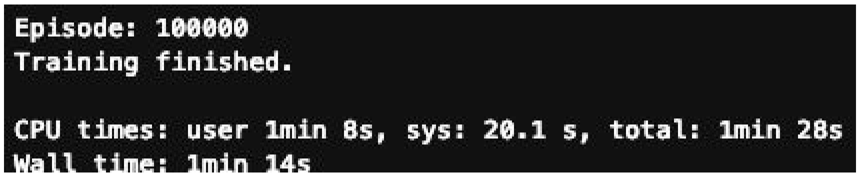
```
[8]: import numpy as np
     q_table = np.zeros([env.observation_space.n, env.action_space.n])
```

We can now create the training algorithm that will update this Q-table as the agent explores the environment over thousands of episodes. In the first part of while not done, we decide whether to pick a random action or to exploit the already computed Q-values. This is done simply by using the epsilon value and comparing it to the random.uniform(0,1) function, which returns an arbitrary number between 0 and 1.

We execute the chosen action in the environment to obtain the next_state and the reward from performing the action. After that, we calculate the maximum Q-value for the actions corresponding to the next_state, and with that, we can easily update our Q-value to the new_q_value:

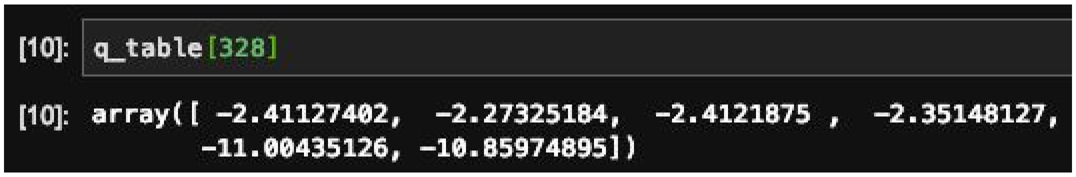


```
[9]: %%time
"""Training the agent"""

import random
from IPython.display import clear_output

# Hyperparameters
alpha = 0.1
gamma = 0.6
epsilon = 0.1

# For plotting metrics
all_epochs = []
all_penalties = []

for i in range(1, 100001):
    state = env.reset()

    epochs, penalties, reward, = 0, 0, 0
    done = False

    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Explore action space
        else:
            action = np.argmax(q_table[state]) # Exploit learned values

        next_state, reward, done, info = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])

        new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
        q_table[state, action] = new_value

        if reward == -10:
            penalties += 1

        state = next_state
        epochs += 1

    if i % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")

print("Training finished.\n")
```



```
Episode: 100000
Training finished.

CPU times: user 1min 8s, sys: 20.1 s, total: 1min 28s
Wall time: 1min 14s
```

Now that the Q-table has been established over 100000 episodes, let's see what the Q-values are at our illustration's state:

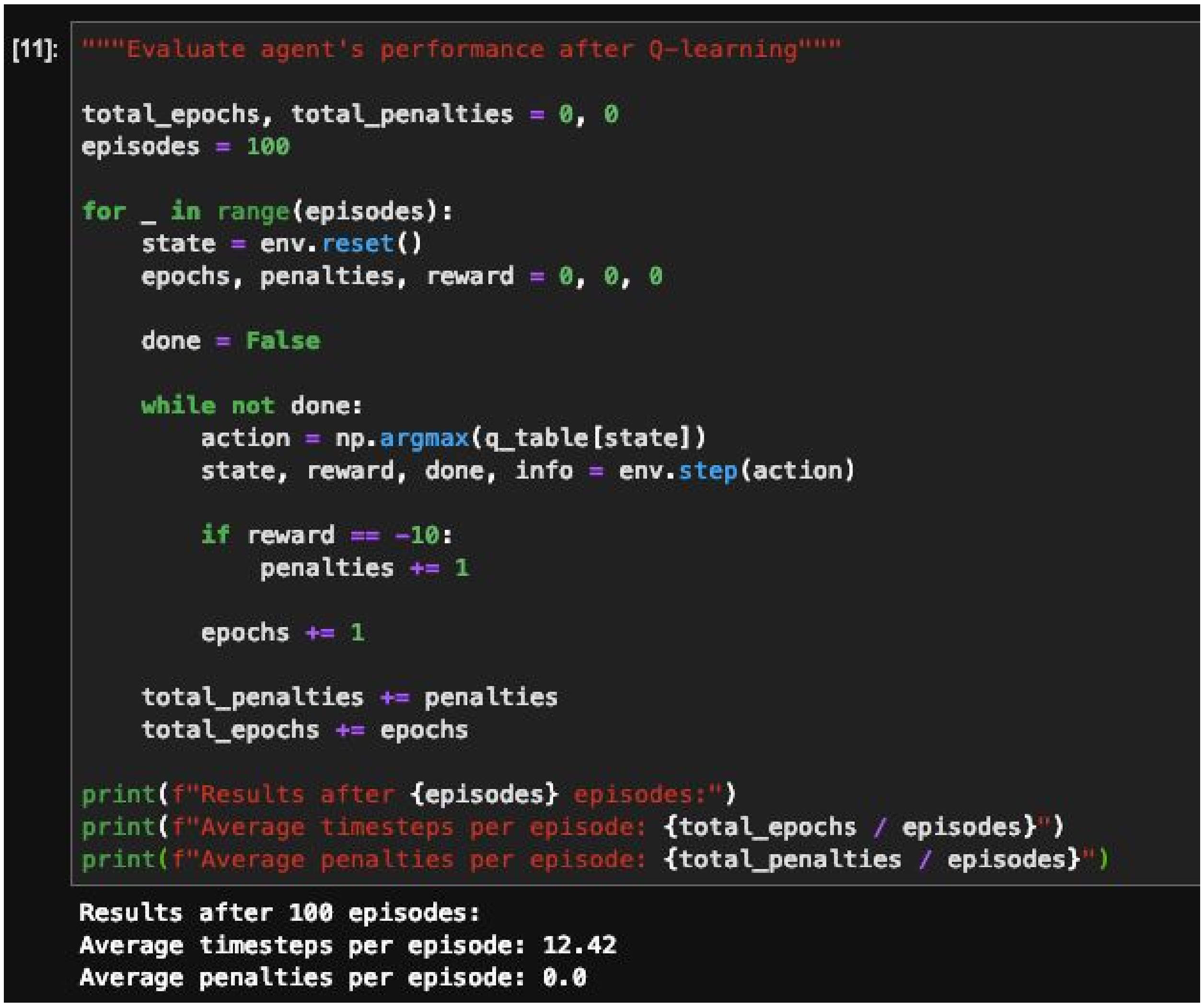


```
[10]: q_table[328]

[10]: array([ -2.41127402,  -2.27325184,  -2.4121875 ,  -2.35148127,
             -11.00435126, -10.85974895])
```

The max Q-value is "north" (-1.971), so it looks like Q-learning has effectively learned the best action to take in our illustration's state!

Evaluating The Agent

Let’s evaluate the performance of our agent. We don’t need to explore actions any further, so now the next action is always selected using the best Q-value:



```
[11]: """Evaluate agent's performance after Q-learning"""

total_epochs, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = False

    while not done:
        action = np.argmax(q_table[state])
        state, reward, done, info = env.step(action)

        if reward == -10:
            penalties += 1

        epochs += 1

    total_penalties += penalties
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")

Results after 100 episodes:
Average timesteps per episode: 12.42
Average penalties per episode: 0.0
```

We can see from the evaluation, the agent’s performance improved significantly and it incurred no penalties, which means it performed the correct pickup/dropoff actions with 100 different passengers.

3. Uninformed Search (Blind Search)

What is it?

“Uninformed Search” creates a search tree without having domain-specific knowledge. For the most part, these require brute force algorithms that try a bulk number of possibilities until it converges. It works by trying every possible node it comes across and trying to get to a solution.

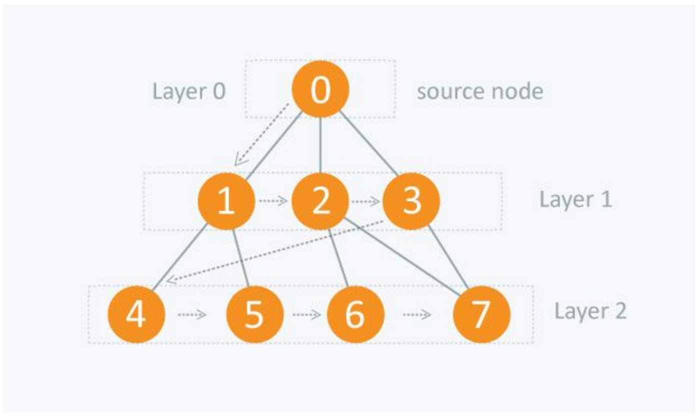
When is it used?

Suppose you’re playing a game (chess, checkers, GO). In these games, you have numerous moves you can make in every given state of the game that creates different outcomes. Every choice brings an entire new dynamic that can be used further into the game that may affect if you win or not. Uninformed Search algorithms work to solve this problem by analyzing every possible move that spawns from your actions and the opposing side’s actions. This method can also be used to test if our data would fit a hypothesis. Since it tries every single possibility to find a way to the “goal state”.

Types

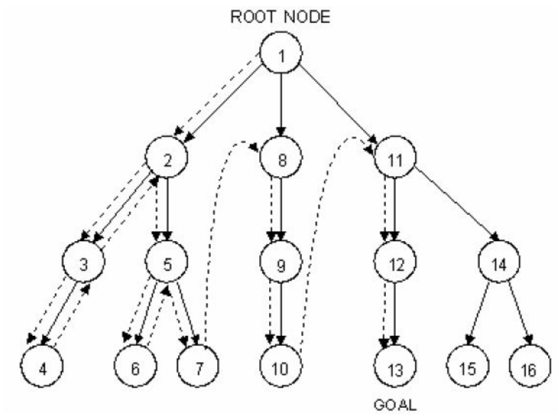
Breadth-First Search (BFS)

BFS analyzes trees from the root node to the neighboring nodes. After every neighboring node is analyzed, it then moves onto the next stage and so on. It tests the neighboring nodes from left to right and it analyzes every single node until there aren’t any levels to go down to left.



Depth-First Search (DFS)

DFS starts from the root node just as the BFS would, but instead of going horizontally, it goes vertically. What this means is that it follows the rightmost branches first, and after this path is done it moves onto the next rightmost path.



Depth Limited Search (DLS)

Follows the same method as the DFS, however it goes down until a certain depth at which it stops and goes back up to analyze the next set of branches, ignoring all the rest that are after the set limit.

Bidirectional Search

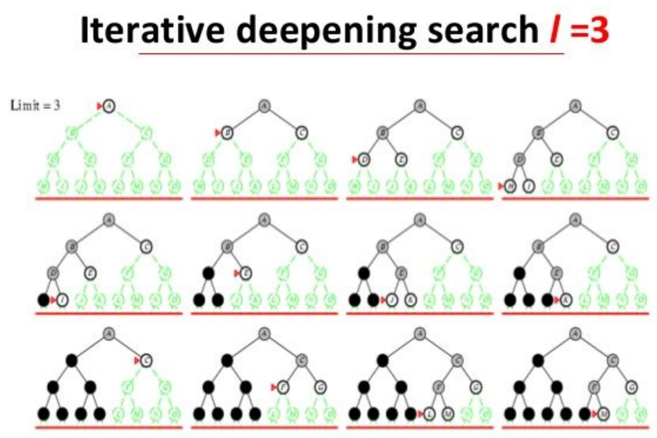
This algorithm searches possibilities from two directions: one from start to finish, and one from finish to start. Once both of these algorithms converge, we then finish the search. This is also a brute force algorithm, however it is relatively faster and requires less memory.

Uniform-Cost Search

It is almost like the BFS algorithm, however it chooses the nodes with the least weight. Since there are many different routes leading to an end result that may be more convoluted than if done with a simpler BFS algorithm. Since it chooses the path of least resistence in some sense, it finds the easiest way to get to a goal state, rather than the most effective.

IDDFS

IDDFS uses the two different methods from BFS and DFS to create a hybrid model that requires less memory.



Expert-Systems (Ivan Bratko)

What is it?

Expert systems are simply algorithms that act as an expert in a certain given application.

Örnekler:

1) Choosing economy or business class for upcoming flights
2) Analyzing problems with printing and paper for printing businesses

3) Searching for petroleum and mineral deposits
4) Helping with diagnosing illnesses

Özellikleri:

1) Solving problems in the area of expertise
2) Interacting with the user before and after the problem is solved to help with analysis.
3) Making use of Domain Knowledge
4) Explaining results to the user to aid humans
5) Analyzes data based on certain hypothesis and if and then rules
6) Can make analysis from end and from the beginning

Evolutionary Programming

1. History of Evolutionary Programming

It was originally conceived by Lawrence J. Fogel in 1960. Fogel was a computer researcher and is known as the father of evolutionary programming. From 1965 to 2007 he continued to apply methods of evolutionary programming to real-world problems in industry, medicine, and defense and helped organize conferences and publications in the areas of machine and human intelligence.

He did his phD at UCLA with the dissertation “On The Origin of Intellect”. His work that promoted evolutionary programming the most was his book "Artificial Intelligence Through Simulated Evolution" co-authored with Owens and Walsh.

2. Selection

Before we define evolutionary programming, we have to talk about a few concepts. Our first concept is selection.

If there is a pool of various individuals, those who are fit enough to copy themselves survive, if not, they extinguish. Reproducing by copy means that the fittest individuals populate the environment while the unfit eventually go extinct. But this only works if we have variety to start with.

Natural Selection happens by letting the individuals perform in an environment where they have to solve a problem. In this case, they have to “live” in an area where there task is to find ways to “survive”.

3. Evolution

We talked about reproducing by copying. But that does not mean the organism evolves. If an organism copies itself to reproduce, how can it evolve?

a. Mutation

One way is that it can mutate. It is a renewable source of variety. But it is dangerous and absolutely random therefore an effective but not very efficient way to evolve.

b. Cross-over

If different and already tested good treats could be shared it would be easier. This is, in very basic terms, sexual reproduction.

When an organism copies itself (if it can), a variety of offspring don’t form. However with sexual reproduction the genes crossover and a variety is formed in the genes. It is much safer and not so random, therefore more efficient than mutation. However, it does not provide renewable variety. Once all combinations have been produced, there will be no more variety. So we still NEED Mutation to maintain variety.

4. What Is Evolutionary Programming?

From observing Natural Selection and Evolution we can see that neither selection nor evolution is a solution to a problem. They provide a way to search for a solution by evolving it. Therefore Evolutionary Programming can be seen as a methodology for searching solutions rather than a solution in itself. • The solution is searched by trying it in the actual problem rather than trying to find the inverse model of the problem. It is a direct solving method rather than an inverse one.

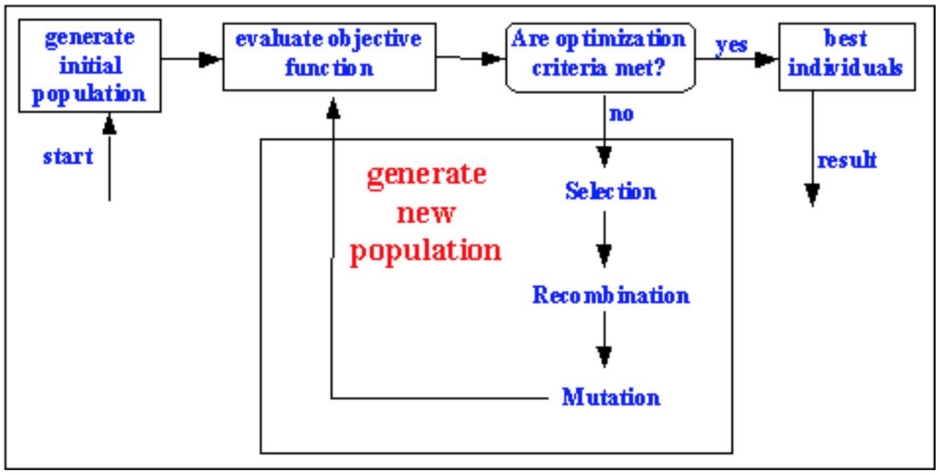
In other words, evolutionary programming is a stochastic optimization strategy similar to genetic algorithms, but instead places emphasis on the behavioral linkage between parents and their offspring.

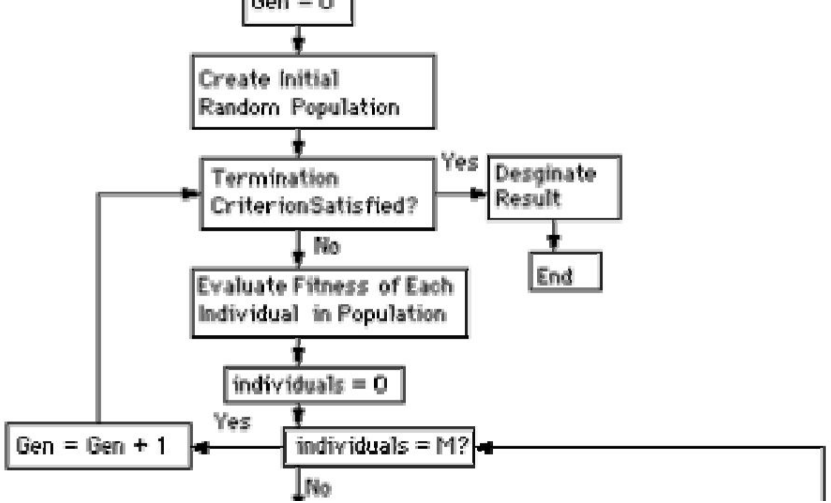
5. The Steps and Process of Evolutionary Programming

a. Choose an initial population of trial solutions at random. The number of solutions in a population is highly relevant to the speed of optimization, but no definite answers are available as to how many solutions are appropriate (other than >1) and how many solutions are just wasteful.

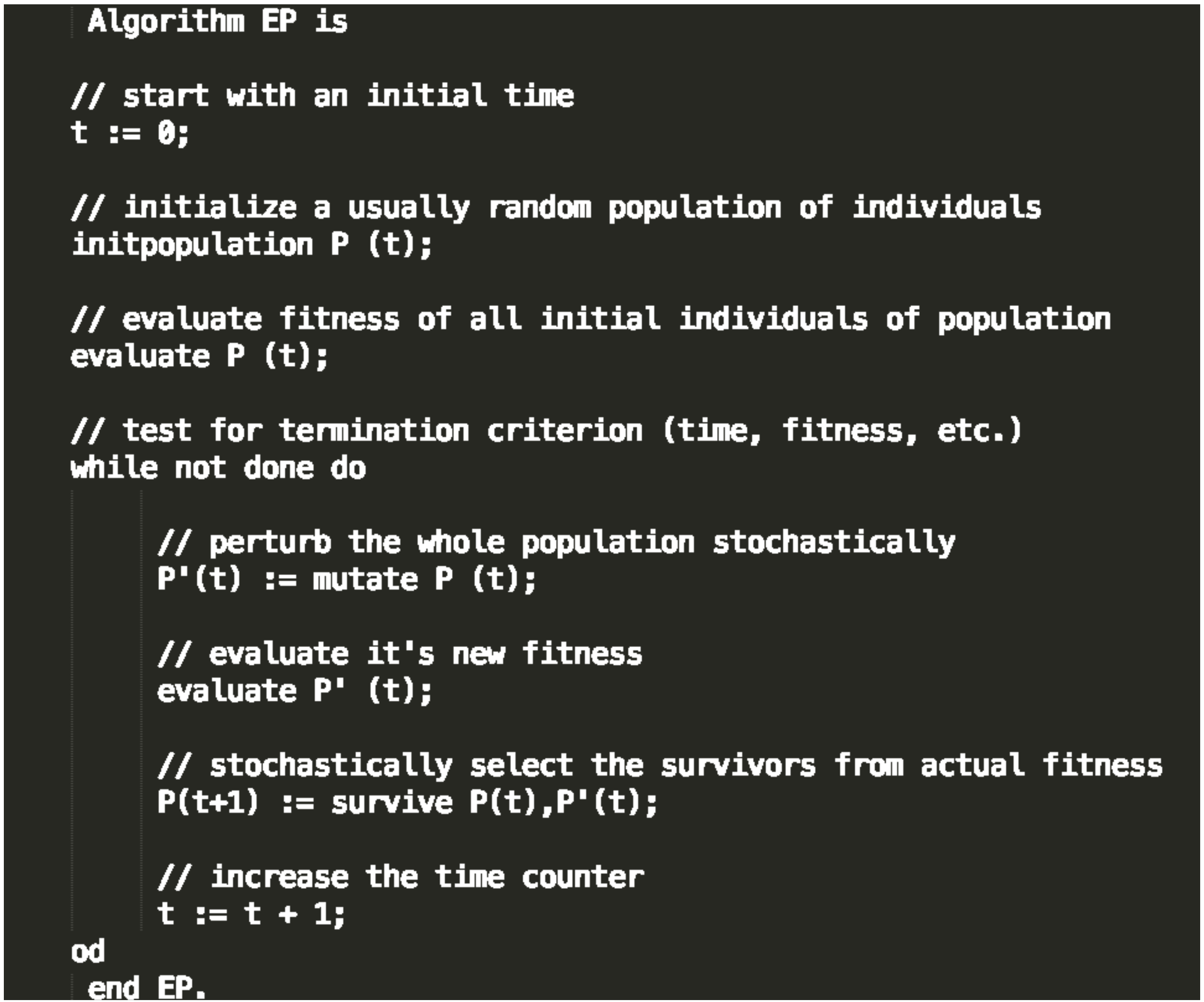
b. Each solution is replicated into a new population. Each of these offspring solutions are mutated according to a distribution of mutation types, ranging from minor to extreme with a continuum of mutation types between.

c. Each OFFSPRING solution is assessed by computing it's FITNESS.



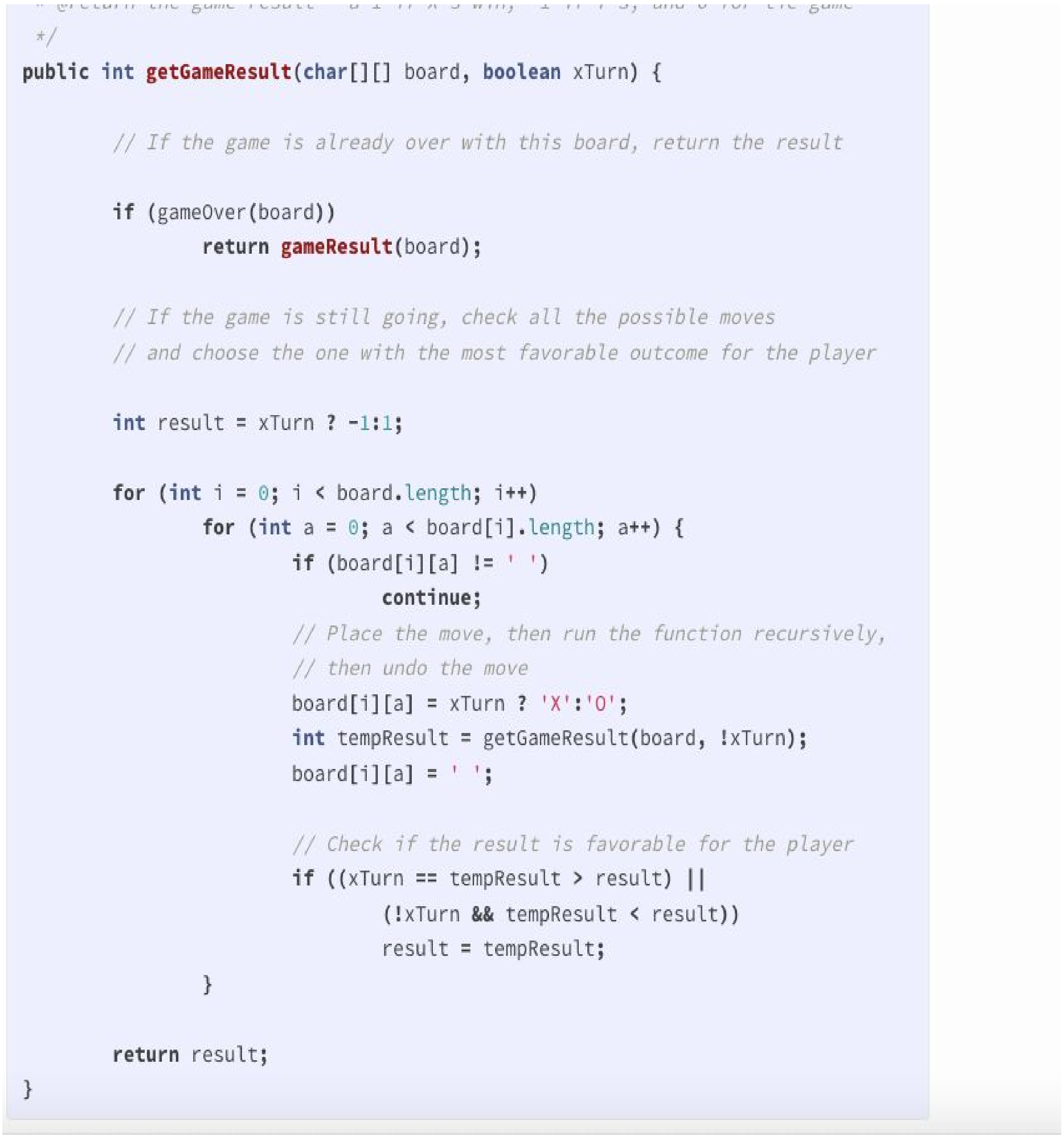

Gen = 0
Create Initial Random Population
Termination CriterionSatisfied?
Yes
Desginate Result
End
No
Evaluate Fitness of Each Individual in Population
individuals = 0
Yes
Gen = Gen + 1
individuals = M?
No

6. Sudo Kod



```
Algorithm EP is

// start with an initial time
t := 0;

// initialize a usually random population of individuals
initpopulation P (t);

// evaluate fitness of all initial individuals of population
evaluate P (t);

// test for termination criterion (time, fitness, etc.)
while not done do

     // perturb the whole population stochastically
     P'(t) := mutate P (t);

     // evaluate it's new fitness
     evaluate P' (t);

     // stochastically select the survivors from actual fitness
     P(t+1) := survive P(t),P'(t);

     // increase the time counter
     t := t + 1;
od
 end EP.
```

Uninformed Search Örneği (Tic tac Toe)



```
 * @return the game result - a 1 if X's win, -1 if Y's, and 0 for tie game
 */
public int getGameResult(char[][] board, boolean xTurn) {

        // If the game is already over with this board, return the result

        if (gameOver(board))
                return gameResult(board);

        // If the game is still going, check all the possible moves
        // and choose the one with the most favorable outcome for the player

        int result = xTurn ? -1:1;

        for (int i = 0; i < board.length; i++)
                for (int a = 0; a < board[i].length; a++) {
                        if (board[i][a] != ' ')
                                continue;
                        // Place the move, then run the function recursively,
                        // then undo the move
                        board[i][a] = xTurn ? 'X':'O';
                        int tempResult = getGameResult(board, !xTurn);
                        board[i][a] = ' ';

                        // Check if the result is favorable for the player
                        if ((xTurn == tempResult > result) ||
                                (!xTurn && tempResult < result))
                                result = tempResult;
                }

        return result;
}
```

Bu Kodun Özellikleri

1. Oyun bitti mi bakar— Bütün kutular dolu ise veya bir oyuncu kazandıysa sonucu verir
2. Bütün karelere bakar
    - Bir kutu dolu ise sonraki kutuya geçer

    - Oyuncunun rengine bağlı olarak “x” veya “o” işareti yerleştirir
    - Dalı update ederek oyuncu için ideal seneryoyu bulmayı dener

Bu kodun açıklaması

Ideal bir şekilde oynandığı var sayılınca bir oyunu kimin kazanacağını veren bir koddur. Bu tarz bir algoritma aslında bir tree yaratmadığı için daha kolaydır. Bu kod yukarıdan aşağıya doğru bütün nodları geri dönmeden kontrol ettiğinden aslında bir DFS örneğidir. Genel olarak başa dönen bir yapıya sahip olduğundan *recursive* bir algoritmadır.

Heuristic Search

A *heuristic* is a method that

- might not always find the best solution
- *but* is guaranteed to find a good solution in reasonable time.
- By sacrificing completeness it increases efficiency.
- Useful in solving tough problems which
    - could not be solved any other way.
    - solutions take an infinite time or very long time to compute.

The *classic* example of heuristic search methods is the travelling salesman problem.
Heuristic Search methods Generate and Test Algorithm

1. generate a possible solution which can either be a point in the problem space or a path from the initial state.
2. test to see if this possible solution is a real solution by comparing the state reached with the set of goal states.
3. if it is a real solution, return. Otherwise repeat from 1.

This method is basically a depth first search as complete solutions must be created before testing. It is often called the British Museum method as it is like looking for an exhibit at random. A heuristic is needed to sharpen up the search. Consider the problem of four 6-sided cubes, and each side of the cube is painted in one of four colours. The four cubes are placed next to one another and the problem lies in arranging them so that the four available colours are displayed whichever way the 4 cubes are viewed. The problem can only be solved if there are at least four sides coloured in each colour and the number of options tested can be reduced using heuristics if the most popular colour is hidden by the adjacent cube.

Hill climbing

Here the generate and test method is augmented by an heuristic function which measures the closeness of the current state to the goal state.

1. Evaluate the initial state if it is goal state quit otherwise current state is initial state.
2. Select a new operator for this state and generate a new state.
3. Evaluate the new state
    - if it is closer to goal state than current state make it current state
    - if it is no better ignore
4. If the current state is goal state or no new operators available, quit. Otherwise repeat from 2.

In the case of the four cubes a suitable heuristic is the sum of the number of different colours on each of the four sides, and the goal state is 16 four on each side. The set of rules is simply choose a cube and rotate the cube through 90 degrees. The starting arrangement can either be specified or is at random.

SIMULATED ANNEALING

This is a variation on hill climbing and the idea is to include a general survey of the scene to avoid climbing false foot hills.
The whole space is explored initially and this avoids the danger of being caught on a plateau or ridge and makes the procedure less sensitive to the starting point. There are two additional changes; we go for minimisation rather than creating maxima and we use the term objective function rather than heuristic. It becomes clear that we are valley descending rather than hill climbing. The title comes from the metallurgical process of heating metals and then letting them cool until they reach a minimal $p = \mathrm{cxp}^{e-\Delta E/kT}$ energy steady final state. The probability that the metal will jump to a higher energy level is given by where *k* is Boltzmann's constant. The rate at which the system is cooled is called the annealing schedule. ΔE is called the change in the value of the objective function and *kT* is called *T* a type of temperature. An example of a problem suitable for such an algorithm is the travelling salesman. The SIMULATED ANNEALING algorithm is based upon the physical process which occurs in metallurgy where metals are heated to high temperatures and are then cooled. The rate of cooling clearly affects the finished product. If the rate of cooling is fast, such as when the metal is quenched in a large tank of water the structure at high temperatures persists at low temperature and large crystal structures exist, which in this case is equivalent to a local maximum. On the other hand if the rate of cooling is slow as in an air based method then a more uniform crystalline structure exists equivalent to a global maximum.The probability of making a large uphill move is lower than a small move and the probability of making large moves decreases with temperature. Downward moves are allowed at any time.

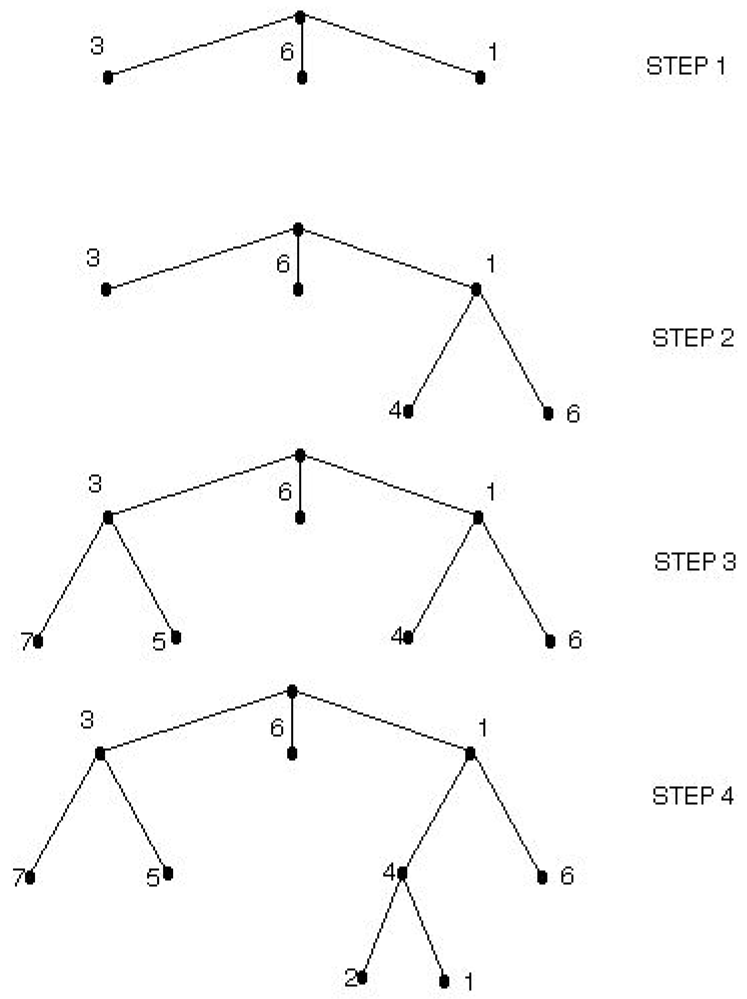
1. Evaluate the initial state.
2. If it is goal state Then quit otherwise make the current state this initial state and proceed.
3. Make variable *BEST_STATE* to current state
4. Set temperature, *T*, according to the annealing schedule
5. Repeat
    1. ΔE -- difference between the values of current and new states of this new state is goal state Then quit
    2. Otherwise compare with the current state
    3. If better set *BEST_STATE* to the value of this state and make the current the new state
    4. If it is not better then make it the current state with probability *p*'. This involves generating a random number in the range 0 to 1 and comparing it with a half, if it is less than a half do nothing and if it is greater than a half accept this state as the next current be a half.
    5. Revise T in the annealing schedule dependent on number of nodes in tree
6. Until a solution is found or no more new operators
7. Return *BEST_STATE* as the answer

Best First Search

A combination of depth first and breadth first searches.
Depth first is good because a solution can be found without computing all nodes and breadth first is good because it does not get trapped in dead ends. The best first search allows us to switch between paths thus gaining the benefit of both approaches. At each step the most promising node is chosen. If one of the nodes chosen generates nodes that are less promising it is possible to choose another at the same level and in effect the search changes from depth to breadth. If on analysis these are no better then this previously unexpanded node and branch is not forgotten and the search method reverts to the descendants of the first choice and proceeds, backtracking as it were.
This process is very similar to steepest ascent, but in hill climbing once a move is chosen and the others rejected the others are never reconsidered whilst in best first they are saved to enable revisits if an impasse occurs on the apparent best path. Also the best available state is selected in best first even its value is worse than the value of the node just explored whereas in hill climbing the progress stops if there are no better successor nodes. The best first search algorithm will involve an OR graph which avoids the problem of node duplication and assumes that each node has a parent link to give the best node from which it came and a link to all its successors. In this way if a better node is found this path can be propagated down to the successors. This method of using an OR graph requires 2 lists of nodes
OPEN is a priority queue of nodes that have been evaluated by the heuristic function but which have not yet been expanded into successors. The most promising nodes are at the front. CLOSED are nodes that have already been generated and these nodes must be stored because a graph is being used in preference to a tree.

Heuristics In order to find the most promising nodes a heuristic function is needed called *f*' where *f*' is an approximation to *f* and is made up of two parts *g* and *h*' where *g* is the cost of going from the initial state to the current node; *g* is considered simply in this context to be the number of arcs traversed each of which is treated as being of unit weight. *h*' is an estimate of the initial cost of getting from the current node to the goal state. The function *f*' is the approximate value or estimate of getting from the initial state to the goal state. Both *g* and *h*' are positive valued variables. Best First The Best First algorithm is a simplified form of the *A** algorithm. From*A** we note that *f*' = *g*+*h*' where *g* is a measure of the time taken to go from the initial node to the current node and *h*' is an estimate of the time taken to solution from the current node. Thus *f*' is an estimate of how long it takes to go from the initial node to the solution. As an aid we take the time to go from one node to the next to be a constant at 1.
Best First Search Algorithm:

1. Start with OPEN holding the initial state
2. Pick the best node on OPEN
3. Generate its successors
4. For each successor Do
    - If it has not been generated before evaluate it add it to OPEN and record its parent
    - If it has been generated before change the parent if this new path is better and in that case update the cost of getting to any successor nodes
5. If a goal is found or no more nodes left in OPEN, quit, else return to 2.



All figures indicate "cost" of move

The *A** Algorithm
Best first search is a simplified *A**.

1. Start with *OPEN* holding the initial nodes.
2. Pick the *BEST* node on *OPEN* such that $f = g + h'$ is minimal.
3. If *BEST* is goal node quit and return the path from initial to *BEST*

   Otherwise

4. Remove *BEST* from *OPEN* and all of *BEST*'s children, labelling each with its path from initial node.

Graceful decay of admissibility
If *h*' rarely overestimates h by more than d then the *A** algorithm will rarely find a solution whose cost is *d* greater than the optimal solution.

Adversarial Search

Adversarial search, also known as Minimax search is known for its usefulness in calculating the best move in two player games where all the information is available, such as chess or tic tac toe. It consists of navigating through a tree which captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players.

It follows that this can only be used to make decisions in zero-sum games, where one player's loss is the other player's gain. Theoretically, this search algorithm is based on von Neumann's minimax theorem which states that in these types of games there is always a set of strategies which leads to both players gaining the same value and that seeing as this is the best possible value one can expect to gain, one should employ this set of strategies.

Von Neumann's Minimax Theorem:

Theorem: Let A be a $m \times n$ matrix representing the payoff matrix for a two-person, zerosum game. Then the game has a value and there exists a pair of mixed strategies which are optimal for the two players.

Alpha - Beta Pruning

Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let's define the parameters alpha and beta.

Alpha is the best value that the maximizer currently can guarantee at that level or above.

Beta is the best value that the minimizer currently can guarantee at that level or above.

Pseudocode:

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node
    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal

// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```

By: Efe Altan 12C

Resources:

- http://users.cs.cf.ac.uk/Dave.Marshall/AI2/node23.html
- http://www.math.udel.edu/~angell/minimax
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Contributions

Introduction to AI (Emre Can Aydoğmuş)
Uninformed Search (Selim Bilgin)
Heuristic Search (Efe Altan)
Machine Learning (Emre Can Aydoğmuş)
Evolutionary Programming (teoman Özkan)
Reinforcement Learning (Ali Çağatay)
AI Applications NLP (Doruk Dölen, Can Parlar)
AI Applications (Sinan Tuna)
Essential Math (?)