Big Data Analytics Experience
Master's in Information Systems

Final Report

July 23rd, 2019

Sushmita Saxena
Satwik Karri
Saketh Valiveti

Contents

Data Sourcing

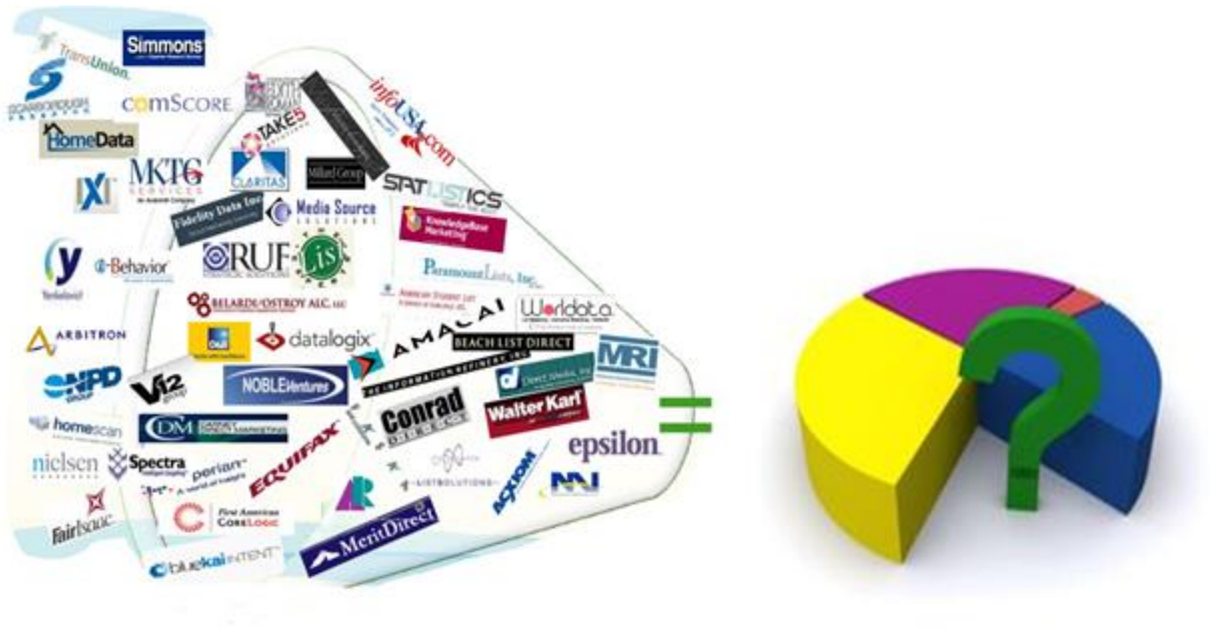
Data sourcing for Reinforcement learning (RL):

This technique implementation utilizes input data as sample sequences that consist of states, rewards, and actions. Based on such training examples, it allows the RL agent, to learn an optimal policy which defines the best possible action in each and every state. So, it is evident that there is no input data required to develop a reinforcement learning.

How Data Sourcing is done in the field of Data Science to help businesses:

Goal: Determining the most valuable mix of information, from across the hundreds of data providers covering the U.S. Consumer and Business populations is the goal.

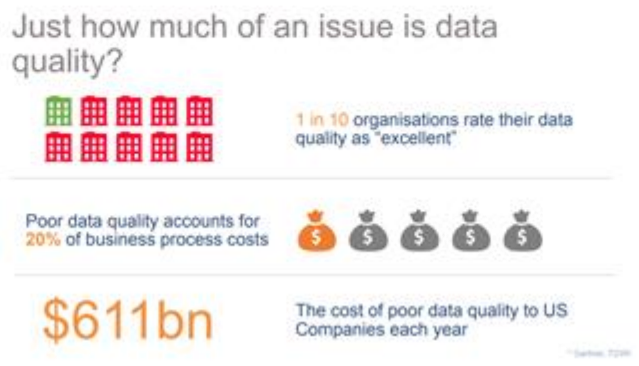
Cutting Through The 'Noise': What is the optimum mix of different data sources is a very costly question to be thought during data sourcing design strategy.



Even though there are various options available for data vendors, more time must be spent considering data vendors and to weight the differences, to be able to better select one that works best for a company and their initiatives. The following factors can be considered to decide:

Value of data: Quality over Quantity

The common issue when sourcing new data is the issue of quality versus quantity. Several businesses pick the cheapest data vendor that is available. It would be prudent to make a worthwhile that can bring good results. For example, 2,000 contacts for $200 sounds like a good deal, the contacts are not the best fit for customers or potential customers and might contain outdated information. However, if you were to spend $2,000 on 900 contacts, you might receive a robust contact profile, which essentially results in yielding a higher response rate from campaigns. Investing so much time and money for building the perfect campaign, would need to invest as much money and time to make sure the audience is right so that you can see the results you want and deserve. This would not be a quick and easy process, because results need to be analyzed over time. Spending ample time on examining the data source is a MAJOR aspect of the data sourcing.



Validation of Data

In case of any investment would require to test the waters and to get the right feel before any commitment.; data should not be different. Vendors will provide a sample contact list to check how accurate their data is, it is also important to see where your potential return on investment can go with their data and to analyze every aspect of information. For instance, the marketing department must test the email addresses , sales must test phone numbers to check if they are company HQ numbers or if they work. When the test has been performed, an analysis of the results should highlight the best choice in providers for your company.

Frequency: One-time vs Recurring

Often businesses will choose to purchase data for a one-time project. Once the project is over, then ultimately the data expires and isn't usable anymore. So, when the time arrives and when you're under pressure to get more leads for the next project, the cycle occurs again. Business is forced to buy new data, overuse it, trash it, then re-purchase most of the same data. It is not feasible to purchase one-time data when you could refresh the original data, adding more records. There can be a frequency of data pulling from the vendor, which can be agreed upon, and some data vendors on the market might offer data cleansing services, to continually make sure your data is cleansed and updated. If businesses are going to invest in marketing campaigns, they should ensure that they can get the most out of the information they already have at your fingertips and then look for options to buy data from second- or third-party vendors. This would be an aspect of data sourcing that is overlooked but can also save a company hundreds or thousands of dollars from having to get new data every time they run a new campaign.

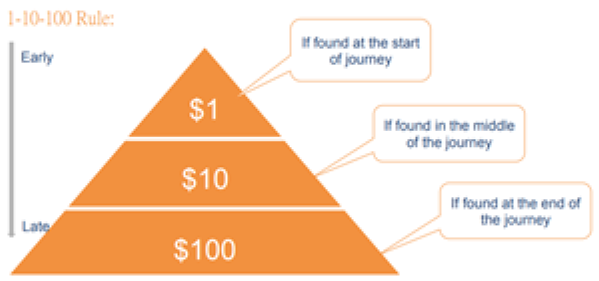
Some ideas we have for better Data Sourcing:

1. Profiling your data –
   • Structure, Format, Frequency, Age, Delivery Method.
   • Communicate it to data providers.
   • Identify issues and gaps.
   `

2. Getting as close to the source as possible- Be mindful of using a manually prepared report as a data source.
   • Look for opportunities to utilize the inputs into the report or spreadsheet rather than the outputs.
   • Collect the data from the earliest possible point of the data supply chain. This might limit exposure to uncontrolled manual intervention.

3. Streamline data sources and keeping it simple
   • Identify redundant data
   • Focus on the essentials
   • Cut out the stuff you don't need

4. Catching issues early in the data journey


1-10-100 Rule:
Early
Late
$1
$10
$100
If found at the start of journey
If found in the middle of the journey
If found at the end of the journey

5. Measuring and acting on data quality issues
   • Define metrics for your data quality
   • Measure for quality consistently
   • Address consistent concerns with strategic solutions (e.g. data cleansing)

Data Cleaning

Data cleaning for Reinforcement learning:

In our project the pre-processing of data includes reducing unwanted noise during training. There are few features which are present in the game's source code, but we are not considering them. For example, the original game has features like variable speed of the agent, some visual elements like clouds on the screen, variation in cactus, we can eliminate such features in the project to avoid complexity. Moreover, are reducing the resolution of the image as the regular CPU would not be able to handle high pixels of images. To do this we are using OpenCV library which would resize, crop and process the image to a lower dimension and single channel image.

How Data Cleaning is done in the field of Data Science to help businesses:

Successful analysis relies on accurate and well-structured data that has been formatted to the specific needs and outputs prepared data which can be utilized for analysis and various other business purposes. With the help of machine learning technologies, we can easily find patterns in the data and make clusters of clean and messy data. As we are using Python in our project, below are the steps used to clean data using python. Data cleaning depends on what kind of data is used for the model. However, we have tried to consolidate few steps which provides a systematic approach in the pre-processing of data.

1. Remove unwanted observations and fix Structural errors The first step to data cleaning is removing unwanted observations from your dataset which includes removing duplicates and irrelevant observations. Duplicate observations usually arise during data collection whereas irrelevant observations are those that do not actually fit the specific problem. In most of the models this is the first step, however, as per the study by Hamid Haidarian Shahri, duplicate elimination can be done due to discrepancies in spelling errors; abbreviations; missing fields; inconsistent formats; invalid, wrong, or unknown codes; word transposition; and such issues can occur multiple times and in any stage of the data cleaning process which makes it essential that the duplicate elimination must be performed after all stages.

2. Filter unwanted outliers Outliers are the tricky one, and it is important to analyze whether to remove outliers. For instance, outliers cannot be removed just because it's a big number because that big number could be very informative for the model. However, if there is a legitimate reason to remove the outliers then this would help improve the model's performance.

3. Handle missing data This is one of the most important stages while preprocessing the data. The most common approach of dealing with missing values is either to drop the observations that have missing values or to impute the missing values depending on the observations. Dropping missing values is suboptimal because when you drop observations, you drop information. Another approach of Imputing missing values is suboptimal because the value was originally missing, but you filled it in, which always lead to a loss in information, no matter how sophisticated your imputation method is. The methods might include replacing the null observations with mean

or the maximum occurrence of the value in a column, or any other imputation method. In real time situations missingness is informative which cannot be ignored. Hence, to deal with this the best way to handle missing data for categorical features is to label them as ’Missing’! This can be done by adding a new feature class. This tells the algorithm that the value was missing and gets around the technical requirement for no missing values. In case of missing numeric data, we can flag and fill the values. For this, flag the observation with an indicator variable of missingness and then, fill the original missing value with 0 just to meet the technical requirement of no missing values. Date cleaning is a crucial process for the success of machine learning algorithms. For most machine learning projects, about 80 percent of the effort is spent on data cleaning. The above-mentioned points would help a lot in refining the dataset and making the machine learning dataset error-proof.

ETL

ETL stands for Extract, Transform and Load. In the data field, ETL refers to the process of extracting, transforming and loading data into the format required. It is a necessary process for optimizing raw data for analytics. Before jumping to conclusions on how ETL adheres to a reinforcement learning project, it is important to understand the core concepts of this process as it lays the foundation on which our analytics runs on.

Extract:

It is the process of collecting desired data from data sources. There are numerous types of data sources ranging from a traditional relational database management system to application program interfaces (API's) to multiple API's sharing data files. There can be unstructured, semi-structured or unstructured data coming in from the data source and sometimes at varied frequencies as well. This means that the extract part of ETL needs to be extremely flexible, resilient and malleable to be able to support diverse data sources and variations in the extraction process.

Transform:

As the name suggests, it is the process of "transforming" raw data to the form where it is optimized to solve the problem at hand or, for a company with various business units, optimized to help solve problems in different types of scenarios. It can be considered as the least changed of the three processes with advances made only to make it more efficient, stable, and reliable.

Three things must be considered when transforming data, they are

1. Data Quality:

It is important to determine and qualify data received as high quality, complete, and acceptable. The system needs to ensure the data points are complete, adhere to the schema, and does not contain data that is not readable or corrupted and incoherent. Another important part of this is to check for any pattern changes in the data compared to previous data. If found, this must be inspected and rectify errors if any.

2. Business Logic: The data needs to be modeled in the way that it suites the business purpose of the analysis.

3. Business Quality: Based off business logic, the transformed data must meet the company's business quality standards for the intended analysis.

Load:

In our opinion, his is the simplest to define but most complex to apply process of the three in ETL. It has gone through major changes with the arrival of varied cloud storage. In the contemporary data architectures, it is important that the system can simultaneously stream and load data into multiple stacks without impacting the performance of the possible parallel loads. One of the best examples can be the Apache Hadoop architecture that is highly efficient in solving problems involving enormous amounts of data and computation while providing the framework for distributed storage and simultaneous processing of big data.

Now back to our application running Dino Run game with Reinforcement Learning the usage of ETL in this project is peculiar in the sense that, unless particularly specified in the Python code, the program does not return the Q-values, so one might think that we have no ETL. However, that is not true.

In our project, we are coding all the components in Python and obtaining the flow of the environment directly from the Chrome browser using an interface called Selenium. In terms of extracting data, we are giving the interpreter images of the game using Python libraries called PIL () and OpenCV. Essentially, we are recording the movement of the game and feeding this to the Python code which would then take actions based on what is on the screen. In this extraction process, we find that there are a few redundant features in the game such as the high score tracker and decided to remove them to improve performance of the model.

The amount of CPU/GPU power we had to process these images was limited, we made the decision to transform the original image size by cutting down the size of the images taken from 1200x300 pixels with multiple channels to a comfortable 40x20 pixels and single channel but use 4 continuous screenshots as a single input to the model to maintain a smooth flow of input to the interpreter. For this, we chose to go with AWS EC2.

Loading the data is quite an issue for a reinforcement learning model. This is because of the inherent qualities of such a model as it must train over a massive number of cycles to attain the high level of accuracy that we eventually see from the model. The complex part in this is that in a Q-learning off-policy model, we want the model to push for the larger goal over the period of
running in the environment rather than reap as much rewards as soon as possible. Controlling this randomness is totally different issue which will be discussed later in the report, but this vision of "long-term benefit" requires the model to retain most, if not all, of the states and actions previously made to ensure decisions in the present and the future are reliant on previously made actions. In addition, recording of the game while simultaneously training and running the model requires a considerably higher storage, and retrieval rate than the common personal computer can offer. To assist in this operation, we chose to go with the free tier model of AWS S3.

Storage

Amazon S3

“AWS has been the market share leader in cloud IaaS for over 10 years.”

Amazon Simple Storage Service - Amazon S3 is an object storage service that can protect any amount of data at a range of use cases, such as mobile applications, websites, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides easy-to-use management features so that we can organize our data and configure finely-tuned access controls to meet the project requirements.

In our project “Dino Run,” we are using Amazon S3

(for better computational power as i7 CPU with no GPU would possibly not be able to handle the size and play the game simultaneously). We are using AWS Free Usage Tier with which we can receive 5GB of Amazon S3 storage in the S3 Standard storage class; 20,000 GET Requests; 2,000 PUT, COPY, POST, or LIST Requests; and 15GB of Data Transfer Out. The reason for choosing Amazon S3 over Windows Azure Storage or Google Cloud Storage is its dominance in the public cloud market. As per Gartner report, AWS is the most mature, enterprise-ready provider, with the deepest capabilities for governing many users and resources. Also, AWS provides an extensive training documents which was also one of the key reasons for selecting S3 storage in our project. Moreover, we can configure and manage S3 buckets; which means we can use an Amazon S3 API to upload objects to a bucket.

Coming to our project of Reinforcement learning on Dino game, we capture pictures of states at regular intervals and then store them in AWS S3 after processing using OpevCV and PIL (Python Imaging Library) libraries. The interface between python and S3 will be AWS SDK for Python (Boto3). Boto is the Amazon Web Services SDK for Python. It enables Python developers to create, manage, and configure AWS services, such as S3 and EC2. Boto provides a simple to use, object-oriented API, as well as low-level access to AWS services. Boto3 has two distinct levels of APIs. Client (or "low-level") APIs provide one-to-one mappings to the underlying HTTP API operations. Resource APIs hide explicit network calls but instead provide resource objects and collections to access attributes and perform actions.

Data storage:

Data storage is a significant factor while designing the architecture of an application, also is its retrieval. While dealing with big data, some of the largest companies, ensure that they have their infrastructure in place to reduce the involvement of third parties. This also increases the reliability of their application as they are always not relying on someone else to keep their use online. This process immensely reduces data related expenses in the long term and provides them a lot more flexibility as they are after-all designing their system. The most famous example of such a shift is the one which Walmart made from Amazon's storage solutions to its own.

For smaller companies or smaller applications, using third-party services like AWS is very advantageous. It reduces the requirement of physical equipment while also getting additional features such as instantaneous scaling in case of an abnormal amount of load on the servers.
In academic projects, the level of complexity has to be determined before deciding on the "best" solution. For non-reinforcement learning projects, the significant factors that help the developers make these decisions are right in front of them – they get to decide how much data their application might produce, what the transfer rate should be, how much minimum data storage is required. However, in reinforcement learning projects, in particular, to the mode-free and off-policy problems, the developers are left to make a substantial guess to approximate how much data storage they require. This approximation is made just based off of how many variables are being counted for, how complex the problem can be as the agent explores more and more of the environment and counters more and more obstacles, and so on.

In our project, we are recording a few factors like Q-value, loss, action, state, and a python pickle file called "D." In python, a pickle file is one in which data is dumped to – a point which will be covered in detail in the following paragraphs. Mainly, we expect that the model has to train for multiple weeks on end to become a master at the game and reach scores that are not possible by a human. Also, we are making sure the values mentioned above are stored so that if the program is stopped for whatever reason, we will have a backup of the "knowledge" the AI has developed till the point of failure.

This ensures that the model isn't required to be run continuously in our system to train – it can resume training whenever the program is run. The importance of this is that the model takes weeks to properly train due to the limited CPU power available.

Data analysis:

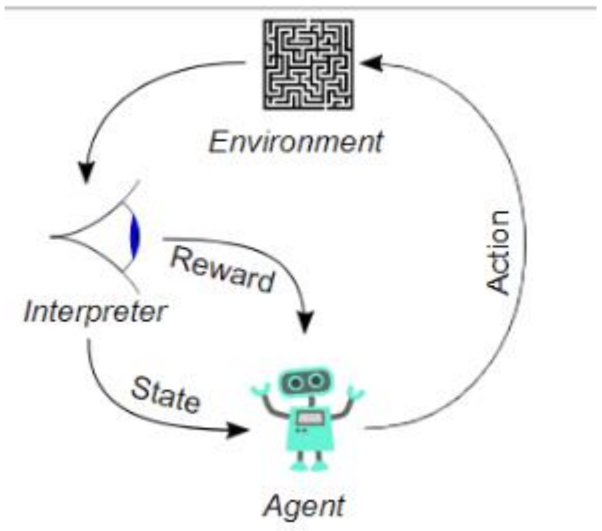
In a reinforcement learning model, the scope for analysis is limited. The concept of reinforcement learning is that the agent explores and learns about the environment to a very high degree of predictive accuracy. Essentially there is very limited output to the user unless specified in print statements in the code. The program is supposed to record and "learn" from previously made actions, and this "learning" is based on the reward/punishment that is given to each action it makes.
The agent is allowed to observe the game, making "random" actions which are usually continually jumping during the observation phase – it is exploring the environment. Once the agent gets to know what it will be an encounter, we now have to make it exploit the environment to make sure it aims for a higher total reward earned per play. The way this is done is by introducing a factor, Epsilon. If the epsilon is low, the program aims to get the highest possible award based on the training it received before. The observing and exploring phase is as crucial as the training phase to ensure a good model at the end, and their important cannot be stressed enough. Therefore, epsilon is usually set to a more considerable value in the initial stages and is mathematically reduced after each action based on whether the action gave it a reward or a punishment. This allows the model to be near perfect as it closes toward the final epsilon value, which would typically be 0.001 or 0.0001 based on the predictive accuracy that the developer wants to achieve.

In our current project, the agent is mainly checking if there is any obstacle approaching and acts according to the training it received by determining the higher Q value of the two actions – jump or no jump. Then it saves the values it obtained as explained in the data storage section, keeps track of the previous steps to make the next one. The concept of epsilon becomes evident here, the agent keeps dumping data into the pickle file and reads a certain number of previous actions to calculate the overall reward and the reason that it received a reward, if the agent keeps jumping it will keep getting rewards but as per the game restrictions the agent cannot jump while mid-air. If the concept of decaying the epsilon is not used, the agent is either stuck making the same mistake or will not "look" for a higher total reward. In lay-man terms, this means that the agent will keep jumping but will receive a heavy punishment for hitting an obstacle even when it issues a jump command which contradicts our training policy - our goal is to make sure the agent does not jump till the exact required point of time where the obstacle is about to hit the agent. The only data we will obtain is the scaling of score vs. the number of games played, which is expected to grow. With this, we will be able to effectively reduce the training time required for the program depending on the amount of accuracy we desire. Thus, we can say there is very minimal analysis required, or that can be done in our program, which is a characteristic that is derived from the simplified game rules and environment. In an AI model that uses multiple AI agents, analysis of their interactions is required to optimize the learning curve.

The reason AI for gaming is highly pursued is the involvement of multiple agents, the un-predictable environment changes, frequency of these changes, and so much more. Establishing a working AI model with a video game as a use case allows the developers to work out the defects and optimize the training of the model without the risk or requirement of using physical objects.

Reinforcement Learning Analogy

A child learning to walk - This might be a new word for many but each and every one of us has learned to walk using the concept of Reinforcement Learning(RL) and this is how our brain still works. A reward system is a basis for any RL algorithm. If we go back to the analogy of child's walk, a positive reward would be a clap from parents or ability to reach a candy and a negative reward would be say no candy. The child then first learns to stand up before starting to walk. In terms of Artificial Intelligence, the main aim for an agent, in our case the Dino , is maximize a certain numeric reward by performing a particular sequence of actions in the environment. The biggest challenge in RL is the absence of supervision (labelled data) to guide the agent. It must explore and learn on its own. The agent starts by randomly performing actions and observing the rewards each action brings and learns to predict the best possible action when faced with a similar state of the environment.



Reinforcement learning (RL) is termed as goal-oriented algorithms, and the main objective is to learn from its own experiences. The main idea is the Reward system and how to maximize the rewards in a particular situation. There are four terms used in RL which are Agent, Environment, Action and Reward. The agent observes the environment, takes an action to interact with the environment, and receives positive or negative reward. The agent randomly performs actions and observe rewards that each action brings. With this, the agent learns the best possible action when there is a similar state of environment. RL can be categorized as model-free and model-based. In Model-free RL, there is no model. The agent creates its own model by sampling and observing. However, in the case of model-based RL, optimal actions are calculated using models. Most popular model-free algorithms are Monte Carlo, Q-Learning, SARSA etc.

We use Q-learning, a technique of RL, where we try to approximate a special function which drives the action-selection policy for any sequence of environment states

Q-learning

It is a model-less implementation of Reinforcement Learning where a table of Q values is maintained against each state, action taken and the resulting reward. In our case, the states are game screenshots and action jump and do nothing[0,1]

Higher level idea about Q-LEARNING:

In Q-learning and related algorithms, an agent tries to learn the optimal policy from its history of interaction with the environment. A history of an agent is a sequence of state-action-rewards:

$\langle s_0,a_0,r_1,s_1,a_1,r_2,s_2,a_2,r_3,s_3,a_3,r_4,s_4...\rangle$,

which means that the agent was in state s_0 and did action a_0, which resulted in it receiving reward r_1 and being in state s_1; then it did action a_1, received reward r_2, and ended up in state s_2; then it did action a_2, received reward r_3, and ended up in state s_3; and so on.

We treat this history of interaction as a sequence of experiences, where an experience is a tuple

$\langle s,a,r,s'\rangle$,

which means that the agent was in state s, it did action a, it received reward r, and it went into state s'. These experiences will be the data from which the agent can learn what to do. As in decision-theoretic planning, the aim is for the agent to maximize its value, which is usually the discounted reward.

Q-learning explained with an example:

The basic gist of Q-learning is that you have a representation of the environmental states s, and possible actions in those states a, and you learn the value of each of those actions in each of those states. Intuitively, this value, Q, is referred to as the state-action value. So in Q-learning you start by setting all your state-action values to 0 (this isn't always the case, but in this simple implementation it will be), and you go around and explore the state-action space. After you try an action in a state, you evaluate the state that it has lead to. If it has lead to an undesirable outcome you reduce the Q value (or weight) of that action from that state so that other actions will have a greater value and be chosen instead the next time you're in that state. Similarly, if you're rewarded for taking a particular action, the weight of that action for that state is increased, so you're more likely to choose it again the next time you're in that state. Importantly, when you update Q, you're updating it for the previous state-action combination. You can only update Q after you've seen what results.

Let’s look at an example in the cat vs mouse case, where you are the mouse. You were in the state where the cat was in front of you, and you chose to go forward into the cat. This caused you to get eaten, so now reduce the weight of that action for that state, so that the next time the cat is in front of you you won’t choose to go forward you might choose to go to the side or away from the cat instead (you are a mouse with respawning powers). Note that this doesn’t reduce the value of moving forward when there is no cat in front of you, the value for ‘move forward’ is only reduced in the situation that there’s a cat in front of you. In the opposite case, if there’s cheese in front of you and you choose ‘move forward’ you get rewarded. So now the next time you’re in the situation (state) that there’s cheese in front of you, the action ‘move forward’ is more likely to be chosen, because last time you chose it you were rewarded.

Now this system, as is, gives you no foresight further than one time step. Not incredibly useful and clearly too limited to be a real strategy employed in biological systems. Something that we can do to make this more useful is include a look-ahead value. The look-ahead works as follows. When we’re updating a given Q value for the state-action combination we just experienced, we do a search over all the Q values for the state we ended up in. We find the maximum state-action value in this state, and incorporate that into our update of the Q value representing the state-action combination we just experienced.
For example, we’re a mouse again. Our state is that cheese is one step ahead, and we haven’t learned anything yet (blank value in the action box represents 0). So we randomly choose an action and we happen to choose ‘move forward’.

Exploration

In the most straightforward implementation of Q-learning, state-action values are stored in a look-up table. So we have a giant table, which is size N x M, where N is the number of different possible states, and M is the number of different possible actions. So then at decision time we simply go to that table, look up the corresponding action values for that state, and choose the maximum, in equation form:

Roadmap ahead:

1. Challenges in reinforcement learning
    a. Credit assignment problem
    b. Exploration-Exploitation dilemma and simple solutions to the problem.
2. Reinforcement learning in mathematical terms
    a. Markov Decision Process.
3. Long-term strategies
    a. Discounted future reward that forms the main basis for the algorithm
4. Estimate future reward
    a. Simple table-based Q-learning algorithm explained
5. Bigger state spaces

    a. Q-table replaced with a (deep) neural network.

6. Actual working
    a. Experience replay technique, that stabilizes the learning with neural networks.

Reinforcement learning lies in between supervised and unsupervised learning. Whereas in supervised learning, we have a target label for each training example and in unsupervised learning one we have no labels at all, in reinforcement learning, we have sparse and time-delayed labels – the rewards. They are based only on those rewards the agent has to learn to behave in the environment.

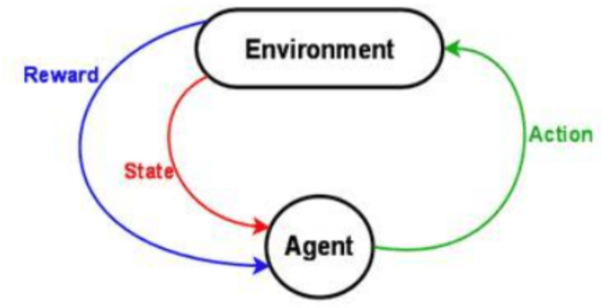
In practice there are many challenges. For instance when you jump and score a reward in the Dino game, it often has nothing to do with the actions you did just before getting the reward. All the efoort was already done, when you successfully jumped over a cactus. This phenomenon is called the credit assignment problem – which means the preceding actions was responsible for gaining the reward and to what extent. Once you have figured out a plan to collect a fixed number of rewards, do we finalize it or experiment with something that would result in even bigger rewards?

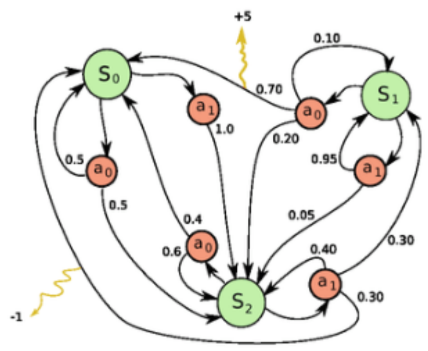
Exploration vs Exploitation problem floats to the surface when our model tends to perform the same actions while learning, in our game the model might learn that jumping leverages better reward rather than staying dormant and in turn apply an always jump policy. However, we would rate our model high to try out random actions while learning which can give better reward. We have a parameter ε, which regulates the randomness of actions. We eventually decay its value to decrease the randomness as we progress and then exploit rewarding actions.

Reinforcement learning is an vital model of how we learn. Appreciation from our parents, marks in school, these are all examples of rewards. Credit assignment difficulties and exploration-exploitation dilemmas come up daily both in business and in relationships. That’s why it is necessary to study this problem, and games form a wonderful sandbox for trying out new approaches.

Markov Decision Process

Suppose the agent, situated in an environment (e.g. Dino game). The environment is in a particular state (e.g. location of the dino and location of the cactus and so on). The agent can make certain actions in the environment (e.g. jump or stay dormant). These actions seldom result in a reward (e.g. increase in score). Actions change the environment and lead to a new state, where the agent can deliver another action, and so on. The rules for how you choose these actions are called policy. The environment, in general, is stochastic, which means the next state may be somewhat arbitrary





Left: reinforcement learning problem. Right: Markov decision process.

The set of states and actions, collectively with rules for transitioning from one state to another, build up a Markov decision process. One event of this process (e.g. one game) forms a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Here Si represents the state, ai is the action and ri+1 is the reward after performing the action. The event ends with concluding state sn . A Markov decision process relies on the Markov assumption, that the probability of the following state si+1 depends exclusively on current state si and action ai, but negative on preceding states or actions.

Discounted Future Reward

To perform better in the long-term, we need to consider into account not only the immediate rewards, but also the expected rewards we are going to get. Given one run of the Markov decision process, we can easily calculate the total reward for one episode:

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

Given that, the total future reward from time point t onward can be expressed as:

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

Because our environment is stochastic, we cannot be certain if we will get the equivalent rewards the following time we perform the same actions. The more further into the future we go, the more it may change. For that reason it is natural to use discounted future reward instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

Now, γ is the discount factor between 0 and 1 – the more into the future the reward is, the less we take it into consideration. It is simple to see, that discounted future reward at time step t can be calculated in terms of the same thing at time step t+1:

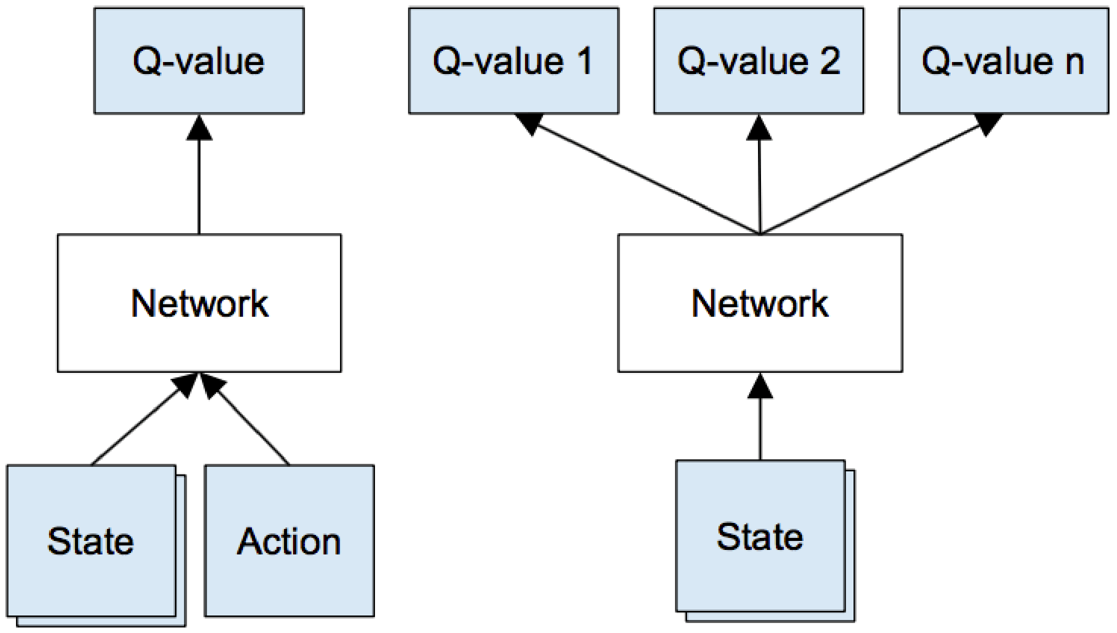
$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

If we set the discount factor γ=0, then our artifice will be short-sighted and we rely only on the immediate rewards. If we wish to balance between now and future rewards, we should set discount factor to something like γ=0.9. If our environment is deterministic and the same actions always result in the identical rewards, then we can set discount factor γ=1.
A good strategy for an agent would be to continually choose an action that maximizes the (discounted) future reward.

Deep Q Network

The state of the environment in the Dino game can be determined by the position of the cactus. This natural representation nevertheless is game specific, we come up with something more universal, that would be fitting for all the games? The apparent choice is screen pixels – they implicitly include all of the relevant data about the game condition, except for the speed and direction of the Dino movement. Two consecutive screens would have these included as well.

This is where deep learning steps in. Neural networks are exceptionally great at coming up with great features for extremely structured data. We could factor our Q-function along with a neural network, that uses the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively, we could use only game screens as input and output the Q-value for every possible action. This method has the advantage, that if we wish to perform a Q-value update or select the action with the highest Q-value, we only have to do one forward pass into the network and have all Q-values for all actions ready immediately.

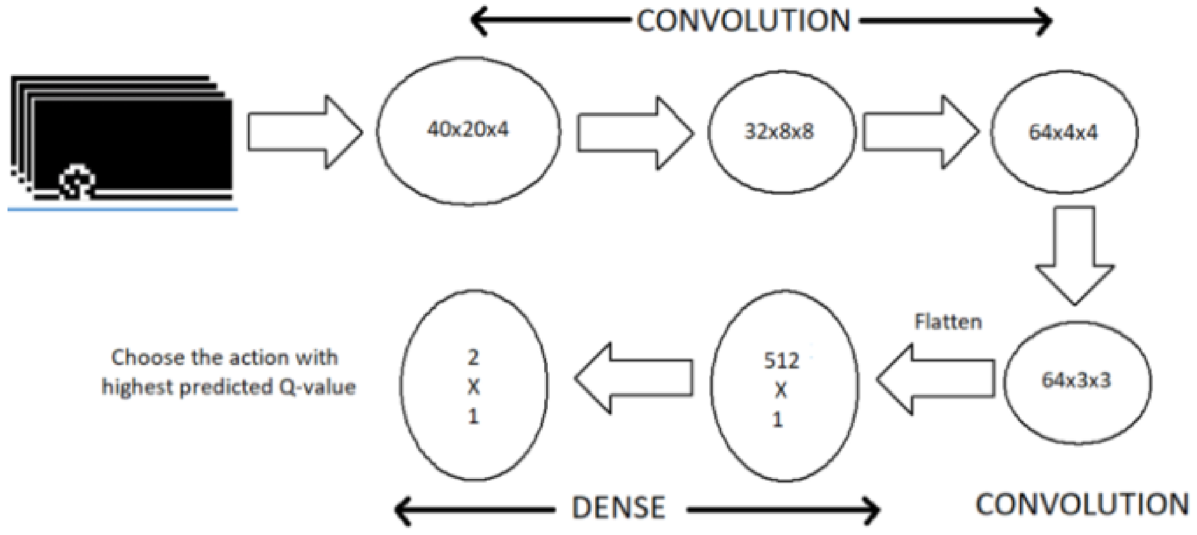


Left: Naive formulation of deep Q-network.

Right: More optimized architecture of deep Q-network, used in DeepMind paper.

NETWORK ARCHITECTURE

We used a series of three Convolutional layers flattened onto a Dense layer of 512 neurons. Pooling layers are intentionally removed. They are handy in image classification problems like ImageNet where we want the network to be insensitive to the location of the object. In our case, however, we care about the position of



Network Architecture with Layers

obstacles
People accustomed with object recognition networks may mark that there are no pooling layers. But if you actually think about it, pooling layers acquire you translation invariance – the architecture becomes insensitive to the location of an object in the image. That makes absolute sense for a classification task like ImageNet, but for games the location of the Cactus is important in determining the potential reward and we wouldn't need to discard this information!

Experience Replay

We now have an thought how to predict the future reward in each state using Q-learning and approximate the Q-function using a convolutional neural network. But it turns out that approximation of Q-values applying non-linear functions is not very permanent. There is a entire bag of tricks that have to be used to really make it converge. And it takes a large amount of time, almost a week on a single GPU.
The most important method is experience replay. During gameplay all the events < s, a, r, s' > are stored in a replay memory. While training the network, random mini-batches from the replay memory are practiced instead of the most recent transition. By doing so, this breaks the similarity of subsequent training samples, which otherwise might encourage the network into a local minimum. Also, experience replay gives the training task more alike to usual supervised learning, which simplifies debugging and examining the algorithm. One could truly collect all those experiences from human gameplay and then train the network on these.

Exploration-Exploitation

Q-learning strives to resolve the credit assignment problem – it generates rewards back in time, until it reaches the critical decision point, which was the actual cause for the earned reward.
Firstly observe, that when a Q-table is initialized randomly, then its predictions are originally random as well. If we select an action with the highest Q-value, the action will be arbitrary and the agent does rough "exploration". As a Q-function focalizes, it returns more uniform Q-values and the degree of exploration drops. So one could say that Q-learning includes the exploration as part of the algorithm. But this exploration is "greedy", it ends with the first effective strategy it finds.

A simple and effective fix for the earlier problem is ε-greedy exploration – with probability ε choose a random action, unless go with the "greedy" action with the highest Q-value.

What have we learned from this project

In the process of developing the AI in this project, we were exposed to many new concepts and these will be discussed in this section.

We learnt a lot about the various packages that we used in doing this project like pickle, json, io, selenium, matplotlib etc but the most nerve-wracking one was Keras. Keras is essentially a Neural Network library that runs on top of Tensorflow or Theano. The main reason we used Keras and not Tensorflow is that Keras is comparatively very user-friendly, modular and extensible. These qualities enable the user to conduct experiments with their code faster which in turn means that the user can tweak the program repeatedly to optimize it within the time that Tensorflow would have taken to do the same.

In this project, we used a specific type of neural network called a Convolutional Neural Network or CNN. Even though the name might be intimidating at first, it is a very simple to understand. Essentially CNN is a type of Deep Neural Network (DNN) that is optimized in processing images. A neural network basically takes in an input, compute required values using randomized parameters in a hidden layer and provide an output which is generally a prediction or classification. Deep neural networks are neural networks with more than 1 hidden layer. DNN is an umbrella that encapsulates all neural network architectures like CNN or RNN (Recurrent Neural Network).

The goal of the CNN we implemented in our project is that it should take a large image input, identify the outlines of the images using Canny Edge detection, and reduce the dimension of the image input to the agent. This reduces the load of processing the image immensely which is a requirement to run the program in our laptops.

A traditional CNN consists mainly of two processes: feature learning and classification. The feature learning process is a series of convolutional layers and pooling layers that scale down the images while increasing the accuracy of the features obtained at the output. In our model, we are not really concerned about the feature accuracy of the output as the input image contains only the edges of the agent and obstacles and we do not need the agent to understand the "features" of any obstacle. The classification layer is where the convoluted/pooled output from the feature learning process are "flattened" – meaning categorized – such that the agent understands the properties of the obstacle approaching it.

In the process of running a CNN, the user has to specify the shape of the input for the first layer along with kernel size, padding, stride and activation functions with each of the CNN layers. These values reduce as the number of layers increase so as to converge to a conclusion about the object in the image. Stride is an integer tuple that specifies stride length of the convolution – basically how many spaces the program shifts before calculating the value.

To counter the occurrence of any asymmetrical output from a CNN layer, we make sure that we specify that the padding should be the "same" and in cases where this is not a preferred solution but a symmetric output is needed, a zero padding is used where the empty spaces are filled in with zeros. The activation function decides whether a neuron should be "activated" or not based on calculating the weighted sum and adding a certain amount of bias to it. The purpose of specifying an activation function is to introduce non-linearity into the model. A CNN without an activating function is essentially a simple linear regression. The activation function we used is called "ReLU" or Rectified Linear activation function. The reason we went with ReLU is that it does not saturate when dealing with extreme values – the gradient is always high if the neuron activates. It is also very quick to evaluate. Alternatives to ReLU were the "sigmoid" or "tanh" activation functions. The reason they were not used is because they tend to saturate very quickly when exposed to extreme values and are not as quick as ReLU in their evaluation. After classification is almost done, the output must be optimized. There are two popular optimization functions in CNN – SGD or Stochastic Gradient Descent and Adam. In our model, we used the Adam optimization algorithm because it updates network weights iteratively based on training data rather than the simple standard weight calculation that the SGD provides.

Future implementation

In this section we wish to portray some of the challenges we faced that limited our approach to solving the problem and how we plan to get around to solving them in the future. First and foremost, due to time constraints we had to simplify the game by removing a lot of noise and reducing the game rules such that there are only two actions that an agent can perform. Also, capturing of the gameplay is done like a video recording of a corner part of the screen which is very inefficient because if there are any objects like windows over the selenium launch browser, then the borders of the windows are identified as objects to the agent. With more time, we believe that we will be able to create a more efficient method by using a window capturing implementation such that only the content of the particular window is only obtained and sent to the agent.

We would also be able to scale the program to a 3-action model by including a "duck" function and ensure that the agent will be able to incorporate and adjust to noise such as clouds, birds, the high score counter and so on. We are confident that with more time, we would be able to optimize the learning curve of the program to reduce the time it takes to reach scores that are almost impossible for any ordinary person can get.

A huge part of our "simplified" approach was due to the fact that we lacked processing power and understanding the concepts of Keras and Reinforcement Learning in-depth took almost 90% of our time to understand what was done in other similar projects and how we could implement reinforcement learning for the Chrome Dino game. We cannot stress on the fact that training of a RL model, even for such a simple model, will take weeks as we are starting from scratch with the equipment we have. Running the model overnight or even multiple days on-end was a real struggle for our laptops as we could not do other work on it while the model was training because of Python taking up 95% of the memory and the constant run-time of the processors resulted in the laptops crossing 95-100 degrees centigrade which caused the CPU to thermal throttle that lead to reduced clock-speed and reduced performance. Of the three group members, Saketh's laptop has the best configuration with an Intel i7 quad-core CPU and 16GB ram. Dismantling the laptop, cleaning the fans and replacing the thermal paste with one of the best brands on the market did not help solve the over-heating issue.

Adding a GPU to aid with faster image processing would be one of the solutions but applying this would require us to use Theano as our backend instead of Tensorflow. At the moment, we have not been able to implement the GPU based model for the game, but we look forward to doing so. The motivation for this is an AI developed by the OpenAI team for a game called DoTA2. Like us, they started with a simple set of rules with the agent exploring what it can do vs. what it should do to win the game. Their approach was similar to the Dino game – the bot has to learn the game from scratch by self-play and does not use imitation learning. The complexity in solving such a problem required an enormous amount of processing power and their solution reflected the same; they used about 60,000 CPU cores on Azure and 256 K80 GPUs on Azure. This allowed their model to collect almost 300 years of experience per day of playing. Details of the OpenAI team's project can be found at https://openai.com/blog/openai-five/

To battle the processing power issues above, we felt that the only solution would be to run the program on a cloud service like AWS. We tried to implement it in AWS's EC2 while storing the data files on S3 so that the program can be run 24x7 but with the limited time, we found it difficult to run it on the cloud completely. However, we intend to do so in the near future.

Conclusion

This project served as our introduction to reinforcement learning. We thought of implementing an AI program to play Chess or Connect 4 but realized very soon the complexity of involving human interaction or enforcing the game rules for them would prove an unsurpassable wall within the given time frame. Completing this project gave us a huge boost of confidence and we believe that we can build models that will be able to play those games or even work with some of the other groups to incorporate an RL model in their image-recognition and recommendation models.

References

1. https://blog.paperspace.com/dino-run/
2. https://github.com/chncyhn/flappybird-qlearning-bot
3. https://www.intel.ai/: Demystifying Deep Reinforcement Learning
    - David Silver's lecture about deep reinforcement learning
    - Slightly awkward but accessible illustration of Q-learning
    - UC Berkley's course on deep reinforcement learning
    - David Silver's reinforcement learning course
    - Nando de Freitas' course on machine learning (two lectures about reinforcement learning in the end)
    - Andrej Karpathy's course on convolutional neural networks