ANALYTICAL MECHANICS
EP3288

Reinforcement Learning in Cart-pole Game

Pranjal Desale EP19BTECH11006
Divyansh Kharbanda EP19BTECH11002
Apoorve Kalot EP17BTECH11003

November 22, 2020

Abstract

Cartpole - known also as an Inverted Pendulum is a pendulum with a center of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the center of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point.

Reinforcement Learning in Cart-pole Game

Name : Pranjal Narendra Desale

Abstract

Cartpole - known also as an Inverted Pendulum is a pendulum with a center of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the center of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point.

Table of contents :

1 Motivation

We were inspired by the automatic re-landing spacecrafts of the famous corporation "SpaceX", Founded by Elon Musk, which had the idea of developing a solution of reusing the spacecraft (which generally gets destroyed on manual re-entry and landing) by using Advance Machine Learning Technologies and Computer vision Techniques, to assist in the self-landing of spacecraft.

Although the level of details, parameters and problems are different and are far beyond the scope of the current level of project, we have tried to replicate the situation by using some of the preexisting state of the art tools. These would work to provide us the required insight in the form of an overall big picture of this spectacular problem.

Along with the fact that Reinforcement Learning (the field of studying and developing the interaction of machine with it's environment based on certain parameters/policy/rules,that helps in 'learning' and delivering results in the most efficient and automated way) is currently the field of interest for aspiring and experienced engineers alike, is used in the project to implement the solution for the problem. Getting to learn this modern technology itself provides us with the extra motivation for this solution.

2 Problem Statement

A pole is attached by an un-actuated joint to a cart, which moves along a friction-less track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

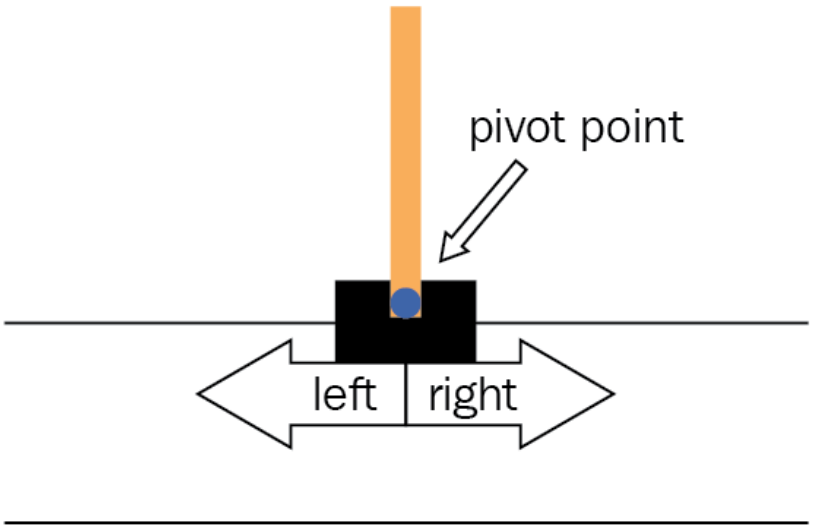

Figure 1: Simplistic Representation

2.1 Physics associated with the model

This is a problem with the law of conservation of momentum at its heart. The pole is in the state of an unstable equilibrium, and is kept upright through application of force by the cart. The force acting on the cart also acts on the pole and works to counteract the force of gravity. This must be done carefully and precisely to prevent the pole from crossing the boundary after which the cart cannot provide sufficient force to prevent the pole form falling

3 Reinforcement Learning Theory

Reinforcement Learning is a general concept that can be simply described with an agent that takes actions in an environment in order to maximize its cumulative reward. The underlying idea is very lifelike, where similarly to the humans in real life, agents in Reinforcement Learning algorithms are given incentives with punishments for bad actions and rewards for good ones.

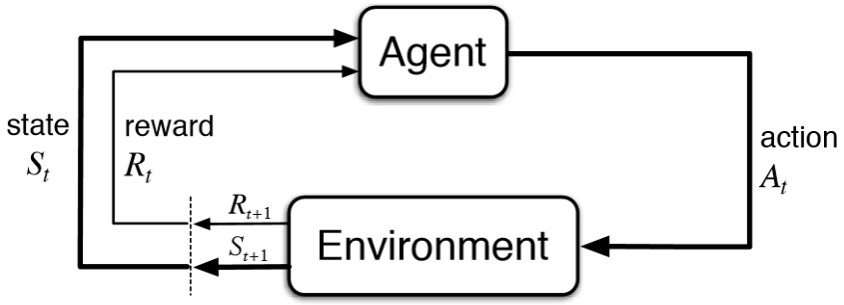

Figure 2: Overview of working of Reinforcement Learning

3.1 Deep Q-Learning (DQN)

DQN is a Reinforcement Learning technique that is aimed at choosing the best action for given circumstances (observation). Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Figure 3: Q-Learning Computation

3.2 Neural Networks

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. The neural network has a graph like structure with the neurons forming the various nodes.The connections between neurons are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times

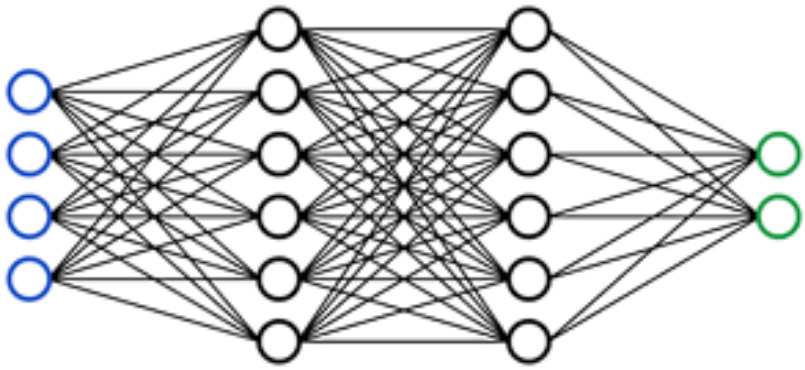

Figure 4: Diagrammatic representation of a neural network

The neural network trains on the data given to it to adjust the weight on it's edges. More the similarity it finds in data, the stronger the weights adjust to look out for similar inputs. Later when a different input with similar features is passed to the network while testing it, the same weighted edges give rise to desired output with similarity to the training data. That is how a neural network "learns". In short the neural network learns by repetition and it's behaviour can be thought abstractly to be muscle memory

4 Code base and it's working

The Code snippet below contains Imports of various packages of python which we will be using during implementation:

1) gym : for using predefined environment for deploying and rendering of Reinforcement learning Algorithms

2) Tensorflow, tflearn : for implementation of Neural Networks and using them for training the machine

3) matplotlib, numpy, seaborn, time : Various computation and data visualisation package.

```
import gym
import time
import tensorflow as tf
import tflearn
import numpy as np
import matplotlib
import seaborn as sns
%matplotlib inline
```

The code snippet below, initialises the environment i.e. Cart-pole Environment in "env" variable

```
env = gym.make("CartPole-v0")
```

To Test whether our environment is being initialised or not,we will try to check it, by running and testing out on it. The 4 values returned by the obs variable represent the cart position, cart velocity, pole angle, and the velocity of the tip of the pole.

```
env_temp = gym.make("CartPole-v0")
obs = env_temp.reset()
print("Parameter of the environment: ",obs)
```

Sample Static Environment

```
obs = env_temp.reset()
env_temp.render() # will render the environment in humanly visible format
time.sleep(7)     # will wait till 7 Seconds
env_temp.close()  # will close the rendered environment
```

Sample Dynamic Environment

Here we try to see what are the actions we can do in the environment and how it changes the environment.We use step(0) and step(1) to move the cartpole left and right, by applying -1 and +1 unit force respectively.

```
print("when step(0) is supplied for action, the resultant change in parameters.\n")
count = 0                       # Counter variable to store the change in the state of
    cartpole
obs = env_temp.reset()          # Resetting the environment before rendering the environment
```

```
while count <= 9:
    for i in range(200):
        time.sleep(1)
        env_temp.render()
        obs, r, d, _ = env_temp.step(0)
```

Here the Parameters:
1) obs : These 4 values represent the cart position, cart velocity pole angle and the velocity of the tip of the pole.
2) r : state, it's a positive number if it is above and negative if it falls.
3) d : it briefs whether we lost the game, True if it falls and False if it is still up.

We keep the cartpole from falling by applying the continuous force according to the situation demands

```
print("when step(0) is supplied for action, the resultant change in parameters.\n")
count = 0                       # Counter variable to store the change in the state of
    cartpole
obs = env_temp.reset()          # Reseting the environment before rendering the environment
while count <= 9:
    for i in range(200):
        time.sleep(1)
        env_temp.render()
        obs, r, d, _ = env_temp.step(0)
        print("Parameter Observed:",obs,r,d)
        if d == True:
            print("count:",count)
            count += 1
            break
env_temp.close()
```

Here the Parameters:
1) obs : These 4 values represent the cart position, cart velocity, pole angle and the velocity of the tip of the pole.
2) r : state, it's a positive number if it is above and negative if it falls.
3) d : it briefs whether we lost the game, True if it falls and False if it is still up

We keep the cartpole from falling by applying the continuous force according to the situation demands

```
print("\nWhen step(1) is supplied for action, the resultant change in parameters.\n")
count = 0
obs = env_temp.reset()
while count <= 9:
    for i in range(200):
        time.sleep(1)
        env_temp.render()
        obs, r, d, _ = env_temp.step(1)
        print("Parameter Observed:",obs,r,d)
        if d == True:
            print("count:",count)
            count += 1
            break
env_temp.close()
```

The Main task is to develop the agent/model which will take decision on it's own to deploy actions in the environment

To that we will using Neural Network, with 3 dense layer (i.e. 3 in between layers) with Rectified linear unit function for each of them, and finally we have a output layer which give us probability for each of the possible action to be taken.

```
observation = tflearn.input_data(shape=[None, 4]) # for storing the observation parameters
net = tflearn.fully_connected(observation, 256, activation="relu") # Dense Layer 1
net = tflearn.fully_connected(net, 256, activation="relu") # Dense Layer 2
net = tflearn.fully_connected(net, 256, activation="relu") # Dense Layer 3
out = tflearn.fully_connected(net, 2, activation="softmax") # final Output Layer

reward_holder = tf.placeholder(tf.float32, [None]) # Variable storing reward i.e. state whether it falls or not
action_holder = tf.placeholder(tf.int32, [None]) # Variable storing associated action for the respective state

responsible_outputs = tf.gather(tf.reshape(out, [-1]), tf.range(0, tf.shape(out)[0] * tf.shape(out)[1], 2) + action_holder)
# above is used to change the shape of accumulated outputs from different frames into single one


loss = -tf.reduce_mean(tf.log(responsible_outputs) * reward_holder) # to specify the loss for the neural network
optimizer = tf.train.AdamOptimizer() # function to be used to minimize the loss function across neural network
update = optimizer.minimize(loss) # function to be used to update the loss.
```

It is constant which we use to provide the amount of importance to the rewards which we get for each supplied actions

The more it is nearer to 1 (such as 0.90-0.99), more weightage is assigned to the reward while more nearer to 0 (such as 0.01-0.10), very less weightage is assigned to the reward

```
gamma = 0.99

def discount_reward(rewards):
    running_reward = 0
    result = np.zeros_like(rewards)
    for i in reversed(range(len(rewards))):
        result[i] = rewards[i] + gamma * running_reward
        running_reward += rewards[i]
    return result

# Example
print(discount_reward([5.0, 1.0, -2.5, 6.0]))
```

Choose an random action and then evaluating the output probabilities, reason why we are picking random action is due to the fact that we don't our agent to be biased towards single action but should also know the consequences of the both the action for random situation.

```
num_episodes = 1500 # No. of sessions, in which for each session we will try to acheive max possible reward
max_time = 200 # Max times of frames the cart-pole is kept from falling, to result in victory in game environment
all_rewards = [] #to keep track of rewards
saver = tf.train.Saver() # To save our Trained model
```

```
train_data = [] # To retrain the saved data
Whole_sess_hist = [] # Here we will be storing whole history of trained parameters


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(num_episodes):
        obs = env.reset()
        episode_reward = 0 # Counter for each episode's final obtained reward
        ep_history = [] # For storing each individual episode history
        for j in range(max_time):
            a_one_hot = sess.run(out, feed_dict={observation: [obs]}).reshape(2)
            action = np.random.choice(a_one_hot, p=a_one_hot) # Picking random action
            action = np.argmax(a_one_hot == action)
            obs1, r, d, _ = env.step(action) # Getting changed parameter for resultant random action
            ep_history.append([obs, r, action]) # Storing those parameter for running episode
#             Whole_sess_hist.append([obs,r,action])
            obs = obs1 # Overwriting initial observation with new one
            episode_reward += r
            if d == True:
                all_rewards.append(episode_reward)
                ep_history = np.array(ep_history)
                ep_history[:, 1] = discount_reward(ep_history[:, 1])
                train_data.extend(ep_history)
                if i % 10 == 0 and i != 0:
                    train_data = np.array(train_data)
                    sess.run(update, feed_dict={observation: np.vstack(train_data[:, 0]),
                                                reward_holder: train_data[:, 1],
                                                action_holder: train_data[:, 2]})
                    """
                    Above part is to update the weights associated with each episode to total data
                    """
                    train_data = []
                break

        if i % 100 == 0 and i != 0:
            """
            For every 100th session of updating, we will be printing the average weightage reward
            """
            print("For after",str(i),"th episode, following weighted reward: ",np.mean(all_rewards[-100:]))
            Whole_sess_hist.append([obs[0],obs[0],obs[0],obs[0],r,action])
            if np.mean(all_rewards[-100:]) == 200:
                break

    saver.save(sess, "/tmp/model.ckpt") #For Saving the weights of the trained model
```

5 Progression of solution with time

Since our model is trained, we will use the trained weights for testing and then will output the final reward for each episode which is in total 200 frames long.

```
max_time = 200 # For how much frame the cart-pole being need to kept stable to have Victory
saver = tf.train.Saver()

with tf.Session() as sess:

    saver.restore(sess, "/tmp/model.ckpt") # Restoring the Trained Model Weights
    #Show the results
    for i in range(10):

        obs = env.reset()
        episode_reward = 0
        for j in range(max_time):

            #Choose an random action for testing
            a_one_hot = sess.run(out, feed_dict={observation: [obs]}).reshape(2)
            action = np.random.choice(a_one_hot, p=a_one_hot)
            action = np.argmax(a_one_hot == action)
            env.render()
            time.sleep(0.005)
            obs, r, d, _ = env.step(action)
            episode_reward += r
            if d == True:
                break
        print("Final Weighted Reward attained after "+str(i)+"th episode: ",episode_reward)
        """
        Giving final reward for each output, after getting actions
        """
env.close()
```

The "output" of the training cycles is characterized by the "reward" parameter that was produced during that cycle of training.

These 4 values represent the cart position, cart velocity, pole angle, and the velocity of the tip of the pole. the next output is the state, it's a positive number if it is above and negative if it falls below. The final output printed briefs whether we lost the game. It is True if it falls and False if it is still up (It signals the end of the game. Step(0) is changing the state of system by applying unit force to the left, and Step(1) is applying unit force to the right

```
when step(0) is supplied for action, the resultant change in parameters.

Parameter Observed: [ 0.03050329 -0.2270382  -0.03631862  0.27727727] 1.0 False
Parameter Observed: [ 0.02596253 -0.4216237  -0.03077307  0.55828771] 1.0 False
Parameter Observed: [ 0.01753005 -0.61630047 -0.01960732  0.84111876] 1.0 False
Parameter Observed: [ 0.00520404 -0.81114935 -0.00278494  1.12757171] 1.0 False
Parameter Observed: [-0.01101894 -1.0062347   0.01976649  1.41937984] 1.0 False
Parameter Observed: [-0.03114364 -1.20159563  0.04815409  1.71817495] 1.0 False
Parameter Observed: [-0.05517555 -1.39723531  0.08251759  2.02544613] 1.0 False
Parameter Observed: [-0.08312026 -1.59310806  0.12302651  2.34248817] 1.0 False
Parameter Observed: [-0.11498242 -1.78910372  0.16987627  2.67033822] 1.0 False
Parameter Observed: [-0.15076449 -1.98502948  0.22328304  3.0097003 ] 1.0 True
count: 0
Parameter Observed: [-0.19046508 -2.18058977  0.28347704  3.36085988] 0.0 True
count: 1
Parameter Observed: [-0.23407688 -2.37536574  0.35069424  3.7235937 ] 0.0 True
count: 2
Parameter Observed: [-0.28158419 -2.56879713  0.42516612  4.09708449] 0.0 True
count: 3
Parameter Observed: [-0.33296013 -2.76017059  0.50710781  4.47985458] 0.0 True
count: 4
Parameter Observed: [-0.38816355 -2.94861989  0.5967049   4.8697362 ] 0.0 True
count: 5
Parameter Observed: [-0.44713594 -3.1331442   0.69409962  5.2638961 ] 0.0 True
count: 6
Parameter Observed: [-0.50979883 -3.31265019  0.79937754  5.65892656] 0.0 True
count: 7
Parameter Observed: [-0.57605183 -3.48602154  0.91255607  6.05100127] 0.0 True
count: 8
Parameter Observed: [-0.64577226 -3.65221536  1.0335761   6.43607338] 0.0 True
count: 9
```

```
When step(1) is supplied for action, the resultant change in parameters.

Parameter Observed: [ 0.01638224  0.16475619  0.02056938 -0.33099916] 1.0 False
Parameter Observed: [ 0.01967737  0.35957939  0.01394939 -0.61712518] 1.0 False
Parameter Observed: [ 0.02686896  0.55450372  0.00160689 -0.90538224] 1.0 False
Parameter Observed: [ 0.03795903  0.74960388 -0.01650076 -1.19755967] 1.0 False
Parameter Observed: [ 0.05295111  0.94493545 -0.04045195 -1.49536815] 1.0 False
Parameter Observed: [ 0.07184982  1.14052528 -0.07035931 -1.80040251] 1.0 False
Parameter Observed: [ 0.09466032  1.33635982 -0.10636736 -2.11409609] 1.0 False
Parameter Observed: [ 0.12138752  1.53237117 -0.14864928 -2.43766449] 1.0 False
Parameter Observed: [ 0.15203494  1.72842046 -0.19740257 -2.77203751] 1.0 False
Parameter Observed: [ 0.18660335  1.92427895 -0.25284332 -3.11777986] 1.0 True
count: 0
Parameter Observed: [ 0.22508893  2.1196077  -0.31519892 -3.4750037 ] 0.0 True
count: 1
Parameter Observed: [ 0.26748108  2.31393782 -0.384699   -3.84327993] 0.0 True
count: 2
Parameter Observed: [ 0.31375984  2.50665454 -0.46156459 -4.22155929] 0.0 True
count: 3
Parameter Observed: [ 0.36389293  2.69698964 -0.54599578 -4.60811879] 0.0 True
count: 4
Parameter Observed: [ 0.41783272  2.88402809 -0.63815816 -5.00055156] 0.0 True
count: 5
Parameter Observed: [ 0.47551329  3.0667349  -0.73816919 -5.395816  ] 0.0 True
count: 6
Parameter Observed: [ 0.53684798  3.24400723 -0.84608551 -5.79035182] 0.0 True
count: 7
Parameter Observed: [ 0.60172813  3.41475398 -0.96189254 -6.18025391] 0.0 True
count: 8
Parameter Observed: [ 0.67002321  3.57800071 -1.08549762 -6.56147291] 0.0 True
count: 9
```

The maximum possible score that the model can have is currently capped at 200, after which we assume that it has solved the problem of balancing the inverted pendulum and it can successfully keep it in air for indefinite periods of time. The training output progress is as follows:

```
For after 100 th episode, following weighted reward:  21.59
For after 200 th episode, following weighted reward:  26.22
For after 300 th episode, following weighted reward:  41.74
For after 400 th episode, following weighted reward:  82.47
For after 500 th episode, following weighted reward:  180.91
For after 600 th episode, following weighted reward:  200.0
```

Figure 5: Reward value progression with training cycle

6 Final Outputs

Since we have completed training the model, we now test it over multiple cycles. Upon running we can see that the model achieves perfect score every time. The output is 200 constantly for every training episode. This signifies that we have successfully solved the problem with our model.

```
INFO:tensorflow:Restoring parameters from /tmp/model.ckpt
Final Weighted Reward attained after 0th episode:  200.0
Final Weighted Reward attained after 1th episode:  200.0
Final Weighted Reward attained after 2th episode:  200.0
Final Weighted Reward attained after 3th episode:  200.0
Final Weighted Reward attained after 4th episode:  200.0
Final Weighted Reward attained after 5th episode:  200.0
Final Weighted Reward attained after 6th episode:  200.0
Final Weighted Reward attained after 7th episode:  200.0
Final Weighted Reward attained after 8th episode:  200.0
Final Weighted Reward attained after 9th episode:  200.0
```

Figure 6: write something

The same can be seen when we run the "gym" environment to see the cart-pole balancing itself successfully. We have in essence taught the model to learn from it's mistakes. This "natural" or "human" methodology of learning has since been applied to more sophisticated systems and it has successfully landed rockets with highest degree of accuracy for manoeuvring and positioning.

7 Ending Remarks

The problem that we solved as the cart-pole problem is just the beginning of what such technologies can achieve. Similar applications are easily extendable to the field of robotics and the automotive industry. Autonomous vehicles, drones and other high altitude aerial vehicles can adapt to adopt this technology to make travel and transportation more safe.
The field of reinforcement learning by itself is as deep as it is widespread. The maths, the algorithms, the subdivisions and it's use in conjunction with other technologies make it's potential impact to last across multiple industries no matter the problem at hand.

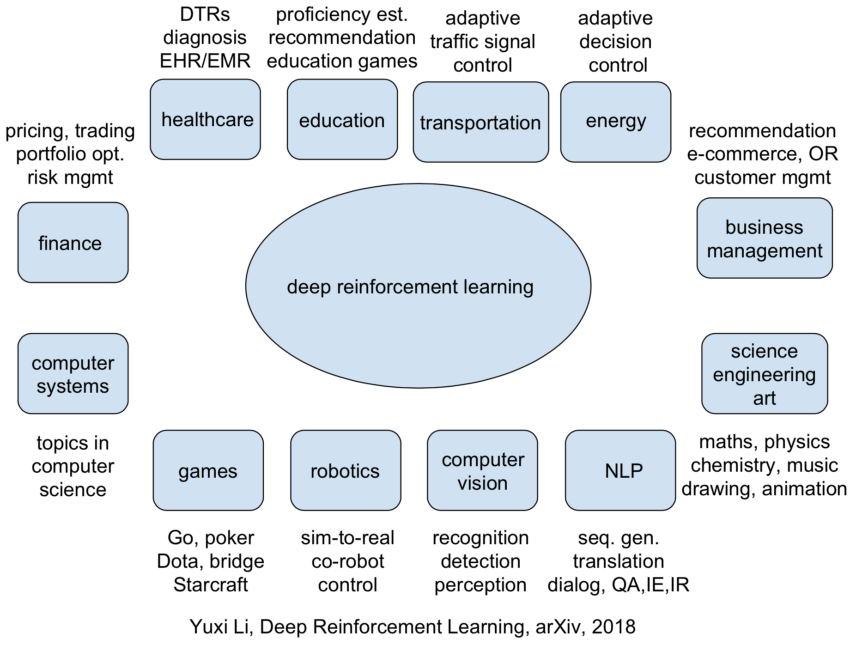

Figure 7: Applications of Reinforcement Learning

With all it's usefulness as a methodology of solving unorthodox problems, Reinforcement Learning faces it's own separate challenges. The main challenge in reinforcement learning lays in preparing the simulation environment, which is highly dependant on the task to be performed. Transferring the model out of the training environment and into to the real world is where things get tricky. Humanity's best brains are working everyday to overcome these hurdles; to ensure that this technology reaches the apex of its merit.

8 Bibliography

https://www.wikipedia.org/

E. Bisong. Google colaboratory. In Building Machine Learning and Deep Learning Models on Google Cloud Platform, pages 59–64. Springer, 2019.

S. Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. The Guardian, 15, 2016.

Google Colaboratory. Online gpu cloud by google. https:// colab.research.google.com/.

A. Gulli and S. Pal. Deep learning with Keras. Packt Publishing Ltd, 2017.

Kaggle. Online gpu cloud with datasets. https://www.kaggle. com/.

Y. Li. Reinforcement learning applications. arXiv preprint arXiv:1908.06973, 2019.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

OpenAI Gym. Toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com/.

C. J. Watkins and P. Dayan. Q-learning. Machine learning, 8(3- 4):279–292, 1992.https://www.sciencedirect.com/topics/neuroscience/reinforcement-learning