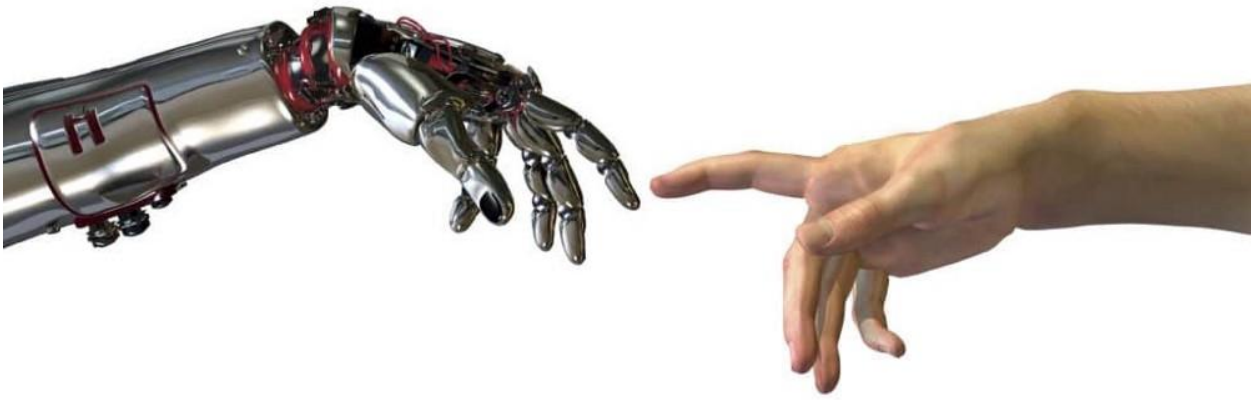
Deep Reinforcement Learning Course

Hello world, I present you the Deep Reinforcement Learning Course.



In this course, I tried to put together all the major breakthroughs and advancements of Reinforcement Learning starting by the very base concepts of RL to sophisticated state-of-the-art algorithms. It's a collection of articles and post published in theaisummer.com over the years.

The goal is to provide an intuitive explanation of the ideas behind all the algorithms that revolutionalize Artificial Intelligence over the past years, without getting into much math. As DRL is now making its way into industrial applications, we need to have a good understanding of how these algorithms work on a higher level before diving deeper into the equations and mathematical proofs.

Artificial Intelligence has the power to potentially solve all the world's major problems such as extreme poverty, quality education, and economic prosperity. And this is not an overstatement. Thus, we all need to know what the heck is going on there and finally look inside the black box. Right now.

As you read through the course, I hope you understand how simple and how powerful the algorithms actually are.

I hope you enjoyed it.

Let's begin!

Contents

1) The secrets behind Reinforcement Learning

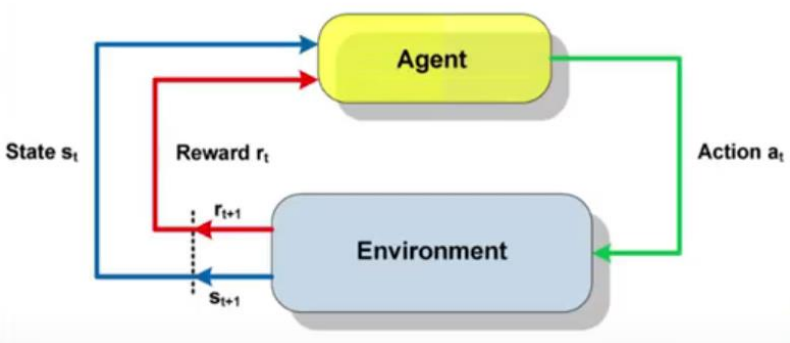
Bots that play Dota2, AI that beat the best Go players in the world, computers that excel at Doom. What's going on? Is there a reason why the AI community has been so busy playing games?

Let me put it that way. If you want a robot to learn how to walk what do you do? You build one, program it and release it on the streets of New York? Of course not. You build a simulation, a game, and you use that virtual space to teach it how to move around it. Zero cost, zero risks. That's why games are so useful in research areas. But how do you teach it to walk? The answer is the topic of today's article and is probably the most exciting field of Machine learning at the time:

You probably knew that there are two types of machine learning. Supervised and unsupervised. Well, there is a third one, called Reinforcement Learning. RL is arguably the most difficult area of ML to understand cause there are so many, many things going on at the same time. I'll try to simplify as much as I can because it is a really astonishing area and you should definitely know about it. But let me warn you. It involves complex thinking and 100% focus to grasp it. And some math. So, take a deep breath and let's dive in:

Markov Decision Processes

Reinforcement learning is a trial and error process where an AI (**agent**) performs a number of **actions** in an **environment**. Each unique moment the agent has a **state** and acts from this given state to a new one. This particular action may on may not has a **reward.** Therefore, we can say that each learning epoch (or episode) can be represented as a sequence of states, actions, and rewards. Each state depends only on the previous states and actions and as the environment is inherently stochastic (we don't know the state that comes next), this process satisfies the Markov property. The Markov property says that the conditional probability distribution of future states of the process depends only upon the present state, not on the sequence of events that preceded it. The whole process is referred to as a Markov Decision Process. Markov Decision Process is the main mathematical tool we use to frame almost any RL problem in a way that is easy to study and experimenting different solutions.



Let's see a real example using Super Mario. In this case:

- The agent is, of course, the beloved Mario
- The state is the current situation (let's say the frame of our screen)
- The actions are: left, right movement and jump
- The environment is the virtual world of each level;
- And the reward is whether Mario is alive or dead.

Ok, we have properly defined the problem. What's next? We need a solution. But first, we need a way to evaluate how good the solution is?

What I am saying is that the reward on each episode it's not enough. Imagine a Mario game where the Mario is controlled by an agent. He is receiving constantly positive rewards through the whole level by just before the final flag, he is killed by one Hammer Bro (I hate those guys). You see that each individual reward is not enough for us to win the game. We need a reward that captures the whole level. This is where the term of the **discounted cumulative expected reward** comes into play.

$$R = \sum_{t=0}^{\infty} \gamma^t r_t,$$

It is nothing more than the sum of all rewards discounted by a factor gamma, where gamma belongs to [0,1). The discount is essential because the rewards tend to be much more significant in the beginning than in the end. And it makes perfect sense.

The next step is to solve the problem. To do that we define that the goal of the learning task is: The agent needs to learn which action to perform from a given state that maximized the cumulative reward over time. Or to learn the **Policy π: S->A.** The policy is just a mapping between a state and an action.

To sum all the above, we use the following equation:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+1}$$

$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

where V (**Value**) is the **expected** long-term reward achieved by a policy (π) from a state (s).

You still with me? If you are, let's pause for 5 seconds cause this was a bit overwhelming:

1…2…3…4…5

Now that we regain our mental clarity, let recap. We have the definition of the problem as a Markov Decision Process and we have our goal to learn the best Policy or the best Value. How we proceed?

We need an algorithm (Thank you, Sherlock…)

Well, there is an abundance of developed RL algorithms over the years. Each algorithm focuses on a different thing, whether it is to maximize the value or the policy or both. Whether to use a model(e.g a neural network) to simulate the environment or not. Whether it will capture the reward on each step or the end. As you guessed, it is not very easy to categorize all those algorithms in classes, but that's what I am about to do.

As you can see we can classify RL algorithms in two big categories: Model-based and Model-free:

Model-based

These algorithms aim to learn how the environment works (its dynamics) from its observations and then plan a solution using that model. When they have a model, they use some planning method to find the best policy. They known to be data efficient, but they fail when the state space is too large. Try to build a model-based algorithm to play Go. Not gonna happen.

Dynamic programming methods are an example of model-based methods, as they require the complete knowledge of the environment, such as transition probabilities and rewards.

Model-free

Model-free algorithms do not require to learn the environment and store all the combination of states and actions. The can be divided into two categories, based on the ultimate goal of the training.

Policy-based methods try to find the optimal policy, whether it's stochastic or deterministic. Algorithms like policy gradients and REINFORCE belong in this category. Their advantages are better convergence and effectiveness on high dimensional or continuous action spaces.

Policy-based methods are essentially an optimization problem, where we find the maximum of a policy function. That's why we also use algorithms like evolution strategies and hill climbing.

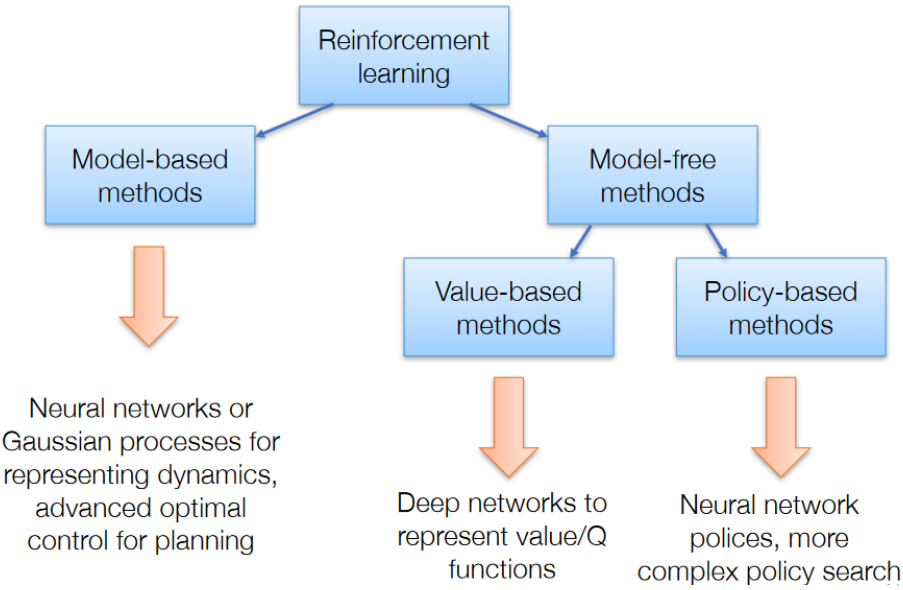
Value-based methods, on the other hand, try to find the optimal value. A big part of this category is a family of algorithms called Q-learning, which learn to optimize the Q-Value. I plan

to analyze Q-learning thoroughly on a next article because it is an essential aspect of Reinforcement learning. Other algorithms involve SARSA and value iteration.

At the intersection of policy and value-based method, we find the **Actor-Critic** methods, where the goal is to optimize both the policy and the value function.

And now to the cool part. In the past few years, there is a new kid in town. And it was inevitable to affect and enhance all the existing methods to solve Reinforcement Learning. I am sure you guessed it. Deep Learning. And thus, we have a new term to represent all those new research ideas.

Deep Reinforcement Learning



Deep neural networks have been used to model the dynamics of the environment(mode-based), to enhance policy searches (policy-based) and to approximate the Value function (value-based). Research on the last one (which is my favorite) has produced a model called Deep Q Network, which is responsible ,along with its many improvements, for some of the most astonishing breakthroughs around the area (take Atari for example). And to excite you, even more, we don't just use simple Neural Networks but Convolutional, Recurrent and many else as well.

Ok, I think that's enough for the first contact with Reinforcement Learning. I just wanted to give you the basis behind the whole idea and present you an overview of all the important techniques implemented over the years. But also, to give you a hint of what's next for the field.

Reinforcement learning has applications both in industry and in research. To name a few it has been used for: Robotics control, Optimizing chemical reactions, Recommendation systems, Advertising, Product design, Supply chain optimization, Stock trading. I could go on forever.

Its probably the most exciting area of AI right now and in my opinion, it has all the rights to be.

This is the first post on a long series of posts , where we are going to unveil Reinforcement Learning secrets and try to explain both the intuition and the math behind all the different algorithms. The main focus will be how Deep Learning is used to greatly enhance the existing techniques and how it has led to revolutionary results in just a few years.

In the next section, we will dive into Q-Learning , the Bellman equation and explain why Deep Q Networks, proposed in 2015, opened the door to a new era in AI.

2)Q Learning and Deep Q Networks

It's time to analyze the infamous Q-learning and see how it became the new standard in the field of AI (with a little help from neural networks).

First things first. In the previous section, we saw the basic concept behind Reinforcement Learning and we frame the problem using an agent, an environment, a state (S), an action(A) and a reward (R). We talked about how the whole process can be described as a Markov Decision Process and we introduced the terms Policy and Value. Lastly, we had a quick high-level overview of the base methods out there.

Remember that the goal is to find the optimal policy and that policy is a mapping between state and actions. So, we need to find which action to take while we stand in a specific state in order to maximize our expected reward. One way to find the optimal policy is to make use of the value functions (a model-free technique).

And here we will get to the new stuff. In fact, there are two value functions that are used today. The state value function V(s) and the action value function Q(s, a) .

- **State value function**: Is the expected return achieved when acting from a state according to the policy.
- **Action value function**: Is the expected return given the state and the action.

What is the difference you may ask? The first value is the value of a particular state. The second one is the value of that state plus the values of all the possible actions from that state.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right]$$

When we have the action value function, the Q value, we can simply choose to perform the action with the highest value from a state. But how do we find the Q value?

What is Q learning?

So, we will learn the Q value from trial and error? Exactly. We initialize the Q, we choose an action and perform it, we evaluate it by measuring the reward and we update the Q accordingly. In first, randomness will be a key player but as the agent explores the environment, the algorithm will find the best Q value for each state and action. Can we describe this mathematically?

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

Thank you Richard E. **Bellmann**. The above equation is known as the Bellman equation and plays a huge part in RL today's research. But what does it state?

The Q value, aka the maximum future reward for a state and action, is the immediate reward plus the maximum future reward for the next state. And if you think about it, it makes perfect sense. Gamma (γ) is a number between [0,1] and its used to discount the reward as the time passes, given the assumption that actions in the beginning are more important than at the end (an assumption that is confirmed by many real-life use cases). As a result, we can **update the Q value iteratively**.

The basic concept to understand here is that the Bellman equation relates states with each other and thus, it relates Action value functions. That helps us iterate over the environment and compute the optimal Values, which give us the optimal Policy.

In its simplest form, Q values is a matrix with states as rows and actions as columns. We initialize the Q-matrix randomly, the agent starts to interact with the environment and measures the reward for each action. It then computes the observed Q values and updates the matrix.

```
env = gym.make('MountainCar-v0')
#initalize Q tabe with zeros
Q = np.zeros([env.observation_space.n,env.action_space.n])

for i in range(episodes):
    s = env.reset()
    reward = 0
    goal_flag = False

    for j in range(200):
        # greedy action
        a = np.argmax(Q[s,:] + np.random.randn(1,env.action_space.n)*(1./(i+1)))
        s_new,r,goal_flag,_ = env.step(a)#state and reward
        maxQ=np.max(Q[s_new,:])
        # Belmann
        Q[s,a] += lr*(r + g*maxQ - Q[s,a])

        reward += r
        s = s_new

        if goal_flag == True:
            break
```

Exploration vs Exploitation

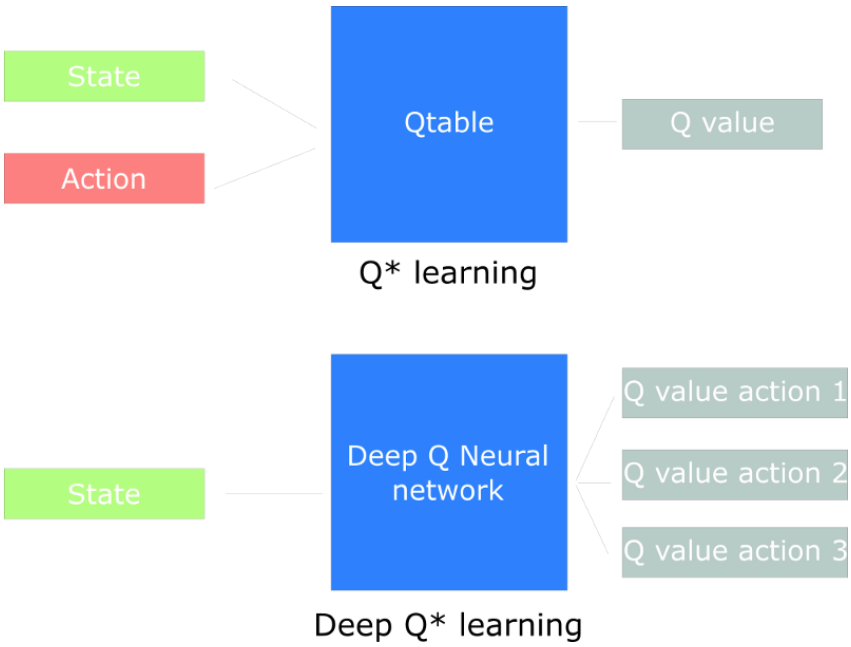
The algorithm, as described above, is a greedy algorithm, as it always chooses the action with the best value. But what if some action has a very small probability to produce a very large reward? The agent will never get there. This is fixed by adding random exploration. Every once in a while, the agent will perform a **random move**, without considering the optimal policy. But because we want the algorithm to converge at some point, we lower the probability to take a random action as the game proceeds.

Why going Deep?

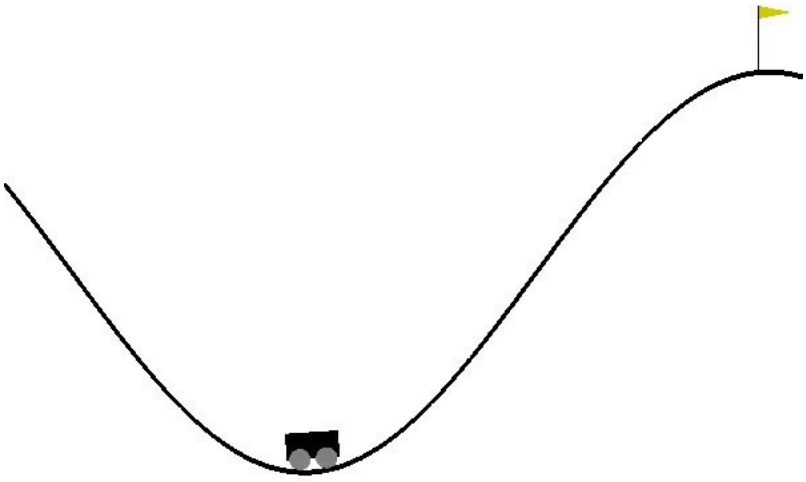
Q learning is good. No one can deny that. But the fact that it is ineffective in big state spaces remains. Imagine a game with 1000 states and 1000 actions per state. We would need a table of 1 million cells. And that is a very small state space comparing to chess or Go. Also, Q learning can't be used in unknown states because it can't infer the Q value of new states from the previous ones.

What if we approximate the Q values using some machine learning model. What if we approximate them using neural networks? That simple idea (and execution of course) was the reason behind DeepMind acquisition from Google for 500 million dollars. DeepMind proposed an algorithm named Deep Q Learner and used it to play Atari games with impeccable mastery.

In deep Q learning, we utilize a neural network to approximate the Q value function. The network receives the state as an input (whether is the frame of the current state or a single value) and outputs the Q values for all possible actions. The biggest output is our next action. We can see that we are not constrained to Fully Connected Neural Networks, but we can use Convolutional, Recurrent and whatever else type of model suits our needs.



I think it's time to use all that stuff in practice and teach the agent to play Mountain Car. The goal is to make a car drive up a hill. The car's engine is not strong enough to climb the hill in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.



I will explain more about Deep Q Networks alongside with the code. First we should build our Agent as a Neural Network with 3 Dense layers and we are going to train it using Adam optimization.

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0  # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse',
                      optimizer=Adam(lr=self.learning_rate))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    #get action
    def act(self, state):
        #select random action with prob=epsilon else action=maxQ
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])
```

Key points:

- The agent holds a memory buffer with all past experiences.
- His next action is determined by the maximum output (Q-value) of the network.
- The loss function is the mean squared error of the predicted Q value and the target Q-value.
- From the Bellman equation we have that the target is R + g*max(Q).
- The difference between the target and the predicted values is called Temporal Difference Error (TD Error)

Before we train our DQN, we need to address an issue that plays a vital role on how the agent learns to estimate Q Values and this is:

Experience Replay

Experience replay is a concept where we help the agent to remember and not forget its previous actions by replaying them. Every once in a while, we sample a batch of previous experiences (which are stored in a buffer) and we feed the network. That way the agent **relives**

its past and improve its memory. Another reason for this task is to force the agent to release himself from oscillation, which occurs due to high correlation between some states and resulting in the same actions over and over.

```
def replay(self, batch_size):
        #sample random transitions
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                Q_next=self.model.predict(next_state)[0]
                target = (reward + self.gamma *np.amax(Q_next))

            target_f = self.model.predict(state)
            target_f[0][action] = target
            #train network
            self.model.fit(state, target_f, epochs=1, verbose=0)
```

Finally we get make our agent interact with the environment and train him to predict the Q values for each next action

```
env = gym.make('MountainCar-v0')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
agent = DQNAgent(state_size, action_size)
done = False
batch_size = 32

for e in range(EPISODES):
    state = env.reset()
    state = np.reshape(state, [1, state_size])

    for time in range(500):
        #env.render()# need OpenGL to run
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        reward = reward if not done else -10
        next_state = np.reshape(next_state, [1, state_size])
        #add to experience memory
        agent.remember(state, action, reward, next_state, done)
        state = next_state
        if done:
            print("episode: {}/{}, score: {}, e: {:.2}"
                  .format(e, EPISODES, time, agent.epsilon))
            break
    #experience replay
    if len(agent.memory) > batch_size:
        agent.replay(batch_size)
```

As you can see is the exact same process with the Q-table example, with the difference that the next action comes by the DQN prediction and not by the Q-table. As a result, it can be applied to **unknown** states. That's the magic of Neural Networks.

You just created an agent that learns to drive the car up the hill. Awesome. And what is more awesome is that the exact same code(I mean copy paste) can be used in many more games, from Atari and Super Mario to Doom(!!!)

Awesome!

Just on more time, I promise.

Awesome!

In the next episode, we will remain on the Deep Q Learning area and discuss some more advanced techniques such as Double DQN Networks, Dueling DQN and Prioritized Experience replay.

3)Taking Deep Q Networks a step further

Today's topic is … well, the same as the last one. Q Learning and Deep Q Networks. Last time, we explained what Q Learning is and how to use the Bellman equation to find the Q-values and as a result the optimal policy. Later, we introduced Deep Q Networks and how instead of computing all the values of the Q-table, we let a Deep Neural Network learn to approximate them.

Deep Q Networks take as input the state of the environment and output a Q value for each possible action. The maximum Q value determines, which action the agent will perform. The training of the agents uses as loss the **TD Error**, which is the difference between the maximum possible value for the next state and the current prediction of the Q-value (as the Bellman equation suggests). As a result, we manage to approximate the Q-tables using a Neural Network.

So far so good. But of course, there are a few problems that arise. It's just the way scientific research is moving forward. And of course, we have come up with some great solutions.

Moving Q-Targets

The first problem is what is called moving Q-targets. As we saw, the first component of the TD Error (TD stands for Temporal Difference) is the Q-Target and it is calculated as the immediate reward plus the discounted max Q-value for the next state. When we train our agent, we update the weights accordingly to the TD Error. But the same weights apply to both the target and the predicted value. You see the problem?

$$\Delta w = \alpha[(R + \gamma \, max_a \, \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)] \, \nabla_w \hat{Q}(s, a, w)$$

We move the output closer to the target, but we also move the target. So, we end up chasing the target and we get a highly oscillated training process. Wouldn't be great to keep the target fixed as we train the network. Well, DeepMind did exactly that.

Instead of using one Neural Network, it uses two. Yes, you heard that right! (like one wasn't enough already).

One as the main Deep Q Network and a second one (called **Target Network**) to update exclusively and periodically the weights of the target. This technique is called **Fixed Q-Targets.** In fact, the weights are fixed for the largest part of the training and they updated only once in a while.

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.model = self._build_model()
        self.target_model = self._build_model()
        self.update_target_model()

    def update_target_model(self):
        # copy weights from model to target_model
        self.target_model.set_weights(self.model.get_weights())
```

Maximization Bias

Maximization bias is the tendency of Deep Q Networks to overestimate both the value and the action-value (Q) functions. Why does it happen? Think that if for some reason the network overestimates a Q value for an action, that action will be chosen as the go-to action for the next step and the same overestimated value will be used as a target value. In other words, there is no way to evaluate if the action with the max value is actually the best action. How to solve this? The answer is a very interesting method and is called:

Double Deep Q Network

To address maximization bias, we use two Deep Q Networks.

On the one hand, the DQN is responsible for the **selection** of the next action (the one with the maximum value) as always.

On the other hand, the Target network is responsible for the **evaluation** of that action.

The trick is that the target value is not automatically produced by the maximum Q-value, but by the Target network. In other words, we call forth the Target network to calculate the target Q value of taking that action at the next state. And as a side effect, we also solve the moving target problem. Neat right? Two birds with one stone. By decoupling the action selection from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably.

```
def train_model(self):
        if len(self.memory) < self.train_start:
            return
        batch_size = min(self.batch_size, len(self.memory))
        mini_batch = random.sample(self.memory, batch_size)

        update_input = np.zeros((batch_size, self.state_size))
        update_target = np.zeros((batch_size, self.state_size))
        action, reward, done = [], [], []

        for i in range(batch_size):
            update_input[i] = mini_batch[i][0]
            action.append(mini_batch[i][1])
            reward.append(mini_batch[i][2])
            update_target[i] = mini_batch[i][3]
            done.append(mini_batch[i][4])

        target = self.model.predict(update_input)
        target_next = self.model.predict(update_target) #DQN
        target_val = self.target_model.predict(update_target) #Target model

        for i in range(self.batch_size):
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
```

```
                # selection of action is from model
                # update is from target model
                a = np.argmax(target_next[i])
                target[i][action[i]] = reward[i] + self.discount_factor *
(target_val[i][a])

        self.model.fit(update_input, target, batch_size=self.batch_size,epochs=1,
verbose=0)
```

You think that's it? Sorry to let you down. We are going to take this even further. What now? Are you going to add a third Neural Network? Haha!!

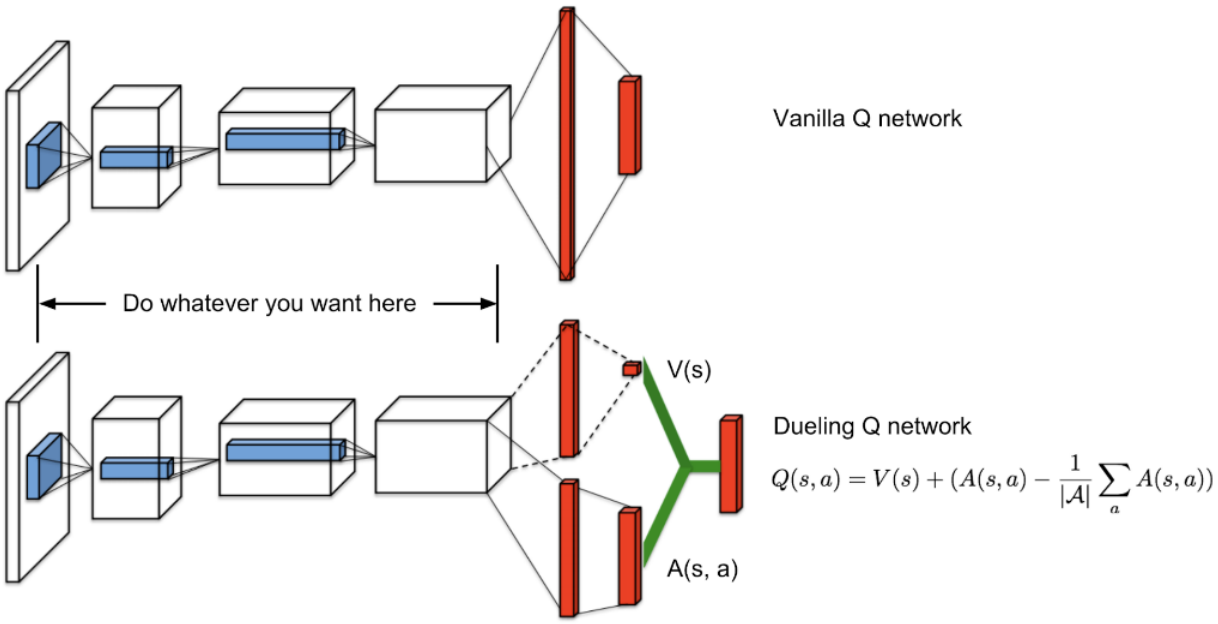
Well kind of. Who's laughing now?

Dueling Deep Q Networks

Let's refresh the basis of Q Learning first. Q values correspond to a metric of how good an action is for a particular state right? That's why it is an action-value function. The metric is nothing more the expected return of that action from the state. Q values can, in fact, be decomposed into two pieces: the state Value function V(s) and the advantage value A(s, a). And yes, we just introduce one more function:

Q(s, a)= V(s)+ A(s,a)

Advantage function captures how better an action is compared to the others at a given state, while as we know the value function captures how good it is to be at this state. The whole idea behind Dueling Q Networks relies on **the representation of the Q function as a sum of the Value and the advantage function**. We simply have two networks to learn each part of the sum and then we aggregate their outputs.



Do we earn something by doing that? Of course, we do. The agents in now able to evaluate a state without caring about the effect of each action from that state. Meaning that the features that determined whether a state is good or nor are not necessarily the same as the features that evaluate an action. And it may not need to care for actions at all. It is not uncommon to have actions from a state that do not affect the environment at all. So why take them into consideration?

* Quick note: If you take a closer look at the image, you will see that to aggregate the output of the two networks we do not simply add them. The reason behind that is the issue of identifiability. If we have the Q, we can't find the V and A. So, we can't back propagate. Instead, we choose to use the mean advantage as a baseline (the subtracted term).

Last but not least, we have on more issue to discuss and it has to do with optimizing the experience replay.

```
def build_model(self):
        input = Input(shape=self.state_size)
        shared = Conv2D(32, (8, 8), strides=(4, 4), activation='relu')(input)
        shared = Conv2D(64, (4, 4), strides=(2, 2), activation='relu')(shared)
        shared = Conv2D(64, (3, 3), strides=(1, 1), activation='relu')(shared)
        flatten = Flatten()(shared)

        # network separate state value and advantages
        advantage_fc = Dense(512, activation='relu')(flatten)
        advantage = Dense(self.action_size)(advantage_fc)
        advantage = Lambda(lambda a: a[:, :] - K.mean(a[:, :], keepdims=True),
                           output_shape=(self.action_size,))(advantage)

        value_fc = Dense(512, activation='relu')(flatten)
        value =  Dense(1)(value_fc)
        value = Lambda(lambda s: K.expand_dims(s[:, 0], -1),
                       output_shape=(self.action_size,))(value)

        q_value = merge([value, advantage], mode='sum')
        model = Model(inputs=input, outputs=q_value)
        model.summary()

        return model
```

Prioritized Experience Replay

Do you remember that experience replay is when we replay to the agent random past experiences every now and then to prevent him from forgetting them? If you don't, now you do. But some experiences may be more significant than others. As a result, we should prioritize them to be replayed. To do just that, instead of sampling randomly (from a uniform distribution), we sample using a priority. As priority, we define the magnitude of the TD Error (plus some constant to avoid zero probability for an experience to be chosen).

$$p = |\delta| + \varepsilon$$

Central idea: **The highest the error between the prediction and the target, the more urgent is to learn about it.**

And to ensure that we won't always replay the same experience, we add some stochasticity and we are all set. Also, for complexity's shake, we save the experiences in a binary tree called SumTree.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

```
from SumTree import SumTree

class PER:  # stored as ( s, a, r, s_ ) in SumTree
    e = 0.01
    a = 0.6
    def __init__(self, capacity):
        self.tree = SumTree(capacity)

    def _getPriority(self, error):
        return (error + self.e) ** self.a

    def add(self, error, sample):
        p = self._getPriority(error)
        self.tree.add(p, sample)

    def sample(self, n):
        batch = []
        segment = self.tree.total() / n

        for i in range(n):
            a = segment * i
            b = segment * (i + 1)

            s = random.uniform(a, b)
            (idx, p, data) = self.tree.get(s)
            batch.append((idx, data))

        return batch

    def update(self, idx, error):
        p = self._getPriority(error)
        self.tree.update(idx, p)
```

That was a lot. A lot of new information, a lot of new improvements. But just think what we can combine all of them together. And we do it.

I tried to give a summary of the most important recent efforts in the field, backed by some intuitive thought and some math. This is why Reinforcement Learning is so important to learn. There is so much

potential and so many capabilities for enhancements that you just can't ignore the fact that is going to be the big player in AI (if it already isn't). But that's why is so hard to learn and to keep up with it.

4)Unravel Policy Gradients and REINFORCE

In this chapter, we are going to keep ourselves busy with another family of Reinforcement learning algorithms, called policy-based methods. If you recall, there are two main groups of techniques when it comes to model-free Reinforcement Learning.

We analyze the first ones in two previous articles where we talked about Q-learning and Deep Q Networks and different improvement on the basic models such as Double Deep Q Networks and Prioritized Replay.

Let's do a quick rewind. Remember that we frame our problems as Markov Decision Processes and that our goal is to find the best Policy, which is a mapping from states to actions. In other words, we want to know what the action with the maximum expected reward from a given state is. In value-based methods, we achieve that by finding or approximating the Value function and then extract the Policy. What if we completely ditch the value part and find directly the Policy. This is what Policy-based methods do.

Don't get me wrong. Q-learning and Deep Q networks are great, and they are used in plenty application, but Policy-based methods offer some different advantages:

- They converge more easily to a local or global maximum and they don't suffer from oscillation
- They are highly effective in high-dimensional or **continuous** spaces
- They can learn **stochastic** policies (Stochastic policies give a **probability distribution** over actions and not a deterministic action. They used in stochastic environments, which they modeled as Partially Observable Markov Decision Processes where we do not know for sure the result of each action)

Hold on a minute. I told about convergence, local maximum, continuous space, stochasticity. What's going on in here?

Well, the thing is that **Policy based reinforcement learning is an optimization problem.** But what does this mean?

We have a policy (π) with some parameters theta (θ) that outputs a probability distribution over actions. We want to find the best theta that produces the best policy. But how we evaluate if a policy is good or bad? We use a policy objective function $J(\theta)$, which most often is the expected accumulative reward. Also, the objective function varies whether we have episodic or continuing environments.

$$\pi_\theta(a|s) = P[a|s]$$
$$J(\theta) = E_{\pi\theta}[\sum \gamma r]$$

So here we are, with an optimization problem in our hands. All we have to do is find the parameters theta (θ) that maximizes J(θ) and we have our optimal policy.

The first approach is to use a brute force technique and check the whole policy space. Hmm, not so good.

The second approach is to use a direct search in the policy space or a subset of it. And here we introduce the term of **Policy Search.** In fact, there are two families of algorithms that we can use. Let's call them:

- Gradient free
- Gradient-based

Think of any algorithm you have ever used to solve an optimization task, which does not use derivatives. That's a gradient-free method and most of them can be used in our case. Some examples include:

- Hill climbing is a random iterative local search
- Simplex: a popular linear programming algorithm (if you dig linear algebra check him out)
- Simulated annealing, which moves across different states based on some probability.
- Evolutionary algorithms that simulate the process of physical evolution. They start from a random state represented as a genome and through crossover, mutation and physical selection they find the strongest generation (or the maximum value). The whole "Survival of the fittest" concept wrapped in an algorithm.

The second family of methods uses Gradient Descent or to be more accurate Gradient Ascent.

In (vanilla) gradient descent, we:

1. Initialize the parameters theta
2. Generate the next episode
3. Get long-term reward
4. Update theta based on reward for all time steps
5. Repeat

But there is a small issue. Can we compute the gradient theta in an analytical form? Because if we can't, the whole process goes to the trash. It turns out that we can with a little trick. We have to assume that policy is differentiable whenever it is non-zero and to use logarithms. Moreover we define the state-action trajectory (τ) as a sequence of states, actions and rewards: τ = (s0,a0,r0, s1,a1,r1…, st,at,rt).

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

$$\nabla_\theta J(\theta) = E_\pi \left[\nabla_\theta (log \pi(\tau|\theta)) R(\tau) \right]$$

I think that's enough math for one day. The result, of course, is that we have the gradient in an analytical form and we can now apply our algorithm.

The algorithm described so far (with a slight difference) is called **REINFORCE** or **Monte Carlo policy gradient**. The difference from vanilla policy gradients is that we got rid of expectation in the reward as it is not very practical. Instead, we use stochastic gradient descent to update the theta. We **sample** from the expectation to calculate the reward for the episode and then update the parameters for each step of the episode. It's quite a straightforward algorithm.

Ok, let's simplify all those things. You can think policy gradients as so:

For every episode that we got a positive reward, the algorithm will increase the probability of those actions in the future. Similarly, for negative rewards, the algorithms will decrease the probability of the actions. As a result, in time, the actions that lead to negative results are slowly going to be filtered out and those with positive results will become more and more likely. That's it. If you want to remember one thing from the whole article, this is it. That's the essence of policy gradients. The only thing that changes every time is how we compute the reward, what policy do we choose (Softmax, Gaussian etc..) and how do we update the parameters.

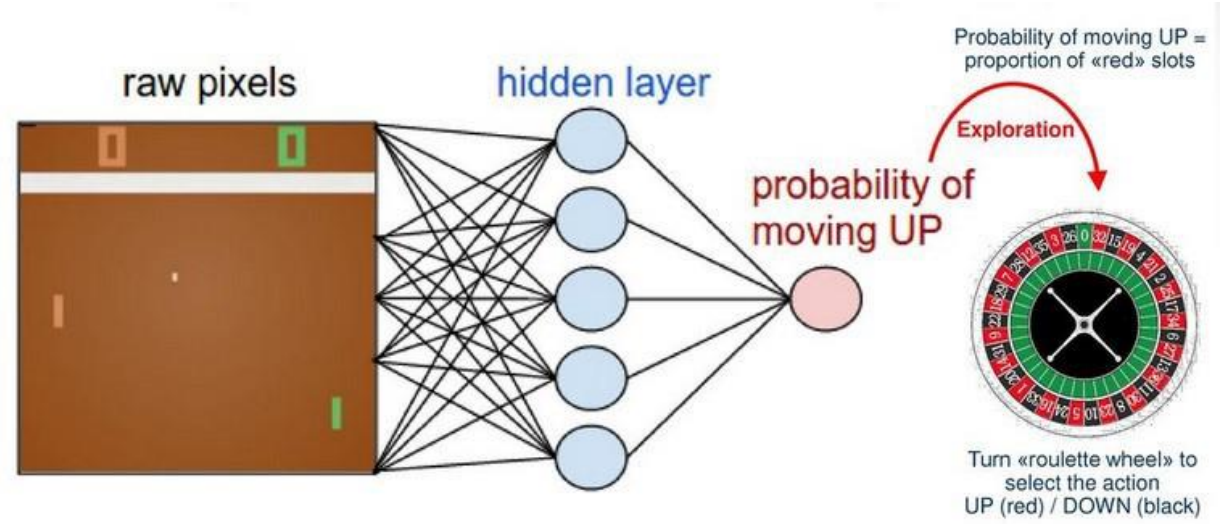
Now let's move on.

REINFORCE is, as mentioned, a stochastic gradient descent algorithm. Taking that into consideration, a question comes to mind. Why not use Neural Networks to approximate the policy and update the theta?

Bingo!!

It is time to introduce neural networks into the equation:

We can, of course, use pretty much any machine learning model to approximate the policy function (π), but we use a neural network such as a Convolutional Network because we like Deep Learning. A famous example is an agent that learns to play the game of Pong using Policy gradients and Neural Networks. In that example, a network receives as input frames from the game and outputs a probability of going up or down.



http://karpathy.github.io/2016/05/31/rl/

We will try to do something similar using the gym environment by OpenAI.

```
class REINFORCEAgent:

    # approximate policy using Neural Network
    # state is input and probability of each action is output of network
    def build_model(self):
        model = Sequential()
        model.add(Dense(self.hidden1, input_dim=self.state_size, activation='relu',
kernel_initializer='glorot_uniform'))
        model.add(Dense(self.hidden2, activation='relu',
kernel_initializer='glorot_uniform'))
        model.add(Dense(self.action_size, activation='softmax',
kernel_initializer='glorot_uniform'))
        model.compile(loss="categorical_crossentropy",
optimizer=Adam(lr=self.learning_rate))
        return model

    # using the output of policy network, pick action stochastically
    def get_action(self, state):
        policy = self.model.predict(state, batch_size=1).flatten()
        return np.random.choice(self.action_size, 1, p=policy)[0]

    # Agent uses sample returns for evaluating policy
    def discount_rewards(self, rewards):
        discounted_rewards = np.zeros_like(rewards)
```

```
        running_add = 0
        for t in reversed(range(0, len(rewards))):
            running_add = running_add * self.discount_factor + rewards[t]
            discounted_rewards[t] = running_add
        return discounted_rewards

    # update policy network every episode
    def train_model(self):
        episode_length = len(self.states)

        discounted_rewards = self.discount_rewards(self.rewards)
        discounted_rewards -= np.mean(discounted_rewards)
        discounted_rewards /= np.std(discounted_rewards)

        update_inputs = np.zeros((episode_length, self.state_size))
        advantages = np.zeros((episode_length, self.action_size))

        for i in range(episode_length):
            update_inputs[i] = self.states[i]
            advantages[i][self.actions[i]] = discounted_rewards[i]

        self.model.fit(update_inputs, advantages, epochs=1, verbose=0)


env = gym.make('CartPole-v1')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
scores, episodes = [], []

agent = REINFORCEAgent(state_size, action_size)

for e in range(EPISODES):
    done = False
    score = 0
    state = env.reset()
    state = np.reshape(state, [1, state_size])

    while not done:
        if agent.render:
            env.render()

        # get action for the current state and go one step in environment
        action = agent.get_action(state)
        next_state, reward, done, info = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        reward = reward if not done or score == 499 else -100

        # save the sample <s, a, r> to the memory
        agent.append_sample(state, action, reward)

        score += reward
        state = next_state

        if done:
            # every episode, agent learns from sample returns
```

```
            agent.train_model()
            score = score if score == 500 else score + 100
            scores.append(score)
            episodes.append(e)
```

You can see that the isn't trivial. We define the neural network model , the monte carlo sampling, the training process and then we let the agent to learn by interacting with the environment and update the weight at the end of each episode.

But policy gradients have their own drawbacks. The most important is that they have a high variance and it can be notoriously difficult to stabilize the model parameters.

Do you wanna know how we solve this?

(Hint: its actor-critic models)

5)The idea behind Actor-Critics and how A2C and A3C improve them

In this chapter the main topic is Actor-Critic algorithms, which are the base behind almost every modern RL method from Proximal Policy Optimization to A3C. So, to understand all those new techniques, you should have a good grasp of what Actor-Critic are and how they work.

But don't be in a hurry. Let's refresh for a moment on our previous knowledge. As you may know, there are two main types of RL methods out there:

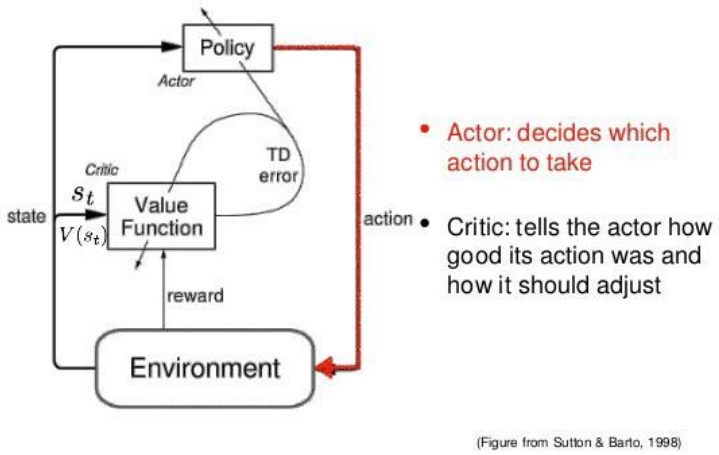
- Value Based: They try to find or approximate the optimal **value** function, which is a mapping between an action and a value. The higher the value, the better the action. The most famous algorithm is Q learning and all its enhancements like Deep Q Networks, Double Dueling Q Networks, etc
- Policy-Based: Policy-Based algorithms like Policy Gradients and REINFORCE try to find the optimal policy directly without the Q -value as a middleman.

Each method has their advantages. For example, policy-based are better for continuous and stochastic environments, have a faster convergence, while Value based are more sample efficient and steady.

When those two algorithmic families established in the scientific communities, the next obvious step is… to try to merge them. And this is how the Actor-Critic was born. Actor-Critics aim to take advantage of all the good stuff from both value-based and policy-based while eliminating all their drawbacks. And how do they do this?

The principal idea is to split the model in two: one for computing an action based on a state and another one to produce the Q values of the action.

The actor takes as input the state and outputs the best action. It essentially controls how the agent behaves by **learning the optimal policy** (policy-based). The critic, on the other hand, **evaluates the action by computing the value function** (value based). Those two models participate in a game where they both get better in their own role as the time passes. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately.



This idea of having two models interact (or compete) with each other is getting more and more popular in the field of machine learning in the last years. Think of Generative Adversarial Networks or Variational Autoencoders for example.

But let's get back to Reinforcement Learning. A good analogy of the actor-critic is a young boy with his mother. The child (actor) constantly tries new things and exploring the environment around him. He eats its own toys, he touches the hot oven, he bangs his head in the wall (I mean why not). His mother (the critic) watches him and either criticize or compliment him. The child listen to what his mother told him and adjust his behavior. As the kid grows, he learns what actions are bad or good and he essentially learns to play the game called life. That's exactly the same way actor-critic works.

The actor can be a function approximator like a neural network and its task is to produce the best action for a given state. Of course, it can be a fully connected neural network or a convolutional or anything else. The critic is another function approximator, which receives as input the environment and the action by the actor, concatenates them and output the action value (Q-value) for the given pair. Let me remind you for a sec that the Q value is essentially the maximum future reward.

The training of the two networks is performed separately and it uses gradient ascent (to find the global maximum and not the minimum) to update both their weights. As time passes, the actor is learning to produce better and better actions (he is starting to learn the policy) and the critic is getting better and better at evaluating those actions. It is important to notice that the update of the weights happen at each step (TD Learning) and not at the end of the episode, opposed to policy gradients.

Actor critics have proven able to learn big, complex environments and they have used in lots of famous 2d and 3d games, such as Doom, Super Mario, and others.

Are you tired? Because I now start getting excited and I plan on keep going. It's a really good opportunity to talk about two very popular improvements of Actor-critic models, A2C and A3C.

Advantage Actor-Critic (A2C)

What is Advantage? Q values can, in fact, be decomposed into two pieces: the state Value function V(s) and the advantage value A(s, a):

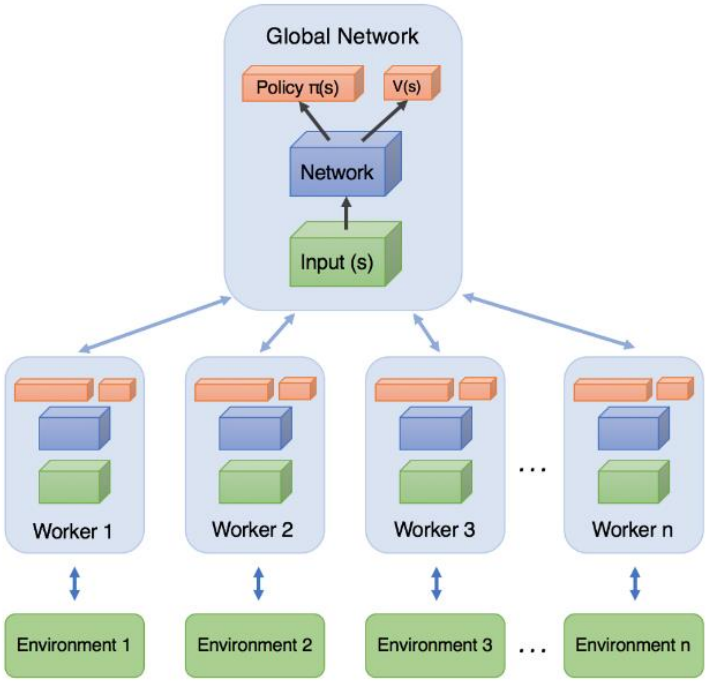
$Q(s, a)= V(s)+ A(s,a) => A(s,a) =Q(s,a) -V(s) => A(s,a)= r+ \gamma V(s\_hat) -V(s)$

Advantage function captures how better an action is compared to the others at a given state, while as we know the value function captures how good it is to be at this state.

You guess where this is going,right? **Instead of having the critic to learn the Q values, we make him learn the Advantage values**. That way the evaluation of an action is based not only on how good the action is, but also how much better it can be. The advantage of the advantage function (see what I did here?) is that it reduces the high variance of policy networks and stabilize the model.

Asynchronous Advantage Actor-Critic (A3C)

A3C's released by DeepMind in 2016 and make a splash in the scientific community. It's simplicity, robustness, speed and the achievement of higher scores in standard RL tasks made policy gradients and DQN obsolete. The key difference from A2C is the Asynchronous part. A3C consists of **multiple independent agents**(networks) with their own weights, who interact with a different copy of the environment in parallel. Thus, they can explore a bigger part of the state-action space in much less time.



The agents (or workers) are trained in parallel and update periodically a global network, which holds shared parameters. The updates are not happening simultaneously and that's where the asynchronous comes from. After each update, the agents resets their parameters to those of the global network and continue their independent exploration and training for n steps until they update themselves again.

We see that the information flows not only from the agents to the global network but also between agents as each agent resets his weights by the global network, which has the information of all the other agents. Smart right?

Back to A2C

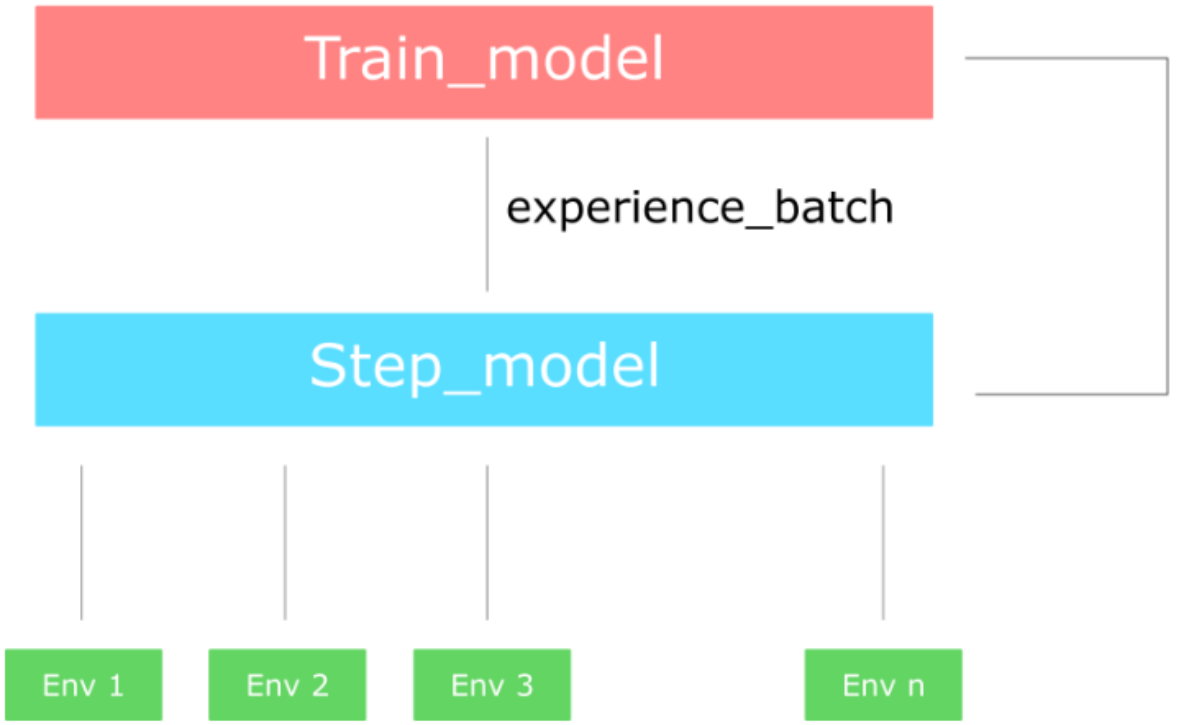
The main drawback of asynchrony is that some agents will be playing with an older version of the parameters. Of course, the update may not happen asynchronously but at the same time. In that case, we have an improved version of A2C with multiple agents instead of one. A2C will wait for all the agents to finish their segment and then update the global network weights and reset all the agents.

But. There is always a but. Some argue that there is no need to have many agents if they are synchronous, as they essentially are not different at all. And I agree. In fact, what we do, is to create **multiple versions of the environment** and just two networks.

The first network (usually referred to as step model) interacts with all the environments for n time steps in parallel and outputs a batch of experiences. With those experience, we train the

second network (train model) and we update the step model with the new weights. And we repeat the process.

If you are confused by the difference of A2C and A3C check out this Reddit post



https://medium.freecodecamp.org/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d

I tried to give you an intuitive explanation behind all these techniques without using much math and code, as things would be more complicated. However, they are not difficult models to implement as they rely on the same ideas as Policy Gradients and Deep Q Networks. If you want to build your own actor-critic model that plays Doom check this. And I think that you should. Only by building the thing ourselves , we can truly understand all the aspects, tricks and benefits of the model.

By the way, I take this opportunity to mention the recently open sourced library by Deepmind called trfl. It exposes several useful building blocks for implementing Reinforcement Learning agent, as they claim. I will try and come back to you for more details.

The idea of combining policy and value based method is now ,in 2018, considered standard for solving reinforcement learning problems. Most modern algorithms rely on actor-critics and expand this basic idea into more sophisticated and complex techniques. Some examples are :

Deep Deterministic Policy Gradients(DDPG),Proximal Policy Optimization (PPO), Trust Region Policy Optimization (TRPO).

6)Trust Region and Proximal policy optimization

Photo from Deepmind

Here, we going to take a step back and return to policy optimization in order to introduce two new methods: trust region policy optimization (TRPO) and proximal policy optimization (PPO). Remember that in policy gradients techniques, we try to optimize a policy objective function (the expected accumulative reward) using gradient descent. Policy gradients are great for continuous and large spaces but suffer from some problems.

- High variance (which we address with Actor-critic models)
- Delayed reward problem
- Sample inefficiency
- Learning rate highly affects training

Especially the last one troubled researchers for quite a long, because it is very hard to find a suitable learning rate for the whole optimization process. Small learning rate may cause vanishing gradients while large rate may cause exploding gradient. In general, we need a method to change the policy not too much but also not too little and even better to always improve our policy. One fundamental paper in this direction is :

Trust region policy optimization (TRPO)

To ensure that the policy won't move too far, we add a constraint to our optimization problem in terms of making sure that the updated policy lies within a trust region. Trust regions are defined as the region in which the local approximations of the function are accurate. Ok, but what does that mean? In trust regions, we determine the maximum step size and then we find the local maximum of the policy within the region. By continuing the same process iteratively, we find the global maximum. We can also expand or shrink the region based on how good the new approximation is. That way we are certain that the new policies can be trustworthy of not leading to dramatically bad policy degradation. We can express mathematically the above constraint using KL divergence(which you can think of as a distance between two probabilities distributions):

The KL divergence between the new and the old policy must be lower than the delta (δ), where delta is the size of the region. I could get into some math, but I think that this will only complicate things rather than clarify them. So essentially, we have just a **constrained optimization problem.**

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t[\mathrm{KL}[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]] \leq \delta. \end{aligned}$$

The question now is how we solve a constrained optimization problem? Using the Conjugate Gradient method. We can, of course, solve the problem analytically (natural gradient descent), but it is computational ineffective. If you dust of your knowledge on numerical mathematics, you might remember that the conjugate gradient method provides a numeric solution to a system of equations. That is way better from a computational perspective. So all we have to do is to approximate linearly the objective function and quadratically the constraint and let the conjugate gradient do its work.

To wrap it all up, the algorithm has the following steps:

- We run a set of trajectories and collect the policies
- Estimate the advantages using an advantage estimation algorithm
- Solve the constrained optimization problem using conjugate gradient
- Repeat

Generally speaking, trust regions are considered pretty standard methods to approach optimization problems. The tricky part is to apply them in a reinforcement learning context in a way that provides an advantage over simple policy gradients.

Although TRPO is a very powerful algorithm, it suffers from a significant problem: that bloody constraint, which adds additional overhead to our optimization problem. I mean it forces us to use the conjugate gradient method and baffled us with linear and quadratic approximations. Wouldn't it be nice if the could somehow include the constraint directly into our optimization objective? As you might have guessed that is exactly what Proximal policy optimization does.

Proximal policy optimization (PPO)

This simple idea gave us a quite simpler and more intuitive algorithm than TRPO. And it turns out that it outperforms many of the existing techniques most of the time. So instead of adding a constraint separately, we incorporate it inside the objective function as a penalty (we subtract the KL divergence times a constant C from the function).

$$\underset{\theta}{\text{maximize}} \sum_{n=1}^{N} \frac{\pi_\theta(a_n \mid s_n)}{\pi_{\theta_{\text{old}}}(a_n \mid s_n)} \hat{A}_n - C \cdot \overline{\text{KL}}_{\pi_{\theta_{\text{old}}}}(\pi_\theta)$$

As soon as we do that there is no need to solve a constrained problem and we can instead use a simple stochastic gradient descent in the above function. And the algorithm is transformed as follows:

- We run a set of trajectories and collect the policies
- Estimate the advantages using an advantage estimation algorithm
- Perform stochastic gradient descent on the objective function for a certain number of epochs
- Repeat

A small caveat is that is hard to choose the coefficient *C* in a way that it works well over the whole course of optimization. To address that we update the coefficient based on how big or small the KL divergence is. If KL is too high, we increase it, or if it is too low, we decrease it.

So this is it? This is the famous proximal policy optimization? Actually no. Sorry about that. It turns out that the function described above is not the same as the original paper. The authors found a way to improve this penalized version into a new, more robust objective function.

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

Hey hey wait. What happened here? Actually, not much. Let me explain. One thing I intentionally left out so far is that fraction of probabilities over there, that seems to appear in TRPO also. Well, that is called importance sampling. We essentially have a new policy that we want to estimate and an old one that we use to collect the samples. With **importance sampling**, we can evaluate a new policy with the samples from the old one and improve sample efficiency. The ratio infers how different the two policies are and we denote is as r(theta).

Using this ratio, we can construct a new objective function to **clip the estimated advantage if the new policy is far away from the old one**. And that's exactly what the above equation does. If an action is a lot more likely under the new policy than the old one, we do not want to overdo the action update, so we clipped the objective function. If it is much less likely under the new policy than the old, the objective action is flattened out to prevent from overdoing the update once again.

It may be just me, but I think I haven't come across a simpler and cleaner reinforcement learning algorithm in a while.

If you want to get into the rabbit hole, check out the baseline code from OpenAI, which will give you a crystal clear image of the whole algorithm and solve all your questions. Of course, it would be much better if you try to implement it from scratch all by yourself. It is not the easiest task, but why not.

I think that's it for now.

Resources

Before we wrap up, I'd like to provide you with some resources and courses that will help you continue your journey in Reinforcement Learning.

These are the best of the best (at least as far as I know)

- Reinforcement Learning Specialization by the University of Alberta (Coursera).
- Reinforcement Learning Explained by Microsoft
- Advanced AI: Deep Reinforcement Learning Course in Python
- The Cutting-Edge AI: Deep Reinforcement Learning in Python

Wrapping up

I hope this course has given you enough knowledge to proceed your endeavor into Reinforcement Learning. I try to keep things as simple as possible, but it wasn't always easy.

Closing I like to mention that there is so much more to learn about Reinforcement Learning. There is a tremendous progress over the past years with new research papers coming out each month.

Some of the topics you may want to look into are:

- Deep Deterministic Policy Gradients
- Soft Actor-Critic
- Inverse RL
- Monte Carlo Tree Search
- Curiosity driven Learning

It is such an amazing field and we have much more to see in the next years. And why not by one of you.

If anyone has any question or want to reach out, you can contact us here: https://theaisummer.com/contact/

And don't forget. AI Summer will continue to provide free content around Artificial Intelligence, Machine Learning and more. So, stay in touch.

Thank you

Best,

Sergios (AI Summer)