- Anúncios
- Tutoriais
- DL
- ML
- cv
- PNL
- 3D
- Seja pago para escrever
- Estamos contratando!

Mais

- Gradient Docs
- Demonstração de ML
- AI Wiki
- Fórum da Comunidade
- Centro de ajuda do Paperspace
- Contactar Vendas

Introdução ao OpenAI Gym: criando ambientes de academia personalizados

Inscrever-se Entrar

Procurar Pesquise em todo o nosso

Pressione Enter para pesquisar

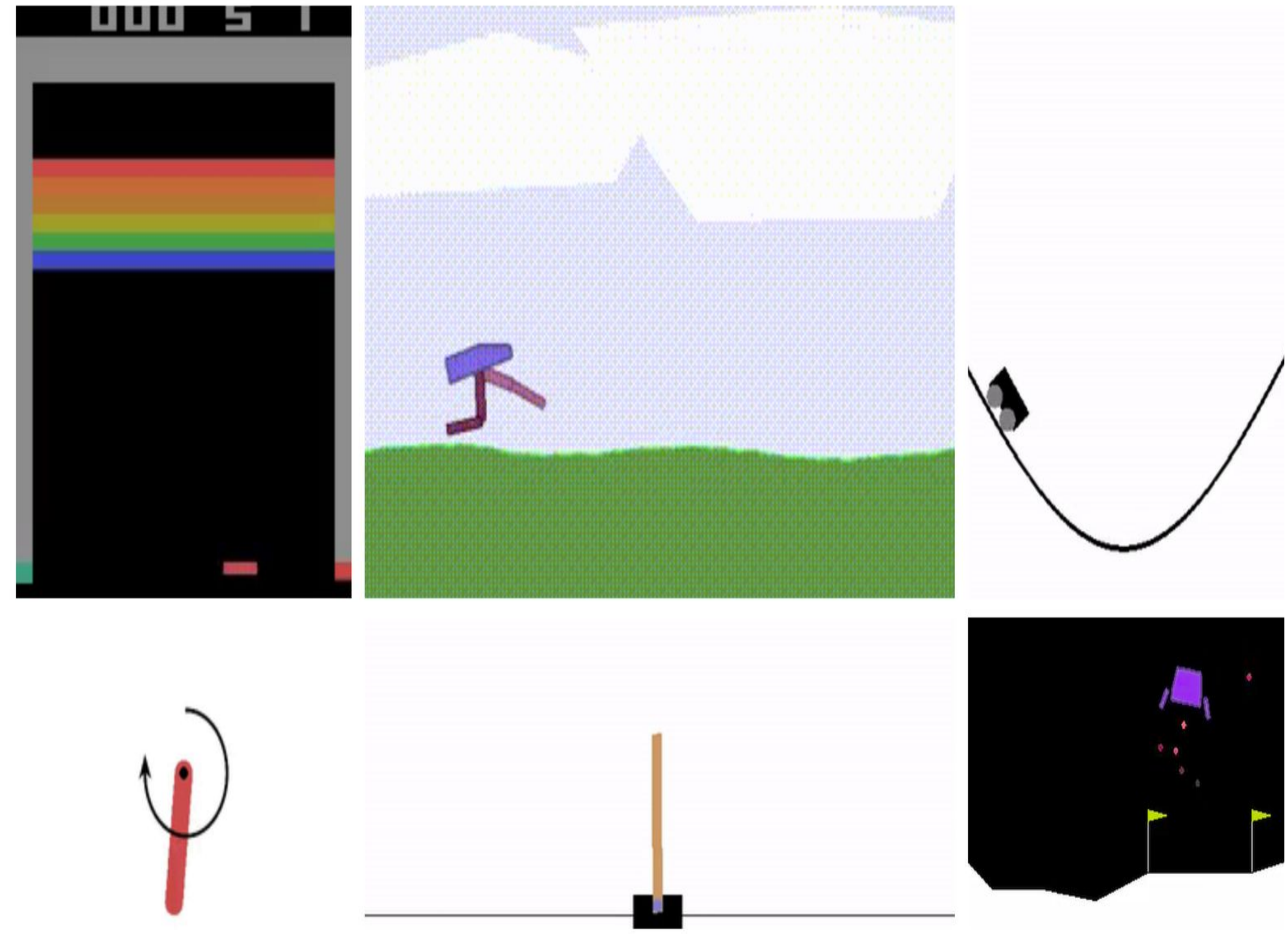
Aprendizagem por Reforço

Introdução ao OpenAI Gym: criando ambientes de academia personalizados

Esta postagem aborda como implementar um ambiente personalizado no OpenAI Gym. Como exemplo, implementamos um ambiente personalizado que envolve voar um Chopper (ou um helicóptero), evitando obstáculos no ar.

9 meses atrás • 13 min de leitura

Por Ayoosh Kathuria



O OpenAI Gym vem com vários ambientes incríveis, desde ambientes com tarefas de controle clássicas até aqueles que permitem treinar seus agentes para jogar jogos Atari como Breakout, Pacman e Seaquest. No entanto, você ainda pode ter uma tarefa em mãos que exige a criação de um ambiente personalizado que não faz parte do pacote Gym. Felizmente, o Gym é flexível o suficiente para permitir que você faça isso e esse é precisamente o assunto deste post.

Neste post, iremos projetar um ambiente personalizado que envolverá voar um Chopper (ou um helicóptero), evitando obstáculos no ar. Observe que esta é a segunda parte da série Open AI Gym, e o conhecimento dos conceitos introduzidos na Parte 1 é considerado um pré-requisito para esta postagem. Portanto, se você não leu a Parte 1, aqui está o link .

Você também pode executar todo o código deste tutorial em uma GPU gratuita com um Gradient Community Notebook .

Dê vida a este projeto

Executar em gradiente

Dependências / Importações

Começamos primeiro com a instalação de algumas dependências importantes.

```
!pip install opencv-python
!pip install pillow
```

Também começamos com as importações necessárias.

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import PIL.Image as Image
import gym
import random

from gym import Env, spaces
import time

font = cv2.FONT_HERSHEY_COMPLEX_SMALL
```

Descrição do Meio Ambiente

O ambiente que estamos criando é basicamente um jogo fortemente inspirado no jogo Dino Run, aquele que você joga no Google Chrome quando está desconectado da Internet. Há um dinossauro e você tem que pular sobre os cactos e evitar bater nos pássaros. A distância percorrida representa a recompensa que você acaba recebendo.

Em nosso jogo, em vez de um dinossauro, nosso agente será um piloto de Chopper.

1. O helicóptero deve cobrir a maior distância possível para obter a recompensa máxima. Haverá pássaros que o helicóptero terá que evitar.
2. O episódio termina em caso de colisão com um pássaro. O episódio também pode terminar se o Chopper ficar sem combustível.
3. Assim como os pássaros, existem tanques de combustível flutuantes (sim, nenhum ponto por estar perto da realidade, eu sei!) Que o Chopper pode coletar para reabastecer o helicóptero em sua capacidade total (que está fixada em 1000 L).

Observe que isso será apenas uma prova de conceito e não o jogo mais esteticamente agradável. Porém, caso você queira aprimorá-lo, este post vai deixar você com conhecimento suficiente para fazê-lo!

A primeira consideração ao projetar um ambiente é decidir que tipo de espaço de observação e espaço de ação iremos usar.

- O espaço de observação pode ser contínuo ou discreto. Um exemplo de um espaço de ação discreto é o de um mundo em grade onde o espaço de observação é definido por células, e o agente poderia estar dentro de uma dessas células. Um exemplo de espaço de ação contínua é aquele em que a posição do agente é descrita por coordenadas de valor real.
- O espaço de ação pode ser contínuo ou discreto também. Um exemplo de espaço discreto é aquele em que cada ação corresponde ao comportamento particular do agente, mas esse comportamento não pode ser quantificado. Um exemplo disso é Mario Bros , onde cada ação levaria a mover para a esquerda, direita, pular, etc. Suas ações não podem quantificar o comportamento que está sendo produzido, ou seja, você pode pular, mas não pular alto, alto ou baixo. No entanto, em um jogo como o Angry Birds, você decide quanto esticar o estilingue (você quantifica).

Classe ChopperScape

Começamos minha implementação da `__init__`função de nossa classe de ambiente `ChopperScape`,. Na `__init__`função, vamos definir os espaços de observação e ação. Além disso, também implementaremos alguns outros atributos:

1. `canvas`: This represents our observation image.
2. `x_min, y_min, x_max, y_max`: This defines the legitimate area of our screen where various elements of the screen, such as the Chopper and birds, can be placed. Other areas are reserved for displaying info such as fuel left, rewards, and padding.
3. `elements`: This stores the active elements stored in the screen at any given time (like chopper, bird, etc.)
4. `max_fuel`: Maximum fuel that the chopper can hold.

```
class ChopperScape(Env):
    def __init__(self):
        super(ChopperScape, self).__init__()



        # Define a 2-D observation space
        self.observation_shape = (600, 800, 3)
        self.observation_space = spaces.Box(low = np.zeros(self.observation_shape),
                                            high = np.ones(self.observation_shape),
                                            dtype = np.float16)


        # Define an action space ranging from 0 to 4
        self.action_space = spaces.Discrete(6,)

        # Create a canvas to render the environment images upon
        self.canvas = np.ones(self.observation_shape) * 1

        # Define elements present inside the environment
        self.elements = []

        # Maximum fuel chopper can take at once
        self.max_fuel = 1000

        # Permissible area of helicper to be
        self.y_min = int (self.observation_shape[0] * 0.1)
        self.x_min = 0
        self.y_max = int (self.observation_shape[0] * 0.9)
        self.x_max = self.observation_shape[1]
```

Elements of the Environment

Once we have determined the action space and the observation space, we need to finalize what would be the elements of our environment. In our game, we have three distinct elements: the Chopper, Flying Birds, and and Floating Fuel Stations. We will be implementing all of these as separate classes that inherit from a common base class called `Point`.

Point Base Class

The `Point` class is used to define any arbitrary point on our observation image. We define this class with the following attributes:

- `(x,y)`: Position of the point on the image.
- `(x_min, x_max, y_min, y_max)`: Permissible coordinates for the point. If we try to set the position of the point outside these limits, the position values are clamped to these limits.
- `name`: Name of the point.

We define the following member functions for this class.

- `get_position`: Get the coordinates of the point.
- `set_position`: Set the coordinates of the point to a certain value.
- `move`: Move the points by certain value.

```
class Point(object):
    def __init__(self, name, x_max, x_min, y_max, y_min):
        self.x = 0
        self.y = 0
        self.x_min = x_min
        self.x_max = x_max
        self.y_min = y_min
        self.y_max = y_max
        self.name = name

    def set_position(self, x, y):
        self.x = self.clamp(x, self.x_min, self.x_max - self.icon_w)
        self.y = self.clamp(y, self.y_min, self.y_max - self.icon_h)

    def get_position(self):
        return (self.x, self.y)

    def move(self, del_x, del_y):
        self.x += del_x
        self.y += del_y

        self.x = self.clamp(self.x, self.x_min, self.x_max - self.icon_w)
        self.y = self.clamp(self.y, self.y_min, self.y_max - self.icon_h)

    def clamp(self, n, minn, maxn):
        return max(min(maxn, n), minn)
```

Now we define the classes `Chopper`, `Bird` and `Fuel`. These classes are derived from the `Point` class, and introduce a set of new attributes:

- `icon`: Icon of the point that will display on the observation image when the game is rendered.
- `(icon_w, icon_h)`: Dimensions of the icon.

If you are viewing the Gradient notebook, the images used for icons are hosted along with the notebook.

```
class Chopper(Point):
    def __init__(self, name, x_max, x_min, y_max, y_min):
        super(Chopper, self).__init__(name, x_max, x_min, y_max, y_min)
        self.icon = cv2.imread("chopper.png") / 255.0
        self.icon_w = 64
        self.icon_h = 64
        self.icon = cv2.resize(self.icon, (self.icon_h, self.icon_w))


class Bird(Point):
    def __init__(self, name, x_max, x_min, y_max, y_min):
        super(Bird, self).__init__(name, x_max, x_min, y_max, y_min)
        self.icon = cv2.imread("bird.png") / 255.0
        self.icon_w = 32
        self.icon_h = 32
        self.icon = cv2.resize(self.icon, (self.icon_h, self.icon_w))

class Fuel(Point):
    def __init__(self, name, x_max, x_min, y_max, y_min):
        super(Fuel, self).__init__(name, x_max, x_min, y_max, y_min)
        self.icon = cv2.imread("fuel.png") / 255.0
        self.icon_w = 32
        self.icon_h = 32
        self.icon = cv2.resize(self.icon, (self.icon_h, self.icon_w))
```

Back to the ChopperScape Class

Recall from Part 1 that any gym `Env` class has two important functions:

1. `reset`: Resets the environment to its initial state and returns the initial observation.
2. `step` : Executes a step in the environment by applying an action. Returns the new observation, reward, completion status, and other info.

In this section, we will be implementing the `reset` and `step` functions of our environment along with many other helper functions. We begin with `reset` function.

Reset Function

When we reset our environment, we need to reset all the state-based variables in our environment. These include things like fuel consumed, episodic return, and the elements present inside the environment.

In our case, when we reset our environment, we have nothing but the Chopper in the initial state. We initialize our chopper randomly in an area in the top-left of our image. This area is 5-10 percent of the image width and 15-20 percent of the image height.

We also define a helper function called `draw_elements_on_canvas` that basically places the icons of each of the elements present in the game at their respective positions in the observation image. If the position is beyond the permissible range, then the icons are placed on the range boundaries. We also print important information such as the remaining fuel.

We finally return the canvas on which the elements have been placed as the observation.

```
%%add_to ChopperScape

def draw_elements_on_canvas(self):
    # Init the canvas
    self.canvas = np.ones(self.observation_shape) * 1

    # Draw the heliopter on canvas
    for elem in self.elements:
        elem_shape = elem.icon.shape
        x,y = elem.x, elem.y
        self.canvas[y : y + elem_shape[1], x:x + elem_shape[0]] = elem.icon

    text = 'Fuel Left: {} | Rewards: {}'.format(self.fuel_left, self.ep_return)
    # Put the info on canvas
    self.canvas = cv2.putText(self.canvas, text, (10,20), font,
               0.8, (0,0,0), 1, cv2.LINE_AA)

def reset(self):
    # Reset the fuel consumed
    self.fuel_left = self.max_fuel

    # Reset the reward
    self.ep_return  = 0

    # Number of birds
    self.bird_count = 0
    self.fuel_count = 0

    # Determine a place to intialise the chopper in
    x = random.randrange(int(self.observation_shape[0] * 0.05), int(self.observation_shape[0] * 0.10))
    y = random.randrange(int(self.observation_shape[1] * 0.15), int(self.observation_shape[1] * 0.20))

    # Intialise the chopper
    self.chopper = Chopper("chopper", self.x_max, self.x_min, self.y_max, self.y_min)
    self.chopper.set_position(x,y)

    # Intialise the elements
    self.elements = [self.chopper]

    # Reset the Canvas
    self.canvas = np.ones(self.observation_shape) * 1

    # Draw elements on the canvas
    self.draw_elements_on_canvas()


    # return the observation
    return self.canvas
```

Before we proceed further, let us now see what our initial observation looks like.

```
env = ChopperScape()
obs = env.reset()
plt.imshow(obs)
```

Since our observation is the same as the gameplay screen of the game, our render function shall return our observation too. We build functionality for two modes, one `human` which would render the game in a pop-up window, while `rgb_array` returns it as a pixel array.

```
%%add_to ChopperScape

def render(self, mode = "human"):
    assert mode in ["human", "rgb_array"], "Invalid mode, must be either \"human\" or \"rgb_array\""
    if mode == "human":
        cv2.imshow("Game", self.canvas)
        cv2.waitKey(10)

    elif mode == "rgb_array":
        return self.canvas

def close(self):
    cv2.destroyAllWindows()

env = ChopperScape()
obs = env.reset()
screen = env.render(mode = "rgb_array")
plt.imshow(screen)
```

Step Function

Now that we have the `reset` function out of the way, we begin work on implementing the `step` function, which will contain the code to transition our environment from one state to the next given an action. In many ways, this section is the proverbial meat of our environment, and this is where most of the planning goes.

We first need to enlist things that need to happen in one transition step of the environment. This can be basically broken down into two parts:

1. Applying actions to our agent.
2. Everything else that happens in the environments, such as behaviour of the non-RL actors (e.g. birds and floating gas stations).

So let's first focus on (1). We provide actions to the game that will control what our chopper does. We basically have 5 actions, which are move right, left, down, up, or do nothing, denoted by 0, 1, 2, 3, and 4, respectively.

We define a member function called `get_action_meanings()` that will tell us what integer each action is mapped to for our reference.

```
%%add_to ChopperScape

def get_action_meanings(self):
    return {0: "Right", 1: "Left", 2: "Down", 3: "Up", 4: "Do Nothing"}
```

We also validate whether the action being passed is a valid action or not by checking whether it's present in the action space. If not, we raise an assertion.

```
# Assert that it is a valid action
assert self.action_space.contains(action), "Invalid Action"
```

Once that is done, we accordingly change the position of the chopper using the `move` function we defined earlier. Each action results in movement by 5 coordinates in the respective directions.

```
# apply the action to the chopper
if action == 0:
    self.chopper.move(0,5)
elif action == 1:
    self.chopper.move(0,-5)
elif action == 2:
    self.chopper.move(5,0)
elif action == 3:
    self.chopper.move(-5,0)
elif action == 4:
    self.chopper.move(0,0)
```

Now that we have taken care of applying the action to the chopper, we focus on the other elements of the environment:

- Birds spawn randomly from the right edge of the screen with a probability of 1% (i.e. a bird is likely to appear on the right edge once every hundred frames). The bird moves 5 coordinate points every frame to the left. If they hit the Chopper the game ends. Otherwise, they disappear from the game once they reach the left edge.
- Fuel tanks spawn randomly from the bottom edge of the screen with a probability of 1 % (i.e. a fuel tank is likely to appear on the bottom edge once every hundred frames). The bird moves 5 co-ordinates up every frame. If they hit the Chopper, the Chopper is fuelled to its full capacity. Otherwise, they disappear from the game once they reach the top edge.

In order to implement the features outlined above, we need to implement a helper function that helps us determine whether two `Point` objects (such as a Chopper/Bird, Chopper/Fuel Tank) have collided or not. How do we define a collision? We say that two points have collided when the distance between the coordinates of their centers is less than half of the sum of their dimensions. We call this function `has_collided`.

```
%%add_to ChopperScape

def has_collided(self, elem1, elem2):
    x_col = False
    y_col = False

    elem1_x, elem1_y = elem1.get_position()
    elem2_x, elem2_y = elem2.get_position()

    if 2 * abs(elem1_x - elem2_x) <= (elem1.icon_w + elem2.icon_w):
        x_col = True

    if 2 * abs(elem1_y - elem2_y) <= (elem1.icon_h + elem2.icon_h):
        y_col = True

    if x_col and y_col:
        return True

    return False
```

Apart from this, we have to do some book-keeping. The reward for each step is 1, therefore, the episodic return counter is updated by 1 every episode. If there is a collision, the reward is -10 and the episode terminates. The fuel counter is reduced by 1 at every step.

Finally, we implement our `step` function. I've wrote extensive comments to guide you through it.

```
%%add_to ChopperScape

def step(self, action):
    # Flag that marks the termination of an episode
    done = False

    # Assert that it is a valid action
    assert self.action_space.contains(action), "Invalid Action"

    # Decrease the fuel counter
    self.fuel_left -= 1

    # Reward for executing a step.
    reward = 1

    # apply the action to the chopper
    if action == 0:
        self.chopper.move(0,5)
    elif action == 1:
        self.chopper.move(0,-5)
    elif action == 2:
        self.chopper.move(5,0)
    elif action == 3:
        self.chopper.move(-5,0)
    elif action == 4:
        self.chopper.move(0,0)

    # Spawn a bird at the right edge with prob 0.01
    if random.random() < 0.01:

        # Spawn a bird
        spawned_bird = Bird("bird_{}".format(self.bird_count), self.x_max, self.x_min, self.y_max, self.y_min)
        self.bird_count += 1

        # Compute the x,y co-ordinates of the position from where the bird has to be spawned
        # Horizontally, the position is on the right edge and vertically, the height is randomly
        # sampled from the set of permissible values
        bird_x = self.x_max
        bird_y = random.randrange(self.y_min, self.y_max)
        spawned_bird.set_position(self.x_max, bird_y)

        # Append the spawned bird to the elements currently present in Env.
        self.elements.append(spawned_bird)

    # Spawn a fuel at the bottom edge with prob 0.01
    if random.random() < 0.01:
        # Spawn a fuel tank
        spawned_fuel = Fuel("fuel_{}".format(self.bird_count), self.x_max, self.x_min, self.y_max, self.y_min)
        self.fuel_count += 1

        # Compute the x,y co-ordinates of the position from where the fuel tank has to be spawned
        # Horizontally, the position is randomly chosen from the list of permissible values and
        # vertically, the position is on the bottom edge
        fuel_x = random.randrange(self.x_min, self.x_max)
        fuel_y = self.y_max
        spawned_fuel.set_position(fuel_x, fuel_y)

        # Append the spawned fuel tank to the elemetns currently present in the Env.
        self.elements.append(spawned_fuel)

    # For elements in the Ev
    for elem in self.elements:
        if isinstance(elem, Bird):
            # If the bird has reached the left edge, remove it from the Env
            if elem.get_position()[0] <= self.x_min:
                self.elements.remove(elem)
            else:
                # Move the bird left by 5 pts.
                elem.move(-5,0)

            # If the bird has collided.
            if self.has_collided(self.chopper, elem):
                # Conclude the episode and remove the chopper from the Env.
                done = True
                reward = -10
                self.elements.remove(self.chopper)

        if isinstance(elem, Fuel):
            # If the fuel tank has reached the top, remove it from the Env
            if elem.get_position()[1] <= self.y_min:
                self.elements.remove(elem)
            else:
                # Move the Tank up by 5 pts.
                elem.move(0, -5)

            # If the fuel tank has collided with the chopper.
            if self.has_collided(self.chopper, elem):
                # Remove the fuel tank from the env.
                self.elements.remove(elem)

                # Fill the fuel tank of the chopper to full.
                self.fuel_left = self.max_fuel

    # Increment the episodic return
    self.ep_return += 1

    # Draw elements on the canvas
    self.draw_elements_on_canvas()

    # If out of fuel, end the episode.
    if self.fuel_left == 0:
        done = True

    return self.canvas, reward, done, []
```

Seeing It in Action

This concludes the code for our environment. Now execute some steps in the environment using an agent that takes random actions!

```
from IPython import display

env = ChopperScape()
obs = env.reset()


while True:
    # Take a random action
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)

    # Render the game
        env.render()

    if done == True:
        break

env.close()
```

Rendering the Environment

Conclusion

That's it for this part, folks. I hope this tutorial gave you some insight into some of the considerations and design decisions that go into designing a custom OpenAI environment. You can now try creating an environment of your choice, or if you're so inclined, you can make several improvements to the one we just designed for practice. Some suggestions right off the bat are:

1. Instead of the episode terminating upon the first bird strike, you can implement multiple lives for the chopper.
2. Design an evil alien race of mutated birds that are also able to fire missiles at the chopper, and the chopper has to avoid them.
3. Do something about when a fuel tank and a bird collides!

With these suggestions, it's a wrap. Happy coding!

Start Discussion 0 replies

- Tags:
- Reinforcement Learning
- OpenAI Gym
- Tutorial

Spread the word

- Share
- Tweet
- Share
- Copy
- Email

https://blog.paperspace.com/

público

Next article

Gradient Notebooks just got the biggest update since 2019

público

Previous article

Introduction to Time Series Analysis

Keep reading

público

The Machine Learning Practitioner's Guide to Reinforcement Learning: Overview of the RL Universe

10 months ago • 17 min read

público

The Machine Learning Practitioner's Guide to Reinforcement Learning: All About Markov Decision Processes

10 months ago • 13 min read

público

Introdução ao OpenAI Gym: os blocos de construção básicos

10 meses atrás • 13 min de leitura

Links principais

- Anúncios
- Tutoriais
- DL
- ML
- cv
- PNL
- 3D
- Seja pago para escrever
- Estamos contratando!

Links secundários

- Gradient Docs
- Demonstração de ML
- AI Wiki
- Fórum da Comunidade
- Centro de ajuda do Paperspace
- Contactar Vendas

Links sociais

- Facebook
- Twitter