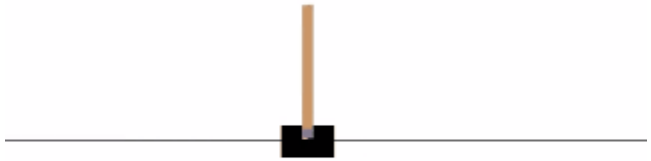
[深度强化学习] blog翻译-使用Keras与Gym仿真环境进行深度Q学习（DQL）

via:https://keon.io/rl/deep-q-learning-with-keras-and-gym/



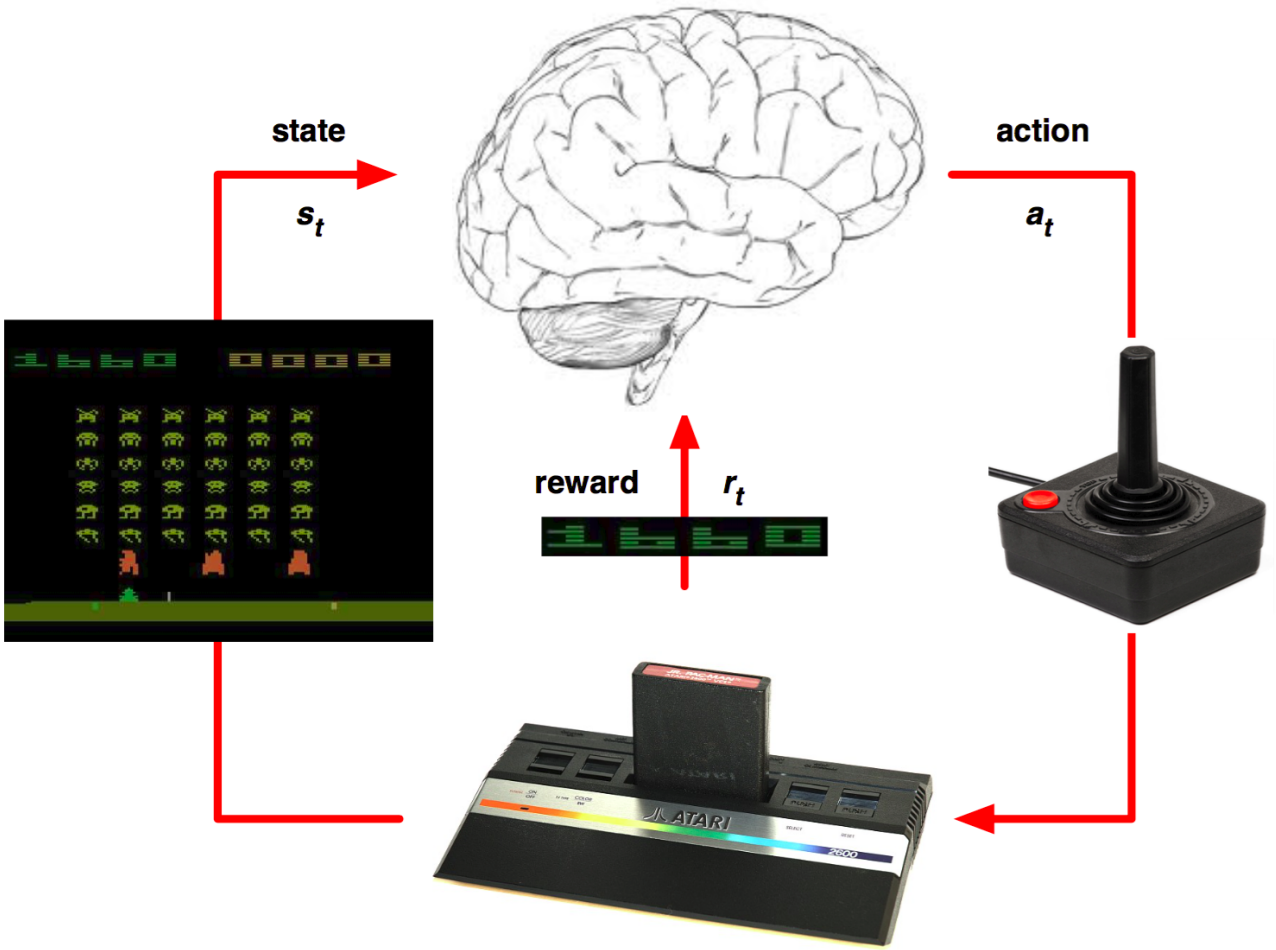
综述

这篇blog将会展示深度强化学习（深度Q学习）是如何使用Keras与Gym环境使机器学会玩CartPole游戏的。只有78行代码哦

我将会解释一切，不需要你对强化学习有任何的先决知识。

这篇文章中使用的代码的仓库在这里：GitHub

强化学习

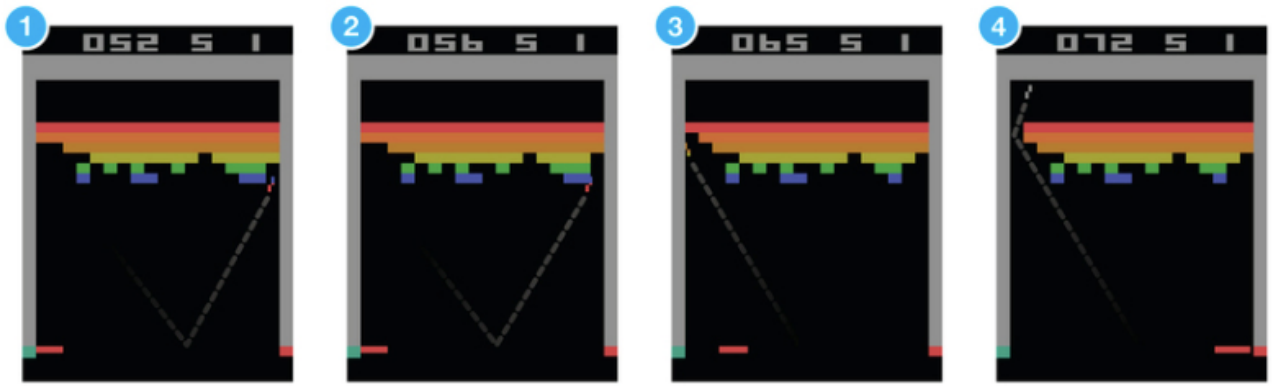


强化学习是一种允许你创造能从环境中交互学习的AI agent 的机器学习算法。就跟我们学习骑自行车一样，这种类型的AI通过试错来学习。如上图所示，大脑代表AI

agent并在环境中活动。当每次行动过后，agent接收到环境反馈。反馈包括回报（reward）和环境的下个状态（state）。回报由模型设计者定义。如果类比人类学习自行车，我们会将车从起始点到当前位置的距离定义为回报。

深度强化学习

2013年，在DeepMind 发表的著名论文*Playing Atari with Deep Reinforcement Learning*中，他们介绍了一种新算法，深度Q网络（DQN）。文章展示了AI agent如何在没有任何先验信息的情况下通过观察屏幕学习玩游戏。结果令人印象深刻。这篇文章开启了被我们成为“深度强化学习”的新时代。这种学习算法是混合了深度学习与强化学习的新算法。



在Q学习算法中，有一种函数被称为Q函数，它用来估计基于一个状态的回报。同样地，在DQN中，我们使用一个神经网络估计基于状态的回报函数。我们将在之后细致地讨论这一部分工作。

3. Re:[如何构建自己的轮式移动机器人系统·从入门到放弃]机器人底层篇

@hellotoyou不好意思，我还是学生，暂时不打算接社会项目...

--63445538

4. Re:[如何构建自己的轮式移动机器人系统·从入门到放弃]机器人底层篇

重要紧急的项目，请认真对待，谢谢~最重要的是有钱。

--hellotoyou

5. Re:[如何构建自己的轮式移动机器人系统·从入门到放弃]机器人底层篇

您好、有业务咨询您，请联系QQ2605286905。大项目哦

--hellotoyou

阅读排行榜

1. [MATLAB&SIMULINK] 如何提取并处理Simscape Power System 中powergui的谐波分析数据(1955)
2. [如何构建自己的轮式移动机器人系统·从入门到放弃]机器人底层篇(1932)
3. [深度强化学习] blog翻译-使用Keras与Gym仿真环境进行深度Q学习（DQL）(928)
4. Readme.txt(42)
5. 招人(41)

评论排行榜

1. [如何构建自己的轮式移动机器人系统·从入门到放弃]机器人底层篇(5)

Cartpole游戏

通常训练一个agent玩Atari游戏通常会好一会儿（从几个小时到一天）。所以我们将训练agent玩一个简单的游戏，CartPole，并使用在上面论文中的一些思想。

CartPole是OpenAI gym中最简单的一个环境。正如你在文章一开始看到的那个gif一样，CartPole的目的就是杆子平衡在移动的小车上。除了像素信息，还有四种信息可以用作状态，像是，杆子的角度和车在滑轨的位置。agent可以通过施加左（0）或右（1）的动作，使小车移动。

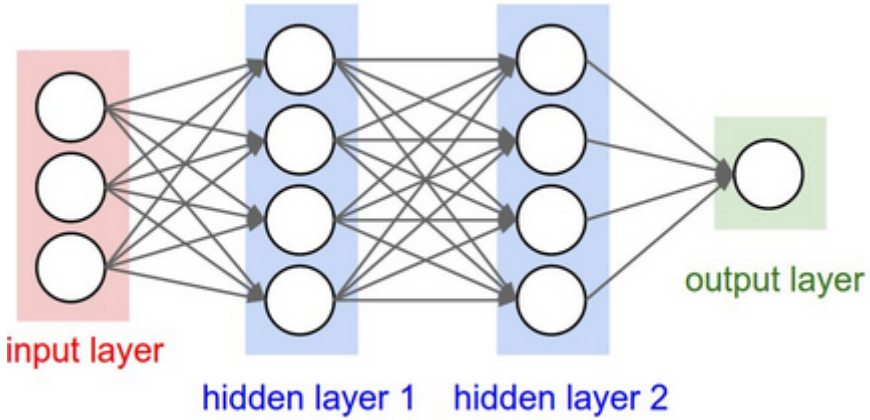
Gym使游戏环境的交互非常方便：

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

如我们上面所说，action要么是0要么是1。当我们将这些数字串入环境中将会得出结果。“env”是游戏环境类。“done”为标记游戏结束与否的布尔量。当前状态“state”，“action”，“next_state”与“reward”是我们用于训练agent的数据。

使用Keras实现简单神经网络

这篇文章不是关于深度学习或神经网络的。所以我们将神经网络试做黑箱算法。这个算法的功能是从成对的输入与输出数据学习某种模式并且可以基于不可见的输入数据预测输出。但是我们应该理解在DQN算法中的那部分神经网络算法。



注意到我们使用的神经网络类似于上图所示的网络。我们使用一个包含四种输入信息的输入层和三个隐藏层。但是我们在输出层有两个节点因为在这个游戏中有两个按钮（0与1）

keras库使基础神经网络的使用变得非常简单。下面的代码会生成一个空的神经网络模型。“activation”，“loss”与“optimizer”是定义神经网络特征的参数但我们不打算在这里讨论它们。

```
# Neural Net for Deep Q Learning

# Sequential() creates the foundation of the layers.
model = Sequential()

# Dense is the basic form of a neural network layer
# Input Layer 4 and Hidden Layer with 128 nodes
model.add(Dense(64, input_dim=4, activation='tanh'))
# Hidden layer with 128 nodes
```

```
model.add(Dense(128, activation='tanh'))
# Hidden layer with 128 nodes
model.add(Dense(128, activation='tanh'))
# Output Layer with 2 nodes
model.add(Dense(2, activation='linear'))

# Create the model based on the information above
model.compile(loss='mse',
              optimizer=RMSprop(lr=self.learning_rate))
```

为了让模型可以基于环境数据理解与预测，我们不得不给它提供数据。下列代码所示，“fit()”方法为模型提供“states”和“target_f”信息。你可以忽视其余参数。

这个训练过程使模型从某个状态“state”预测回报函数值“target_f”。

```
model.fit(state, target_f, nb_epoch=1, verbose=0)
```

当你在模型调用"predict()"函数时，模型根据之前训练过的数据将预测现在状态的回报函数

```
prediction = model.predict(state)
```

实现深度Q算法（DQN）

DQN算法最重要的特征是记忆（remember）与回顾（replay）方法。它们都有很简明的概念。

记忆（remember）

对于DQN来说一个挑战就是运用在算法中的神经网络区域通过覆盖掉先前学习的经验来遗忘它们。所以我们需要记录下先前的经验与观察值以便再用这些先前数据训练模型。我们将调用代表经验的数组数据“memory”和“remember()”函数来添加状态，回报，和下次状态到“memory”中。

在本例中，“memory”列表中有以下形式的数据：

```
memory = [(state, action, reward, next_State)...]
```

“remember()”只是简单的存储上述这些数据：

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

“done”只是一个标记是否为最后一个状态的布尔量。

是不是很简单？？

回放（replay）

"replay()"从存储在“memory”中的数据（经验）中训练神经网络。首先，我们从“memory”中抽出部分数据并叫他们“bathces”

```
batches = min(batch_size, len(self.memory))
batches = np.random.choice(len(self.memory), batches)
```

上面的代码将打乱memory中的bathces的索引数。举个例子，如果batchce为[1,5,2,7]，每个数据代表在memory中的索引数1,5,2,7。

为了使agent在长期运行中表现的更好，我们不仅仅需要考虑即时回报（immediate rewards），还要考虑未来回报（future rewards）。为了实现这一目标，我们定义“discount rate”（折扣因子）即“gamma”。这样，agent将学习已有的状态然后想方设法最大化未来回报。

```
for i in batches:
    # Extract informations from i-th index of the memory
    state, action, reward, next_state = self.memory[i]

    # if done, make our target reward (-100 penality)
    target = reward

    if not done:
      # predict the future discounted reward
      target = reward + self.gamma * \
               np.amax(self.model.predict(next_state)[0])

    # make the agent to approximately map
    # the current state to future discounted reward
    # We'll call that target_f
    target_f = self.model.predict(state)
    target_f[0][action] = target

    # Train the Neural Net with the state and target_f
    self.model.fit(state, target_f, nb_epoch=1, verbose=0)
```

agent如何选择行为？

我们的agent在最初的一部分时间会随机选择行为，这被“exploration rate”或“epsilon”参数表征。这是因为在最初对agent最好的策略就是在他们掌握模式前尝试一切。当agent没有随机选择行为，它会基于当前状态预测回报值并且选择能够将回报最大化的行为。“np.argmax()”函数可以取出“act_values[0]”中的最大值。

```
def act(self, state):
    if np.random.rand() <= self.epsilon:
        # The agent acts randomly
        return env.action_space.sample()

    # Predict the reward value based on the given state
    act_values = self.model.predict(state)

    # Pick the action based on the predicted reward
    return np.argmax(act_values[0])
```

“act_values[0]”中的数据类似“[0.67, 0.2]”，每个数字分别代表0和1的回报，于是“argmax()”会取出更大数值所代表的的行为。比如在[0.67, 0.2]中，argmax()返回0因为0索引代表的数据的回报最大。

超参数

有一些超参数是强化学习agent所必需的，你会在下面一次又一次的看到这些参数。

·**episodes** 我们想让agent玩游戏的次数

·**gamma** discount rate（折扣因子），以便计算未来的折扣回报。

·**epsilon** exploration rate，这个比率表征一个agent随机选择行为的程度

·**epsilon_decay** 上述参数的衰减率。我们希望随着agent更擅长游戏的同时减少它探索的次数。

·**epsilon_min** 这个参数是我们希望agent采取的最少的探索次数。

·**learning_rata** 这个参数决定了神经网络在每次迭代时的学习率（学习程度）。

编写深度Q学习agent的代码

我在上面分别描述了agent算法的每一部分。下面的代码实现了上述讨论的一切并编写了一个整洁的类，叫做“DQNAgent”

```
# Deep-Q learning Agent
class DQNAgent:
    def __init__(self, env):
        self.env = env
        self.memory = []
        self.gamma = 0.9  # decay rate
        self.epsilon = 1  # exploration
        self.epsilon_decay = .995
        self.epsilon_min = 0.1
        self.learning_rate = 0.0001
        self._build_model()
```

```
    def _build_model(self):
        model = Sequential()
        model.add(Dense(128, input_dim=4, activation='tanh'))
        model.add(Dense(128, activation='tanh'))
        model.add(Dense(128, activation='tanh'))
        model.add(Dense(2, activation='linear'))
        model.compile(loss='mse',
                      optimizer=RMSprop(lr=self.learning_rate))
        self.model = model

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

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return env.action_space.sample()
        act_values = self.model.predict(state)
        return np.argmax(act_values[0])  # returns action

    def replay(self, batch_size):
        batches = min(batch_size, len(self.memory))
        batches = np.random.choice(len(self.memory), batches)
        for i in batches:
            state, action, reward, next_state, done = self.memory[i]
            target = reward
            if not done:
              target = reward + self.gamma * \
                       np.amax(self.model.predict(next_state)[0])
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, nb_epoch=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

让我们来训练它吧！

这部分会比较短，我会在注释中解释。

```
if __name__ == "__main__":

    # 为agent初始化gym环境参数
    env = gym.make('CartPole-v0')
    agent = DQNAgent(env)

    # 游戏的主循环
    for e in range(episodes):

        # 在每次游戏开始时复位状态参数
        state = env.reset()
        state = np.reshape(state, [1, 4])

        # time_t 代表游戏的每一帧
        # 我们的目标是使得杆子尽可能长地保持竖直朝上
        # time_t 越大，分数越高
        for time_t in range(5000):
            # turn this on if you want to render
            # env.render()

            # 选择行为
            action = agent.act(state)

            # 在环境中施加行为推动游戏进行
```

```
            next_state, reward, done, _ = env.step(action)
            next_state = np.reshape(next_state, [1, 4])

            # reward缺省为1
            # 在每一个agent完成了目标的帧agent都会得到回报
            # 并且如果失败得到-100
            reward = -100 if done else reward

            # 记忆先前的状态，行为，回报与下一个状态
            agent.remember(state, action, reward, next_state, done)

            # 使下一个状态成为下一帧的新状态
            state = copy.deepcopy(next_state)

            # 如果游戏结束done被置为ture
            # 除非agent没有完成目标
            if done:
                # 打印分数并且跳出游戏循环
                print("episode: {}/{}, score: {}"
                      .format(e, episodes, time_t))
                break
        # 通过之前的经验训练模型
        agent.replay(32)
```

结果

在一开始，agent通过随机行为探索游戏环境

```
episode: 1/5000, score: 27
episode: 2/5000, score: 11
episode: 3/5000, score: 34
episode: 4/5000, score: 33
episode: 5/5000, score: 8
episode: 6/5000, score: 22
episode: 7/5000, score: 47
episode: 8/5000, score: 22
episode: 9/5000, score: 54
episode: 10/5000, score: 16
```

算法会经过多个阶段训练agent

1.小车操作agent试图平衡杆子

2.但是出界，游戏结束

3.当它距离边界太近时它不得不移动小车，于是杆子掉了。

4.agent最后掌握了平衡并学会控制杆子。

经过几百个episodes的训练后，它开始学习如何最大化分数。

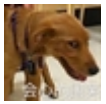
```
episode: 284/5000, score: 1331
episode: 285/5000, score: 124
episode: 286/5000, score: 259
episode: 287/5000, score: 138
episode: 288/5000, score: 170
episode: 289/5000, score: 13
episode: 290/5000, score: 365
episode: 291/5000, score: 1499
episode: 292/5000, score: 274
episode: 293/5000, score: 498
episode: 294/5000, score: 529
episode: 295/5000, score: 284
episode: 296/5000, score: 1355
episode: 297/5000, score: 911
episode: 298/5000, score: 1414
```

一个大师级CartPole玩家都诞生了。

文章中使用的代码在GitHub中。我为想要跳过训练的朋友提供了训练好的权重。

参考

- *Playing Atari with Deep Reinforcement Learning*
- Human-level Control Through Deep Reinforcement Learning



63445538
关注 - 1
粉丝 - 7

+加关注

« 上一篇：记录-2016年12月11日

» 下一篇：招人

posted @ 2017-02-24 20:44 63445538 阅读(927) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· Facebook认怂React专利，但问题依旧没有解决？
· 多款重量级产品亮相，Google认真做起硬件来连苹果都害怕
· 未向苹果征收130亿欧元税款 欧盟起诉爱尔兰政府
· 苹果要求三星向韩国监管机构施压严查高通
· 小米被摄影师指责盗用视频素材 未获授权即使用
» 更多新闻...