به نام خدا

پروژه هشتم درس یادگیری عمیق

آشنایی با یادگیری تقویتی عمیق

استاد درس: دکتر صفابخش

نگارش: زهرا اخلاقی

شماره دانشجویی: ۴۰۱۱۳۱۰۶۴

زمستان ۱۴۰۲

مدل پیاده‌سازی شده:

مدل پیاده سازی شده یک MLP دارای و لایه پنهان به اندازه ۱۶ و ۸ میباشد. در این مدل اندازه ورودی برابر با اندازه ویژگی‌های محیط و اندازه لایه خروجی برابر با تعداد عملکرد های ممکن میباشد.

در نهایت هدف این شبکه این است که در هر لحظه کاری را که بیشترین gain را دارد، احتمال بیشتری داشته باشد.

```
class MLP(torch.nn.Module):
    def __init__(self, input_size, hidden_sizes, output_size):
        super(MLP, self).__init__()
        layers = []
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            layers.append(torch.nn.Linear(sizes[i], sizes[i + 1]))
            if i < len(sizes) - 2:
                layers.append(torch.nn.ReLU())
        self.model = torch.nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)
```

```
model = MLP(number_observation_features, [16,8], number_actions)

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

num_epochs = 20
```

:Policy

این قسمت مشخص میکند که برای یک مشاهده در محیط، رفتار باید چگونه باشد. تابع زیر برای انجام این کار نوشته شده است، مدل MLP و حالت محیط را از ورودی می‌گیرد و سپس یک توزیع دسته‌بندی (چند جمله‌ای) را به عنوان خروجی دارد که می‌توان از آن برای انتخاب عمل مناسب در محیط استفاده کرد که به‌طور تصادفی با توجه به این احتمالات توزیع می‌شوند.

```
from torch.distributions.categorical import Categorical

def get_policy(model, observation):
    observation_tensor = torch.as_tensor(observation, dtype=torch.float32)
    logits = model(observation_tensor)

    # Categorical will also normalize the logits for us
    return Categorical(logits=logits)
```

:Action

در این مرحله عمل مناسب انتخاب میشود، برای انجام این کار در تابع زیر خروجی تابع get_policy را به عنوان ورودی میگیرد (همانطور که در قسمت قبل گفته شد خروجی تابع policy به صورت categorical است و احتمال هر عمل را مشخص میکند) و در این تابع عمل مناسب را انتخاب میکند و سپس لگاریتم احتمال آن عمل محاسبه میشود که میتواند برای محاسبه گرادیان استفاده شود، خروجی این تابع عمل مناسب و گرادیان محاسبه شده می‌باشد.

```
def get_action(policy):
    action = policy.sample()  # Unit tensor

    # Converts to an int, as this is what Gym environments require
    action_int = int(action.item())

    # Calculate the log probability of the action, which is required for
    # calculating the loss later
    log_probability_action = policy.log_prob(action)

    return action_int, log_probability_action
```

:Calculating The LOSS

مقدار loss با توجه به رابطه ی داده شده، در تابع زیر محاسبه می شود. تابع زیر حاصل جمع لگاریتم احتمالاتی محاسبه شده در هر اپیزود ضربدر دستاورد آن عمل میباشد.

```
def calculate_loss(epoch_log_probability_actions, epoch_action_rewards):
    return -(epoch_log_probability_actions * epoch_action_rewards).sum()
```

تابع train_one_epoch یک "epoch" از gradian policy را اجرا می کند که در آن عامل برای تعدادی از قسمت‌ها در محیط با استفاده از تابع policy عملی را انجام می‌دهد و به دنبال آن یک مرحله به روز رسانی با استفاده از تابع loss انجام میشود، هدف کلی آموزش یک مدل RL است که بتواند در یک محیط خاص به طور مستقل عمل کند و به اهداف مطلوب برسد.

در هر اپیزود:

1. مدل با محیط تعامل می کند و برای تعداد مشخصی از گام ها (timesteps) عمل می کند.
2. در هر گام، مدل یک عمل را بر اساس سیاست (احتمالات عمل) فعلی انتخاب می کند.
3. مدل پاداش را از محیط دریافت می کند و سیاست را بر اساس آن به روز می کند.
4. این فرآیند تا زمانی که اپیزود به پایان برسد (معمولاً زمانی که مدل به هدف خاصی می رسد یا به بن بست می رسد) ادامه می یابد.

پس از هر اپیزود:

1. عملکرد مدل بر اساس پاداش های جمع آوری شده ارزیابی می شود.
2. پارامترهای مدل بر اساس شیب سیاست به روز می شوند.
3. این فرآیند برای تعداد مشخصی از اپیزودها تکرار می شود تا زمانی که مدل به طور موثری یاد بگیرد که با محیط تعامل کند.

خروجی حاصل از اجرای ۱۰۰ ایپاک به صورت زیر است:

```
for epoch in range(num_epochs):
        average_return = train_one_epoch(env, model, optimizer,5000,200)
        if (epoch+1)%10==0:
           print('epoch: %3d \t return: %.3f' % (epoch+1, average_return))
```

```
epoch:  10       return: 23.806
epoch:  20       return: 24.335
epoch:  30       return: 26.010
epoch:  40       return: 28.680
epoch:  50       return: 27.656
epoch:  60       return: 32.096
epoch:  70       return: 35.338
epoch:  80       return: 39.148
epoch:  90       return: 44.786
epoch: 100       return: 49.554
```