



Универзитет „Св. Кирил и Методиј“ во Скопје
**ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И
КОМПЈУТЕРСКО ИНЖЕНЕРСТВО**

Факултет за информатички науки и компјутерско
инженерство, Скопје

Семинарска работа

Тема:

Тренирање на агент со Deep Reinforcement Learning
користејќи PPO со GAE и Impala архитектурата во Procgen
околина

Изработил:
Матеј Митев

Ментор:
Проф. Андреа Кулаков

Содржина

Вовед	3
Основни термини.....	3
Procgen околина.....	4
Actor-Critic метод.....	5
Конволуциска мрежа	5
Residual Convolutional Network	6
Actor-critic делот во конволуциската мрежа	7
Функции на загуба (Loss функции).....	7
Critic Loss (Mean Squared Error - MSE)	7
Advantage	7
Generalized Advantage Estimation (GAE)	8
Proximal Policy Optimization (PPO)	9
Процесот на ажурирање на мрежата	10
Собирање и чување на податоци	11
Помошни функции	12
Хиперпараметри.....	13
Главен циклус.....	14
Нормализација на adventage-от.....	15
Евалуација на моделот	15
Референци	16

Вовед

Во оваа семинарска работа ја истражувам примената на Deep Reinforcement Learning алгоритмите, конкретно Proximal Policy Optimization (PPO) заедно со Generalized Advantage Estimation (GAE), користејќи ја Impala архитектурата за тренирање на агент во Procgen околината. Оваа тема ја избраав бидејќи сакав подобро да ги запознаам можностите на Deep Reinforcement Learning и да го применам на реалистичен проблем, бидејќи прв пат ја проучувам оваа област. Исто така, целта ми беше да се запознаам со модерни техники кои овозможуваат стабилно и ефикасно тренирање на агенти кои учат од слики, што е актуелен предизвик во светот на вештачката интелигенција.

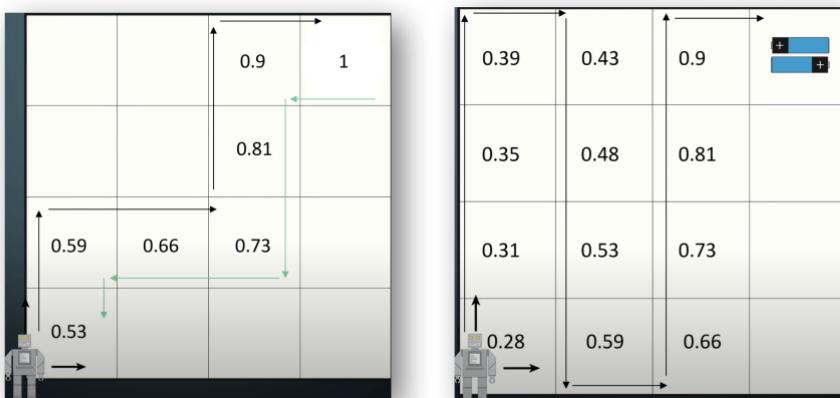
После тренирањето агентот научува сам да го помине ова ниво и да стигне до паричката, а во меѓувреме да не биде пресечен од сивите сечила.



Основни термини

- **State (состојба)** – претставува моменталната позиција во која се наоѓа агентот во околината. Во мојот случај, состојбата е претставена преку слика составена од пиксели која агентот ја добива од околината (играта). Врз основа на состојбата, агентот одлучува која акција ќе ја направи следно.
- **Trajectory (траекторија)** – е движењето на агентот во едно комплетно играње на играта, од почеток до крај. Тоа претставува низа од состојби, преземени акции и добиени награди од почетокот на играта па сè до нејзиниот крај.
- **Reward (награда)** – претставува моменталната повратна информација што агентот ја добива од околината во моменталната состојба. Наградата му кажува на агентот дали направил добра или лоша акција во конкретната состојба. На пример, во мојот случај, во околината Procgen, сите полиња низ кои поминува агентот имаат reward 0, а единствено достигнувањето на крајната цел има reward од 10.

- **Return** – е вредноста која се пропагира напазад низ траекторијата низ која се движел агентот. За различни траектории се добиваат различни returns.



Ова е пример за наједноставна игра. Тука во горниот десен ќош е представена награда 1. Таа награда се пропагира напазад низ состојбите низ кои поминал агентот. Во reinforcement learning, агентот има за цел да ги максимизира токму овие returns, односно да избере акција која има најголема вредност за return-от.

- **Value (вредност)** – претставува проценка или предвидување на тоа колкава е вредноста за return - от во одредена состојба. Наједноставна калкулација на value-то е средна вредност од сите returns во повеќе траектории. Но, ваквата едноставна проценка не е добра, бидејќи не ги зема во предвид сите карактеристики на состојбите, и притоа не е ефикасна за сложени игри каде што просторот на состојби е огромен или непрегледен. Во мојот код, оваа проценка ја прави невронската мрежа. Ваквата проценка помага агентот да одлучи дали состојбата во која се наоѓа моментално е поволна или не, и на тој начин го насочува кон оптимално однесување во иднина.

Procgen околина

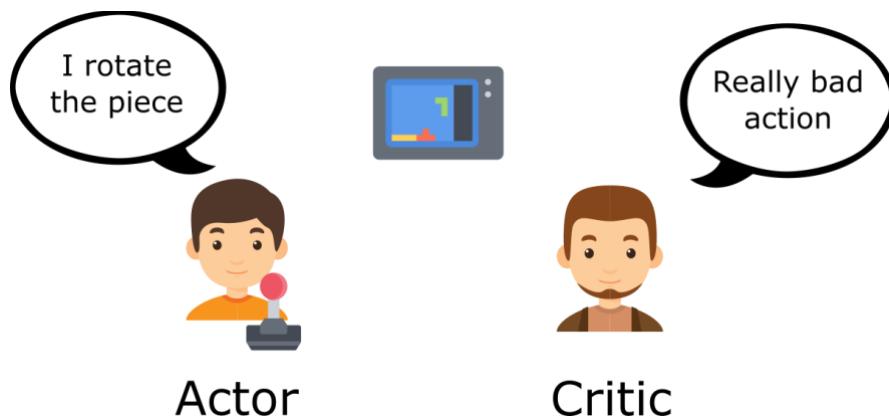
Јас го користам **Procgen** како околина за тренирање на агентот. Тоа е библиотека која содржи множество од различни едноставни игри, за кои има голем број на уникатни нивоа со различна тежина. Со ова, Procgen овозможува создавање на многу побогати и разновидни тренинг сценарија, со што агентот генерално полесно учи како да ги решава поставените задачи и станува поотпорен на overfitting. Исто така, Procgen ми овозможува значително побрзо тренирање на моделот, бидејќи поддржува паралелно извршување на повеќе независни инстанци од истата игра истовремено, што значително го забрзува процесот на собирање искуства и податоци за тренирање. За секоја состојба од играта, Procgen враќа единствено слика од пиксели(64x64) која го претставува моменталниот state, како и reward-от кој ја претставува моменталната награда. Reward-от што го враќа Procgen е единствената бројчена информација што агентот ја добива за да може да оцени дали напредува во играта или не. Во конкретниот случај во играта на која го тренирам агентот - „coinrun“, reward-от изнесува +10

кога агентот ќе стигне до целта(паричката), а во сите останати состојби е 0. Procgen не враќа негативен reward кога агентот ќе ја изгуби играта, туку едноставно ја означува состојбата како завршна состојба.

Actor-Critic метод

Actor-Critic е еден од најчесто користените методи во Deep Reinforcement Learning. Овој метод комбинира два различни пристапи за подобрување на учењето на агентот:

1. **Actor (Актер)** – Овој дел од моделот е одговорен за изборот на акција во дадена состојба. Тој учи политика (policy) која определува која акција треба да се избере во секој момент, базирајќи се на веројатносна дистрибуција. Излезот од Actor е листа со веројатности за сите можни акции, а финалната акција се бира со **sample()** од таа дистрибуција.
2. **Critic (Критичар)** – Овој дел од моделот ја оценува моменталната состојба на агентот, односно предвидува колкава е value вредноста за таа состојба. Critic делот му помага на Actor делот за подобро одлучување.



Конволуциска мрежа

Во овој проект ја користам IMPALA архитектурата, која е добро позната во полето на Deep Reinforcement Learning. Оваа архитектура е дизајнирана специјално за обработка на визуелни информации и ефикасно тренирање на агентите во сложени околини, како што е Procgen.

Користам конволуциска мрежа, бидејќи состојбата на играта е претставена како слика од пиксели, па неопходно е да користам конволуциска невронска мрежа за да ги извлечам релевантните карактеристики од сликата. Наместо агентот директно да го анализира секој пиксел, CNN автоматски учи кои делови од сликата се важни за носење на одлуки. Ова е многу важно во Reinforcement Learning, бидејќи овозможува агентот да разбере форми, движења и важни елементи од околината. Сите карактеристики што конволуциските слоеви ги извлекле од 64x64 пикセルната слика се трансформираат во компактен вектор кој

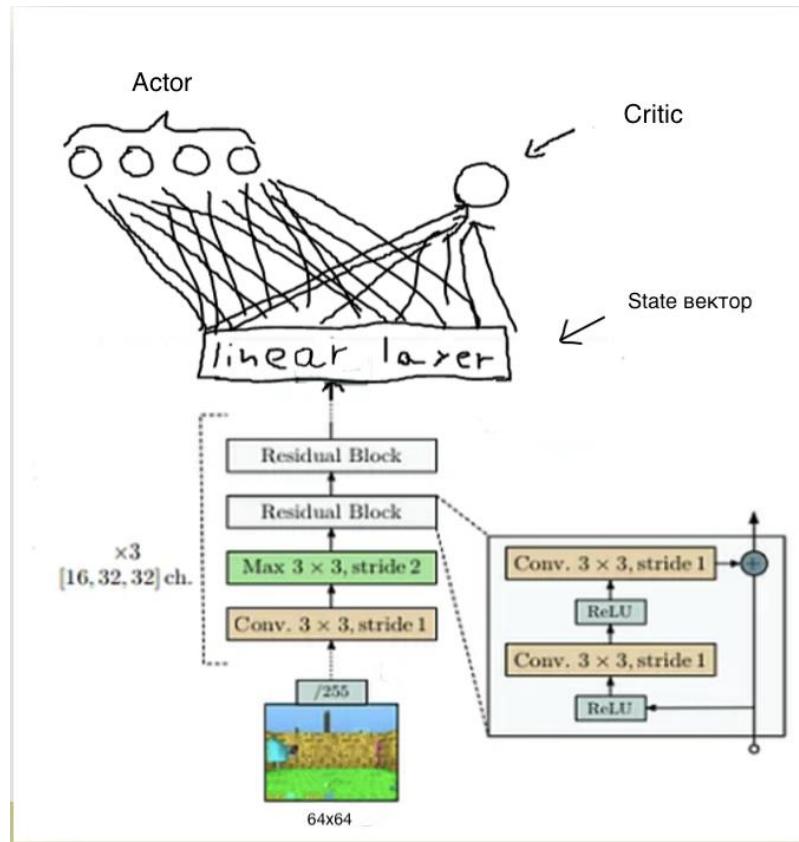
претставува претстава за состојбата. Во сликата подоле, каде што е претставена архитектурата, векторот на состојбата е тој „State вектор“.

Residual Convolutional Network

Еден од проблемите при користење на длабоки конволуциски мрежи е vanishing gradient проблемот. Овој проблем се јавува кога градиентите стануваат премногу мали, што предизвикува бавно учење или целосно запирање на ажурирањето на тежините во пораните слоеви на мрежата. За да се реши овој проблем, се користат Residual Convolutional Networks.

Наместо секој слој да ја менува целата информација што доаѓа од претходниот слој, оваа архитектура дозволува дел од информацијата директно да се пренесе на подлабоките слоеви, прескокнувајќи неколку конволуциски слоеви. Ваквите конекции се нарекуваат „skip connections“. Со овој пристап, градиентите подобро се пренесуваат низ мрежата, што го забрзува и стабилизира процесот на учење.

На сликата може да се види како сликата од играта (64×64 пиксели) влегува во конволуциската мрежа, каде што поминува најпрво поминува низ конволуциски слоеви со ReLU активација, па потоа им се прави Max pooling, за да се намали димензијата на сликата. Исто така информацијата тече и низ тие „skip connections“.



Actor-critic делот во конволуциската мрежа

Бидејќи го користам Actor-critic методот, оваа мрежа има два излези:

1. Actor

Излезот за Actor делот е Softmax слој, кој претставува веројатност за сите можни акции. Секој неврон одговара на една можна акција што агентот може да ја избере. Во текот на тренингот, агентот учи која акција треба да биде избрана за максимален успех во играта.

2. Critic

Излезот од Critic е еден неврон, кој ја предвидува value вредноста за дадена состојба.

Функции на загуба (Loss функции)

Бидејќи невронската мрежа има два дела – Actor и Critic, потребни ни се две loss функции, секоја за различен дел од мрежата.

Critic Loss (Mean Squared Error - MSE)

Loss функцијата за Critic делот е едноставна Mean Squared Error (MSE) помеѓу:

- Вредноста што ја предвидува мрежата за даден state.
- Return-от, кој претставува "истинска" (оценета со GAE методот) вредност на state-от.

Advantage

Advantage претставува разликата помеѓу:

- Истинската вредност на state-от (R) – што се добива од награди
- Предвидената вредност на state-от (V) – што ја предвидува Critic мрежата

$$A_t = R_t - V_t$$

Ако Advantage > 0 , тоа значи дека агентот е во подобра состојба отколку што мрежата предвидела, па Actor мрежата треба повеќе да ја користи таа акција. Но, во Reinforcement learning return-от никогаш не е совршено точен. Затоа користам GAE за да добијам подобра проценка.

Generalized Advantage Estimation (GAE)

GAE пресметува подобра проценка на Advantage-от со користење на понатамошни награди и со тоа ја намалува варијансата на проценките.

Најпрвин пресметуваме δ (delta), која ја претставува грешката во предвидувањето. Односно доколку не користевме GAE ова ќе ни беше Advantage-от. Return-от за моменталната состојба го калкулираме како reward-от за моменталната состојба плюс намалена предвидената вредност за идната состојба.

$$\delta_{t+1}^V = r_t + \gamma \mathbf{V}(s_{t+1}) - \mathbf{V}(s_t)$$

GAE дополнително ја подобрува проценката за Advantage-от со тоа што ја намалува варијансата преку сумирање на повеќе идни грешки (δ) со одреден коефициент (τ). На овој начин, наместо да се потпреме само на еден чекор во иднина, GAE ги вклучува и придонесите од сите идни чекори на агентот во траекторијата. Тоа му овозможува на агентот подобро да го процени влијанието на своите акции врз иднината.

$$\hat{\mathbf{A}}_t^{GAE(\gamma, \tau)} = \sum_{l=0}^{\infty} (\gamma \tau)^l \delta_{t+1}^V$$

Во код оваа GAE апроксимација е реализирана со тоа што одиме наназад низ траекторијата на агентот и за секоја состојба пресметуваме делта(δ). Потоа ја пресметуваме проценката со тоа што на делтата на моменталната состојба додаваме делти од идните состојби. На крај ги пресметуваме и returns-от кои ќе беа потребни во critic loss функцијата. Тие ги пресметувам како предвидената вредност за состојбата плюс GAE проценката.

```
def compute_gae(next_value, rewards, masks, values, gamma=0.999, tau=0.95):
    gae = 0
    returns = deque()
    gae_logger = deque()

    for step in reversed(range(len(rewards))):
        delta = rewards[step] + gamma * next_value * masks[step] - values[step]
        gae = delta + gamma * tau * masks[step] * gae
        next_value = values[step]

        returns.appendleft(gae + values[step])
        gae_logger.appendleft(gae)

    return returns, gae_logger
```

Proximal Policy Optimization (PPO)

PPO е алгоритам кој се користи за оптимизација на политиката на агентот. Политиката ја претставува веројатноста за избор на секоја можна акција во дадена состојба. PPO има за цел да ја подобри политиката на агентот со текот на времето, но на стабилен и контролиран начин.

PPO користи сооднос на политиките, кој покажува колку веројатноста на дадена акција, според новата политика се разликува од веројатноста на истата акција, според старата политика:

- $\pi(a,s)$ е веројатноста за избор на акција a , во состојба s според новата политика(со ажурирани тежини на мрежата).
- $\pi_{old}(a,s)$ е веројатноста за истата акција според старата политика(пред ажурирање на тежините на мрежата).

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

Без контрола, овој сооднос може да стане преголем или премал, што може да доведе до нестабилност во учењето и многу екстремна промена на тежините. За да се спречат преголеми промени во политиката, PPO воведува клипинг механизам. Формулата за клипирана loss функција е:

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

- \hat{A} – претставува GAE апроксимацијата, која проценува дали одредена акција е подобра или полоша од очекуваното.
- ϵ (јас го поставил на 0.2) е хиперпараметар кој ја контролира дозволената промена на политиката.

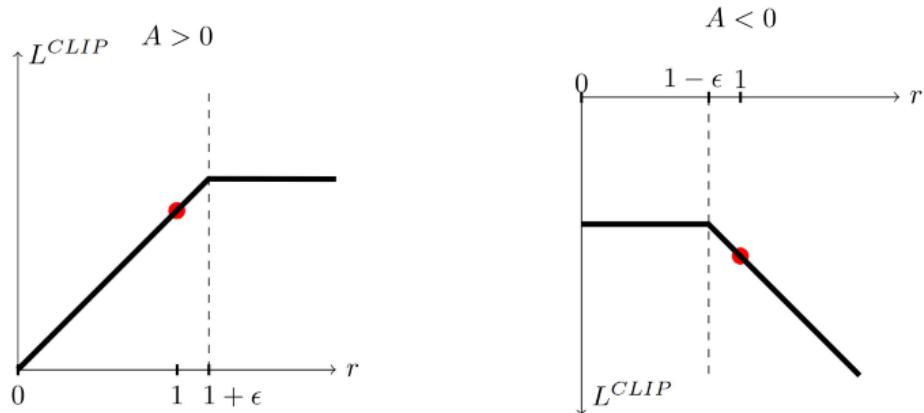
Функцијата $\min()$ гарантира дека политиката нема да се промени премногу брзо:

- Ако соодносот стане преголем (над $1+\epsilon$), тогаш тој се ограничува на $1+\epsilon$ (во мојот случај на 1.2).
- Ако соодносот стане премал (под $1-\epsilon$), исто така се ограничува прометата.

Оптимизирањето со помош на оваа функција се одвива така што ние сакаме да ја максимираме нејзината вредност, што значи дека сакаме да ја зголемиме веројатноста на добите акции и да ја намалиме веројатноста на лошите акции. Ова се случува на следниот начин:

- Кога $A > 0$, тоа значи дека акцијата што сме ја направиле била подобра од очекуваното. Поради тоа, сакаме да ја зголемиме веројатноста за таа акција, за да агентот ја користи почесто во иднина. Но, PPO го ограничува зголемувањето, за да спречи премногу големи промени во политиката.

- Кога $A < 0$, тоа значи дека акцијата што сме ја направиле била полоша од очекуваното. Затоа, сакаме да ја намалиме нејзината веројатност, за агентот во иднина да ја избегнува. И тука PPO го ограничува намалувањето на веројатноста, за да спречи премногу нагли промени во политиката.



Процесот на ажурирање на мрежата

```
def ppo_update(data_buffer, ppo_epochs, clip_param):
    for _ in range(ppo_epochs):
        for data_mini_batch in data_buffer:

            new_dist, new_value = rl_model(data_mini_batch["states"])

            actor_loss = ppo_loss(new_dist, data_mini_batch["actions"], data_mini_batch["log_probs"], data_mini_batch["advantages"],
                                  clip_param)

            critic_loss_v = critic_loss(new_value, data_mini_batch["returns"])

            whole_loss = critic_loss_v - actor_loss
            optimizer.zero_grad()
            whole_loss.backward()

            nn.utils.clip_grad_norm_(rl_model.parameters(), 40)
            optimizer.step()
```

Во оваа функција, ги изминувам сите мини-бачови и врз основа на нив се ажурираат тежините на невронската мрежа. Во следната точка детално ќе биде објаснето за мини-бачовите и истите како се конструирани. Користам 3 епохи за тренирање, што значи дека податоците ќе бидат обработени три пати пред да преминеме на нов batch. Ова помага да се искористат максимално податоците од едно играње пред тие да бидат заменети со нови искуства.

За секој мини-бач ги пресметуваме actor loss и critic loss функциите. Потоа, ги комбинираме двете загуби во една функција наречена `whole_loss`. Во таа функција:

- Actor loss се одзема, бидејќи PyTorch користи „gradient descent“, ние сакаме да ја максимираме оваа функција (да ги направиме подобрите акции повеќе веројатни).

- Critic loss се додава, бидејќи сакаме да ја минимизираме (да ја намалиме грешката при предвидување на вредноста на состојбите).

На крајот, се пресметуваат градиентите и се ажурираат тежините на мрежата.

Собирање и чување на податоци

Доколку ги ажурираме тежините на моделот по секој чекор што го прави агентот, тренирањето ќе биде многу нестабилно и би барало многу хардверски ресурси. Ова би довело до висока варијанса, што значи дека тежините на мрежата би можеле драстично да се менуваат при секое ажурирање, наместо постепено да конвергираат кон подобра политика.

За да се избегне ова, се користат бачеви од податоци, каде што агентот прво собира податоци преку повеќе чекори, а потоа ги користи овие податоци за ажурирање на мрежата.

Но, ажурирање на тежините само еднаш по цел бач исто така има свои недостатоци. Ако бачот е голем, ќе има многу малку ажурирања на мрежата, што ќе ја забави оптимизацијата. Дополнително, PPO алгоритмот не дозволува преголеми промени во политиката, што значи дека едно ажурирање по бач не би било доволно. Затоа, бачот го разделувам на мини-бачови и секој мини-бач се користи за ажурирање на тежините на мрежата.

Исто така чекорите во мини-бачовите се рандомизирани. Ако не беа рандомизирани, моделот ќе научеше силни зависности од последователните состојби, што значи дека нема да може добро да генерализира на нови, невидени сценарија. Исто така и доколку последователните податоци се корелирани може да ни overfit-не мрежата.

```
class ReplayBuffer:
    def __init__(self, data_names, buffer_size, mini_batch_size, device):
        self.data_keys = data_names
        self.data_dict = {}
        self.buffer_size = buffer_size
        self.mini_batch_size = mini_batch_size
        self.device = device

        for name in self.data_keys:
            self.data_dict[name] = deque(maxlen=self.buffer_size)

    def data_log(self, data_name, data):
        data = data.cpu().split(1)
        self.data_dict[data_name].extend(data)

    def __iter__(self):
        batch_size = len(self.data_dict[self.data_keys[0]])
        batch_size = batch_size - batch_size % self.mini_batch_size

        ids = np.random.permutation(batch_size)
        ids = np.split(ids, batch_size // self.mini_batch_size)
        for i in range(len(ids)):
            batch_dict = {}
            for name in self.data_keys:
                c = [self.data_dict[name][j] for j in ids[i]]
                batch_dict[name] = torch.cat(c).to(self.device)
            batch_dict["batch_size"] = len(ids[i])
            yield batch_dict
```

За да ги зачувам собраниите податоци, користам Replay Buffer структура. Големината на оваа структура е голема да собере точно еден бач од податоци. Оваа структура служи како меморија каде што ги чувам податоците за секој чекор што агентот го направил. Во Replay Buffer-от ги чувам:

1. states – состојби (слики во пиксели) во кои се наоѓал агентот при извршување на секоја акција.
2. actions – акциите што ги избрал агентот во дадените состојби.
3. returns – пресметаниот Return за секоја состојба(со GAE методот)
4. log_probs – логаритамска веројатност за акцијата која била одбрана во дадена состојба
5. values – проценетата вредност за секоја состојба, предвидена од Critic делот на мрежата.
6. advantages – пресметани од GAE методот

Во функцијата `pro_update`, кога го изминуваме баферот, всушност ја повикуваме функцијата `__iter__`. Таа функција најпрво ќи меша сите податоци собрани од еден беч. Ги дели тие податоци во подеднакво големи групи(мини-бечови). Така мини-бечовите ги испраќа еден по еден во циклусот за ажурирање на тежините на мрежата.

Помошни функции

```
def state_to_tensor(state_dict, device):  
    if type(state_dict) is dict:  
        state = torch.FloatTensor(state_dict['rgb'].transpose((0, 3, 1, 2))).to(device)  
    else:  
        state = torch.FloatTensor(state_dict.transpose((2, 0, 1))).unsqueeze(0).to(device)  
  
    return state / 256  
  
def tensor_to_unit8(tensor):  
    numpy_float = tensor.squeeze(0).cpu().numpy().transpose((1, 2, 0))  
    return (numpy_float * 255).astype(np.uint8)
```

`State_to_tensor` функцијата ја претвора состојбата добиена од Procgen во PyTorch tensor за да може да биде процесирана од невронската мрежа. Состојбата што ја враќа Procgen е слика претставена како низа од пиксели. Исто така, вредностите на пикселите од 0-255 ги нормализирам од 0-1, бидејќи невронските мрежи подобро функционираат со помали вредности.

`Tensor_to_unit` функцијата ја врши спротивната трансформација. Таа ја претвора сликата од PyTorch tensor во класична слика со пиксел вредности од 0 до 255. Ова ни е потребно за визуелизација на тестирањето и креирање на GIF анимации за да можеме да го видиме однесувањето на агентот.

```

def run_tests(dist_mode, env_name, num_levels, train_test="train"):
    if train_test == "train":
        start_level = 0
    else:
        start_level = num_levels

    scores = []
    for i in range(num_levels):
        env = gym.make("procgen:procgen-" + env_name + "-v0",
                       start_level=start_level + i, num_levels=1, distribution_mode=dist_mode)
        scores.append(test_agent(env))

    return np.mean(scores)

```

Run_tests функцијата се користи за тестирање на агентот. Доколку параметарот train_test е поставен на „train“, агентот се тестира на првите 20 нивоа(тие се нивоата на кои го тренирам агентот). Доколку е „test“, тогаш агентот се тестира на следните 20 нивоа. Функцијата ја повикува test_agent функцијата за секое ниво, а потоа ги собира сите добиени резултати и враќа просечен reward, што претставува оценка за тоа колку е добар агентот на сите тест-нивоа.

```

def test_agent(env, log_states=False):
    start_state = env.reset()
    state = state_to_tensor(start_state, device)

    if log_states:
        states_logger = [tensor_to_unit8(state)]

    done = False
    total_reward = 0
    with torch.no_grad():
        while not done:
            dist, _ = rl_model(state)
            action = dist.sample().item()

            next_state, reward, done, _ = env.step(action)
            total_reward += reward
            state = state_to_tensor(next_state, device)
            if log_states:
                states_logger.append(tensor_to_unit8(state))

    if log_states:
        return total_reward, states_logger
    else:
        return total_reward

```

Test_agent функцијата игра едно целосно ниво и ги мери перформансот на агентот. Доколку log_states е поставено на True, тогаш ги зачува сите состојби во кој бил агентот, и токму ова го користам за да го направам GIF-от.

Хиперпараметри

num_steps = 256

Оваа променлива дефинира колку чекори треба да направи секој агент пред да се ажурира мрежата. Бидејќи користиме num_envs = 64, тоа значи дека во еден batch, паралелно работат 64 агенти и секој агент игра по 256 чекори. Доколку некој агент заврши едно ниво порано (на пример, за 20 чекори), веднаш му започнува ново ниво сè додека не ги одигра сите 256 чекори.

max_frames = 1000000

Овој параметар дефинира колку вкупно чекори (states) ќе има во целото тренирање.

rpo_epochs = 3

Означува колку пати ќе поминеме низ сите мини-бачови за време на ажурирањето на мрежата. Поголема вредност значи подетално учење, но истовремено и поголемо процесирање.

gamma, tau, clip_param

Овие параметри се користат во loss функциите. Gamma и tau се параметри за GAE методот. Gamma = 0.999 е факторот на дисконт кој одредува колку агентот ги вреднува идните награди, при што вредност блиска до 1 значи дека идните награди имаат големо влијание. Tau = 0.95 контролира баланс помеѓу варијансата и пристрасноста во проценката на adventage-to . clip_param = 0.2 е границата за промена на политиката во PPO, која спречува преоголеми ажурирања на веројатностите на акциите, осигурувајќи стабилност во учењето.

buffer_size

Го дефинира големината на Replay Buffer-от, кој мора да собере еден цел batch од податоци. Бидејќи num_steps = 256 и num_envs = 64, вкупниот број на состојби зачувани во еден batch е $256 \times 64 = 16384$.

Главен циклус

Во главниот циклус, се одвива целокупната интеракција на агентот со околината и ажурирањето на невронската мрежа со користење на собраните податоци. Надворешниот while циклус се извршува сè додека вкупниот број на интеракции (frames_seen) не ја достигне поставената граница max_frames. Ова ни осигурува дека агентот нема да тренира бесконечно, туку во однапред дефиниран број на чекори. Потоа, иницијализираме deque структури во кои ќе ги чуваме податоците.

Во внатрешниот циклус (step < num_steps), агентот носи одлуки за акциите што ќе ги изврши врз основа на моменталната состојба. Откога ќе донесе одлука за акцијата, таа акција се применува во procgen околината и се добива нова состојба. Бидејќи добивките во Procgen се релативно големи (за собирање паричката се добива добивка 10), па за да се избегне нестабилност во тренирањето, ги скалирам добивките, така што сите позитивни вредности стануваат 1. Секој чекор кој што ќе го направат паралелните агенти го зачувуваме во структурите, кои после завршување на бечот ќе се логираат во баферот.

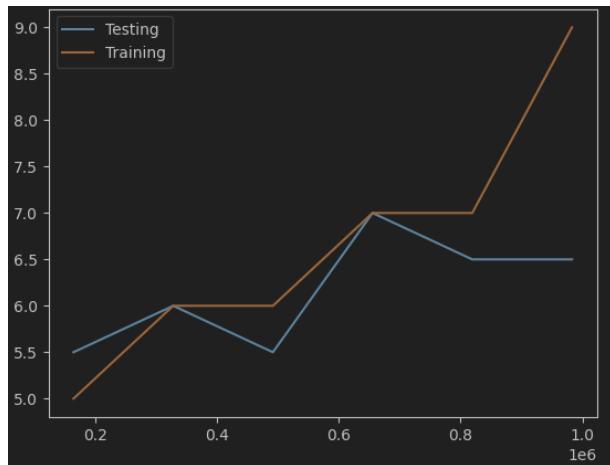
Откако ќе се соберат сите податоци за еден беч, пресметуваме returns и adventages. Па откога ќе заврши бечот се логираат податоците во баферот и се повикува функцијата rpo_update, која ќе изврши ажурирање на тежините на мрежата.

Нормализација на adventage-от

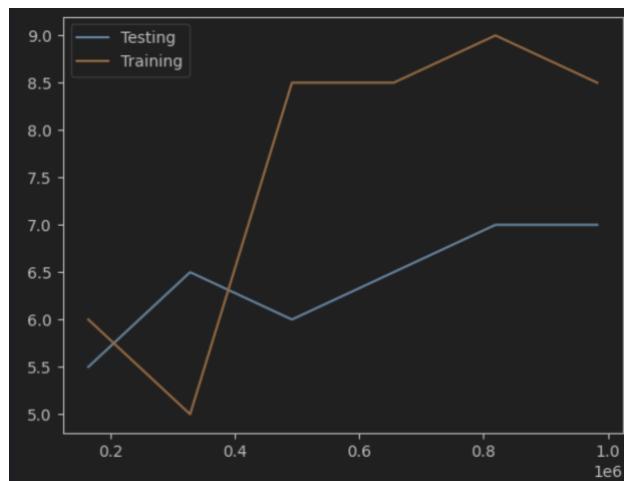
Додадов нормализација на adventage-от , што значително ја подобрува стабилноста на тренирањето и му помага на моделот да научи побрзо. Без нормализација, вредностите на adventage-от можат да бидат многу различни по големина, што го отежнува тренирањето. Исто така, ако пресметаниот Adventage е многу мал, тоа значи дека агентот нема јасен сигнал за тоа кои акции се добри, а кои не, што може да го забави учењето. Па со нормализација тој сигнал се прави подинамичен и подобар за тренирање. Пример доколку вредностите за adventage-от се [0.1, 0.12, 0.08, 0.09, 0.11], агентот нема многу да ги разликува, но после нормализацијата ќе се добијат следниве вредности [1.2, 1.5, -0.8, -0.6, 0.9], кои се многу попогодни за тренирање на мрежата.

Евалуација на моделот

Без користење на нормализација, ова се резултатите од тренирањето:

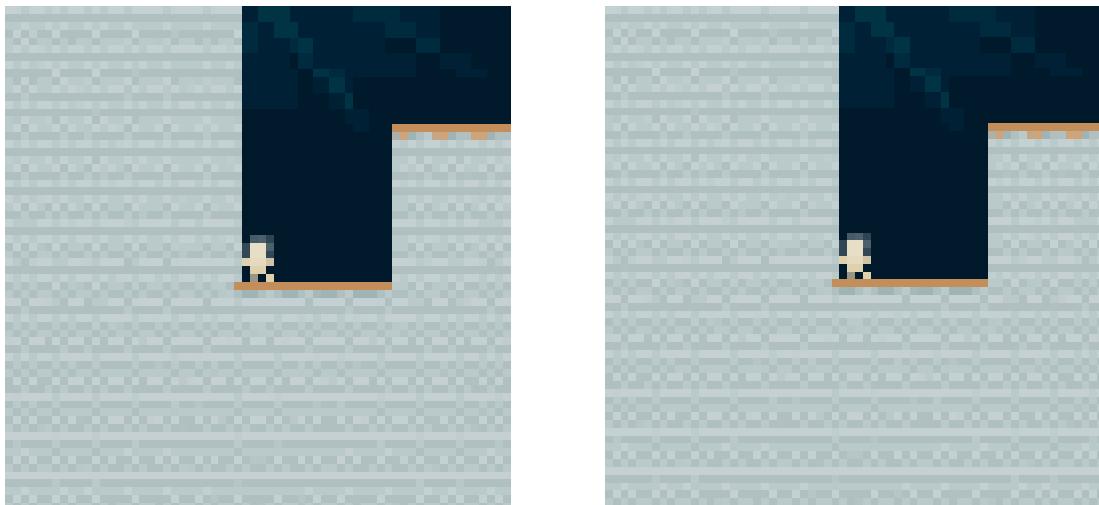


Агентот ќе добива reward 10, доколку ја земе паричката. Па на графикот за тестирачките нивоа можеме да видиме, дека приближно во 65% од нивоата успева да ја достигне паричката, а во останатите не ја достигнува. Со нормализација се добива следниот график за евалуација:



Може лесно да се воочи, дека на агентот многу помалце интерации му требаат за да научи да игра на тренирачкото множество и на крај од тренирањето успешно совладува 70% од нивоата за тестирање.

Приказ на играње на играта на агентот без нормализација(лево) и на финалниот агент со нормализација(десно):



Референци

<https://www.davidsilver.uk/teaching/>

<https://spinningup.openai.com/en/latest/>

<https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html>

<https://github.com/openai/procgen>

<https://github.com/voiler/IMPALA/blob/master/model.py>

<https://www.youtube.com/watch?v=hlv79rcHws0>