Multi-Arm Bandits(MAB) 코드 분석

1. 개요

1.1. 선정 목적

- Multi-Arm Bandits(이하 MAB) 알고리즘의 빠른 이해 가능
- 변수명, 구현 방법이 관련 논문들의 정의 및 psudeo code와 일치
- Module화가 잘 되어 있어 MAB Framework 구축 용이

1.2. 정의

- A MAB is a tuple $< A, R >$, where A is a set of k possible actions
- $P(R|A)$: an unknown probability distribution of rewards given the action chosen
- At each time step t the agent selects an action $A_t \in A$
- At each time step t the environment generates a reward $R_t \sim P(\cdot, A)$
- $N_t(a)$: t 시점까지 특정 arm(a)이 선택된 횟수; $N_t(a) = \sum_{t=1}^{T} \mathbf{1}_{A_t=a}$ ($\mathbf{1}$: indicator function)
- Action–value: Expected(mean) reward for action a: $Q_t(a) = E[R_t|A_t]$
  - 즉 t 시점까지의 reward 합계 / arm이 선택된 총 횟수; $\frac{1}{N_t(a)} \sum_{t=1}^{T} R_t \cdot \mathbf{1}_{A_t=a}$
- Optimal value: $V_t^* = Q_t(a^*) = \max_a Q_t(a)$
- Regret: Opportunity loss for one step: $l_t = E[V_t^* - Q_t(a)]$
- Goal: Maximize cumulative reward: $\sum_{t=1}^{T} R_t$ = minimize total regret $E[\sum_{t=1}^{T}(V_t^* - Q_t(a))]$

1.3. 코드 구성

- Main.py: 메인 구동 및 plotting
- Bandit.py: Bandit 확률 분포 정의 (가우시안, 베르누이 등)
- Policy.py: MAB 알고리즘의 core부 (greedy, MAB 등)
- Agent.py: 실제 arm을 당겼을 때의 event 처리
- Environment.py: Bandit과 Agent의 인스턴스를 취해 실제 MAB 구동

2. Main Module: Main.py

- MAB 알고리즘 구동에 필요한 매개변수 및 하이퍼파라메터 설정
- Bandit 인스턴스 생성 (GaussianBandit, BinomialBandit, BernoulliBandit 중 선택)
- Policy 인스턴스 생성 (EpsilonGreedyPolicy, GreedyPolicy, RandomPolicy, UCBPolicy, SoftmaxPolicy 중 선택)
- Agent 인스턴스 생성 (Agent, GradientAgent, BetaAgent 중 선택) (복수의 agent들을 배열로 묶어서 벤치마크 가능한 구조)
- Environment 인스턴스 생성 및 MAB 구동

```
n_arms = 10 # arm의 갯수, 즉 k
n_trials = 1000 # Number of iterations (time)
n_experiments = 500 # Converge 척도 실험; 각자 다른 세팅의(default:random) 500
개의 k arm들을 구동시켜 평균을 취하기 위한 목적임
bandit = bd.GaussianBandit(n_arms)

# Greedy, 하이퍼파라메터(epsilon)별 EpsilonGreedy 알고리즘 테스트 가능
agents = [
    bd.Agent(bandit, bd.GreedyPolicy()),
    bd.Agent(bandit, bd.EpsilonGreedyPolicy(0.01)),
    bd.Agent(bandit, bd.EpsilonGreedyPolicy(0.1)),
]

env = bd.Environment(bandit, agents, 'Epsilon-Greedy')
scores, optimal = env.run(n_trials, n_experiments)
# 결과 plot
env.plot_results(scores, optimal)
```

3. Sub Module: Bandit.py

- Bandit의 확률 분포 정의
- 알고리즘 벤치마킹용으로는 GaussianBandit, BinomialBandit이 주로 사용됨

3.1. 클래스 개요

- **MultiArmedBandit**: 수퍼클래스로 하기 3개 클래스들의 부모 클래스
- **GaussianBandit**: Stationary 환경의 MAB 알고리즘 벤치마킹용
- **BinomialBandit**: 추천 평점 (discrete user rating) 등에 활용 가능
- **BernoulliBandit**: 제목/링크 클릭 여부, 웹페이지 방문 여부, 추천 유/무 등에 활용 가능

3.2. 클래스 설명

- **MultiArmedBandit**: Bandit 수퍼클래스
  - *k*: arm의 총 개수
  - *action*value_: arm을 당겼을 때 얻을 수 있는 reward의 기대값;
    (expected reward given some action $q_*(a) = E[R_t|A_t = a]$)
  - *optimal*: k개의 arm중 가장 reward가 높은 arm의 index로 ground truth
  - *reset()*: action_value (for i = 1 to k)와 optimal을 모두 0으로 초기화
  - [reward, is_optimal] = *pull(action)*: i번째 arm을 당겼을 때의 reward와 optimal과의 일치 여부 리턴
    (예: 가장 reward가 높은 arm의 index가 4이고 알고리즘으로 예측한 최적 arm의 index가 4이면 True)
- **GaussianBandit**: action_value(즉, ground truth)의 범위가 가우시안(정규) 분포에 의해 결정됨
  (예: 1번째 arm의 action_value: 0.583, 2번째 arm의 action_value: -0.422)
  - *mu*: 평균 (default=0)
  - *sigma*: 표준편차 (default=1)
  - *reset()*: action_value 값의 범위를 가우시안 분포로 설정하고 optimal값을 action_value의 최대 index로 설정
    `action_values = np.random.normal(mu, sigma, k)`
  - *pull(action)*: i번째 arm을 당겼을 때의 reward의 값 범위가 가우시안 분포 내에서 결정됨
    `return (np.random.normal(action_values[action]),`

```
action == optimal)</code>
```

- **BinomialBandit**: action_value(즉, ground truth)의 범위가 이항 분포에 의해 결정됨
  (Note: 시행 횟수 n이 크면 이항 분포는 가우시안 분포를 따름)
  - *n*: n번 베르누이 독립 시행 횟수 (즉, 베르누이 분포는 n=1)
  - *p*: 각 시행에서의 성공 확률 (0<p<1)
  - *t*: number of trials (iterations)
  - *model*: pymc의 Model 함수 호출
  - *bin*: 이항 분포
  - *pull(action)*: i번째 arm을 당길 때마다 _cursor 값이 1 증가
- **BernoulliBandit**: action_value(즉, ground truth)의 범위가 베르누이 분포에 의해 결정됨
  (예: 1번째 arm의 action_value: 0.6의 확률로 1, 2번째 arm의 action_value: 0.4의 확률로 0)
  BinomialBandit 클래스를 상속받는 클래스로 생성부에서 n=1, p=None, t=None로 설정

4. Sub Module: Policy.py

- 탐색(Exploration)과 획득(Exploitation)의 trade-off를 조절, 즉 MAB 알고리즘의 core부
  - 탐색만 취하면 random성에 의존하기 때문에 불리함 (다만, 확률 모델에 기반하여 사후 분포를 모델링할 수 있다면 탐색 위주로 하여도 좋은 결과를 얻을 수 있음, 예: Thompson Sampling)
  - 획득만 취하면, 즉 greedy한 방법을 적용하면, 단기간에는 이득이지만 중장기적으로 볼 때 불리함
- 직관적으로 성능을 좋게 하려면?
  - 초기에는 탐색을 많이 하여 최적의 arm을 선택할 수 있게 유도
  - 초기 prior reward를 매우 높게 설정
    (prior reward가 크면 regret가 커지므로, 그에 따라 탐색을 더 빈번하게 시도하므로)
  - Annealing 기법을 적용

4.1. 클래스 개요

- **Policy**: 수퍼클래스로 하기 5개 클래스들의 부모 클래스
- **EpsilonGreedyPolicy**: ϵ 확률로 random하게 탐색하고 $1-\epsilon$의 확률로 greedy하게 가장 좋은 arm을 선택
- **GreedyPolicy**: EpsilonGreedyPolicy의 파생 클래스로 $\epsilon = 0$인 경우
- **RandomPolicy**: EpsilonGreedyPolicy의 파생 클래스로 $\epsilon = 1$인 경우
- **UCBPolicy**: Upper Confidence Bound(이하 UCB)를 설정하여 reward에 대한 불확실성을 고려
  - As-is: 주사위를 여러 번 던졌을 때 기대값은 3.5이지만, 두 번만 던졌을 때 눈금이 1, 3이 나오면 기대값이 2가 나오므로 실제 기대값과 편차가 심함
  - To-be: 주사위를 두 번만 던졌을 때 [2, 5.2]의 범위로 Confidence Interval을 정하고 횟수가 증가할 수록 Confidence Interval을 감소시켜 불확실성을 줄임
- **SoftmaxPolicy**: 각 arm의 성공 확률을 비율로 고려하여 성공 확률이 높을수록 arm을 선택하는 비율이 높게 유도

4.2. 클래스 설명

- **Policy**
  - *[action] = choose(agent)*: 최적의 arm을 선택하는 멤버 함수로 Agent 클래스에서 호출
  - Policy 클래스의 모든 파생 클래스들은 choose 멤버 함수만 정의되어 있음
- **EpsilonGreedyPolicy**
  - $A_t \leftarrow \begin{cases} \text{argmax}_a Q_t(a) & \text{with probability } 1-\epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$

  - $Q_t(a)$ = sum of rewards when a taken prior to t / number of times a prior to t
    즉 $t-1$ 시점까지의 reward 합계 / arm이 선택된 총 횟수, 코드의 변수명은 value_estimates
  - 평균으로 계산 가능하지만 각 루프마다 일일이 계산하면 bottleneck이 되므로, 실제 구현에는 이전 평균값을 이용해서 간단히 계산할 수 있는 수식으로 변환하여 사용
- **GreedyPolicy**: $\epsilon = 0$
- **RandomPolicy**: $\epsilon = 1$
- **UCBPolicy**: Upper Confidence Bound(이하 UCB)를 설정하여 reward에 대한 불확실성을 고려
  - $A_t = \text{argmax}_a[Q_t(a) + U_t(a)] = \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$
    $N_t(a)$: t 시점까지 특정 arm(a)이 선택된 횟수, 코드 내 변수는 action_attempts
    c: Confidence level를 조절하는 하이퍼파라메터로 값이 높을수록 탐색 빈도 증가, 논문에서는 $\sqrt{2}$로 설정
  - 직관적 이해1: 계속 특정 arm만 선택하면 UCB term이 작아지므로 Confidence Interval이 감소하여 탐색 빈도 감소 ($N_t(a)$ 가 증가하므로)
  - 직관적 이해2: log의 역할은 UCB decay로 점차 Confidence Interval을 감소시키는 역할
  - 직관적 이해3: 잘 선택되지 않은 arm은 Confidence Interval이 감소하여 탐색 빈도 증가
- **SoftmaxPolicy**: 각 arm의 성공 확률을 비율로 고려하여 성공 확률이 높을수록 arm을 선택하는 비율이 높게 유도
  - random으로 생성된 값을 $p = \frac{\exp(Q_t(a))}{\sum_{b=1}^{k} \exp(Q_t(b))}$ 와 비교

5. Sub Module: Agent.py

- Bandit 인스턴스와 Policy 인스턴스를 취함
- 각 time step 별로 arm을 주어진 policy(예: greedy, UCB)에 따라 선택하는 역할
- 여러 agent들을 배열로 묶어서 알고리즘 벤치마킹용으로 처리 가능

5.1. 클래스 개요

- **Agent**: 수퍼클래스로 하기 2개 클래스들의 부모 클래스
- **GradientAgent**: 각 action별로 preference를 학습 (R_t와 $\bar{R}_t$ 의 차이, $\bar{R}_t$: t 시점까지의 모든 arm에 대한 reward의 평균)
  GradientAgent의 성능이 Agent보다 더 좋지만 preference를 계산하기 위한 추가 로직 필요
- **BetaAgent**: Beta 분포를 prior로 가정하고 Bandit의 분포를 베르누이 분포나 이항 분포의 형태를 가지는 likelihood로 가정하여 확률 분포를 모델링
  Thompson Sampling에서 활용

5.2. 클래스 설명

- **Agent**
  - *policy*: Policy 인스턴스
  - *k*: arm 개수 (Bandit 인스턴스에서 받아옴)
  - *prior*: Initial action value (default는 0이지만 값을 높게 부여하여 탐색 비중을 높일 수 있게 설정 가능)
  - *gamma*: update rule이나 평균 계산 시에 쓰이는 stepsize로 gamma=None일 경우 $\gamma = N_t(a)$
  - *value*estimates_: 각 arm의 Action value의 추정치로 초기값은 `prior*np.ones(self.k)`
  - *action*attempts_: 각 arm의 당김 횟수로 초기값은 `np.zeros(self.k)`
  - *t*: time step
  - *last*action_: $t-1$ 시점에서 선택한 arm의 index
  - *[action] = choose()*: Policy 클래스의 choose 멤버 함수를 호출하여 last_action 멤버 변수로 저장
  - *observe(reward)*: Environment 인스턴스에서 호출하는 멤버 함수

    - action_attempts[last_action] += 1
    - gamma값에 따른 step size 설정
    - value_estimate 업데이트;
      $NewEstimate \leftarrow OldEstimate + StepSize[TargetReward - OldEstimate]$
      `self.valueestimates[self.last_action] += g*(reward - q)`
    - t += 1
- **GradientAgent**
  - *alpha* = Stochastic gradient ascent의 step-size 하이퍼파라메터
  - *baseline*: Default=True (True일 경우는 average_reward을 baseline으로 하여 reward와 average_reward의 차이를 계산)
  - *average*reward_: 평균 reward
  - *observe(reward)*: Stochastic gradient ascent에 따라 preference 업데이트
    $H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbf{1}_{a=A_t} - \pi_t(a))$
    상기 식을 더 직관적으로 풀어 쓰면, 선택된 하나의 arm에는
    $H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t))$ 이고 그 외의 모든 arm들에는
    $H_{t+1}(a) = H_t(a) - \alpha(R_t - \bar{R}_t)(\pi_t(a))$
    ※ 수식에서의 +, -기호 중요!
    $\pi_t(a)$ 는 t시점에서의 value_estimates에 대한 softmax function, 즉 action a를 선택할 확률
    직관적 이해: 특정 arm의 reward가 average_reward보다 크다면 preference가 증가하여
    향후 그 arm을 선택할 확률이 증가하고 반대로 average_reward보다 작다면 preference가 감소하여
    향후 그 arm을 선택할 확률이 감소함.

```
self.action_attempts[self.last_action] += 1

if self.baseline:
    diff = reward - self.average_reward
    self.average_reward += 1/np.sum(self.action_attempts) * diff

pi = np.exp(self.value_estimates) / np.sum(np.exp(self.value_estimate
s))

ht = self.value_estimates[self.last_action]
ht += self.alpha*(reward - self.average_reward)*(1-pi[self.last_actio
n])
self._value_estimates -= self.alpha*(reward - self.average_reward)*pi
self._value_estimates[self.last_action] = ht
self.t += 1
```

- **BetaAgent**
  - *n*: Bandit 인스턴스의 n ($n = 1$ 이면 베르누이 분포, 그보다 크면 이항 분포)
  - *ts*: Thompson Sampling 여부
    - True일 때는 value_estimates를 beta 분포에서 random하게 선택, False일 때는 alpha/(alpha+beta)로 선택
  - *prior*: prior 분포 (beta 분포)
  - *value*estimates_: 각 arm의 Action value의 추정치로 초기값은 `np.ones(self.k)`
  - *alpha*: beta 분포의 파라메터로 arm을 play해서 획득한 reward (성공 정도)
  - *beta*: beta 분포의 파라메터로 arm을 play해서 잃은 reward (실패 정도)
  - *observe(reward)*: alpha, beta 파라메터 정의 및 value_estimates 계산

6. Sub Module: Environment.py

- *bandit*: Bandit 인스턴스
- *agents*: Agent 인스턴스
- *label*: label명 (plotting시 제목)
- *[scores, optimal] = run(trials, experiments)*: Trials(number of time steps)와 experiments로 MAB 구동 (알고리즘 벤치마킹 시에는 experiments=100~2000으로 설정, 실제 적용 시에는 experiments=1로 설정)
  - scores: T시점까지의 reward의 총 합계 (average reward 산출용)
  - optimal: 알고리즘으로 reward가 가장 높을 거라 예측한 arm과 실제로 reward가 가장 높은 arm이 일치한다면 counter를 증가 (% of optimal action 산출용)

```
scores = np.zeros((trials, len(self.agents)))
optimal = np.zeros_like(scores)

for _ in range(experiments):
    self.reset()
    for t in range(trials):
        for i, agent in enumerate(self.agents):
            action = agent.choose()
            reward, is_optimal = self.bandit.pull(action)
            agent.observe(reward)

            scores[t, i] += reward
            if is_optimal:
                optimal[t, i] += 1

return scores / experiments, optimal / experiments
```

References

- Code: Python library for Multi-Armed Bandits (https://github.com/bgalbraith/bandits)
  - https://github.com/bgalbraith/bandits (https://github.com/bgalbraith/bandits)
- Theory: Reinforcement Learning: An Introduction, 2nd edition, 2016 Sep. Draft Version (http://ufal.mff.cuni.cz/~straka/courses/npfl114/2016/sutton-bookdraft2016sep.pdf)
  - http://ufal.mff.cuni.cz/~straka/courses/npfl114/2016/sutton-bookdraft2016sep.pdf (http://ufal.mff.cuni.cz/~straka/courses/npfl114/2016/sutton-bookdraft2016sep.pdf)
- Understanding: SanghyukChun's Blog: Machine Learning 스터디 (20) Reinforcement L earning (http://sanghyukchun.github.io/76/)
  - http://sanghyukchun.github.io/76/ (http://sanghyukchun.github.io/76/)