Online and Reinforcement Learning (2025)
Home Assignment 4

Davide Marchi 777881

Contents

1 Policy Gradient Methods

1.1 Baseline

We are given that the policy gradient theorem can be generalized to include an arbitrary baseline $b(s)$:

$$\nabla_\theta J(\pi) = \sum_{s \in S} \mu_\pi(s) \sum_{a \in A} \nabla_\theta \pi(s,a) \left(Q_\pi(s,a) - b(s) \right),$$

where:

- S is the state space.
- A is the action space.
- $\pi(s,a)$ is the probability of choosing action a in state s.
- $\mu_\pi(s)$ is the stationary state distribution under policy π.
- $Q_\pi(s,a)$ is the state-action value function.

The term

$$\sum_{a \in A} \nabla_\theta \pi(s,a) b(s)$$

acts as a control variate, and we must show that its expectation is zero, i.e.,

$$\mathbb{E}\left[\sum_{a \in A} \nabla_\theta \pi(s,a) b(s) \right] = 0.$$

Proof

For any state $s \in S$, note that $\pi(s,\cdot)$ is a probability distribution over A. Therefore, by definition:

$$\sum_{a \in A} \pi(s,a) = 1.$$

Differentiating both sides of the equation with respect to θ, we obtain:

$$\sum_{a \in A} \nabla_\theta \pi(s,a) = \nabla_\theta \left(\sum_{a \in A} \pi(s,a) \right) = \nabla_\theta(1) = 0.$$

Since $b(s)$ does not depend on the action a, it can be factored out of the summation:

$$\sum_{a \in A} \nabla_\theta \pi(s,a)\, b(s) = b(s) \sum_{a \in A} \nabla_\theta \pi(s,a) = b(s) \cdot 0 = 0.$$

Taking the expectation with respect to the stationary distribution $\mu_\pi(s)$, we have:

$$\mathbb{E}_{s\sim\mu_\pi}\left[\sum_{a\in A}\nabla_\theta\pi(s,a)b(s)\right] = \sum_{s\in S}\mu_\pi(s)\cdot 0 = 0.$$

Thus, we conclude that

$$\mathbb{E}\left[\sum_{a\in A}\nabla_\theta\pi(s,a)b(s)\right] = 0.$$

1.2 Lunar

1. Derivation of the Analytical Expression for the Score Function

I consider a softmax policy defined by

$$\pi(s,a) = \frac{\exp\left(\theta_a^\top s\right)}{\sum_{b\in A}\exp\left(\theta_b^\top s\right)},$$

where θ_a is the parameter vector corresponding to action a and $s \in \mathbb{R}^d$ is the state feature vector.

Taking the logarithm of the policy, I have:

$$\log\pi(s,a) = \theta_a^\top s - \log\left(\sum_{b\in A}\exp\left(\theta_b^\top s\right)\right).$$

I now differentiate this expression with respect to the parameters θ_i, for any action i. There are two cases:

Case 1: $i = a$ Differentiate $\log\pi(s,a)$ with respect to θ_a:

$$\nabla_{\theta_a}\log\pi(s,a) = \nabla_{\theta_a}\left[\theta_a^\top s\right] - \nabla_{\theta_a}\log\left(\sum_{b\in A}\exp\left(\theta_b^\top s\right)\right).$$

The first term is simply:

$$\nabla_{\theta_a}(\theta_a^\top s) = s.$$

For the second term, using the chain rule,

$$\nabla_{\theta_a}\log\left(\sum_{b\in A}\exp\left(\theta_b^\top s\right)\right) = \frac{1}{\sum_b\exp(\theta_b^\top s)}\cdot\nabla_{\theta_a}\left(\sum_b\exp(\theta_b^\top s)\right).$$

Since only the term with $b = a$ depends on θ_a, it follows that

$$\nabla_{\theta_a}\left(\sum_b\exp(\theta_b^\top s)\right) = \exp(\theta_a^\top s)\, s.$$

Thus,

$$\nabla_{\theta_a} \log \left(\sum_b \exp(\theta_b^\top s) \right) = \frac{\exp(\theta_a^\top s)}{\sum_b \exp(\theta_b^\top s)} s = \pi(s,a)\, s.$$

Therefore, for $i = a$,

$$\nabla_{\theta_a} \log \pi(s,a) = s - \pi(s,a)\, s = \big(1 - \pi(s,a)\big) s.$$

Case 2: $i \neq a$ For $i \neq a$, the first term is zero (since θ_i does not appear in $\theta_a^\top s$), and only the normalization term contributes:

$$\nabla_{\theta_i} \log \pi(s,a) = -\nabla_{\theta_i} \log \left(\sum_b \exp(\theta_b^\top s) \right).$$

Again, only the term with $b = i$ depends on θ_i, so

$$\nabla_{\theta_i} \left(\sum_b \exp(\theta_b^\top s) \right) = \exp(\theta_i^\top s)\, s,$$

and hence,

$$\nabla_{\theta_i} \log \pi(s,a) = -\frac{\exp(\theta_i^\top s)}{\sum_b \exp(\theta_b^\top s)} s = -\pi(s,i)\, s.$$

Combined Expression Thus, for every action i, the gradient is given by

$$\nabla_{\theta_i} \log \pi(s,a) = \begin{cases} \big(1 - \pi(s,a)\big) s, & \text{if } i = a, \\ -\pi(s,i)\, s, & \text{if } i \neq a. \end{cases}$$

In vector form (where the policy parameters are arranged in rows corresponding to actions), this can be compactly written as:

$$\nabla_\theta \log \pi(s,a) = \big(e_a - \pi(s)\big) s^\top,$$

with e_a denoting the one-hot vector for the action a.

2. Implementation of the Gradient Function

In my implementation, I only added the parts required to compute the analytical gradient for the softmax policy. The modified function `gradient_log_pi` in my `Softmax_policy` class is as follows:

```python
def gradient_log_pi(self, s, a):
    # Compute the probability vector for state s
    prob = self.pi(s)
    # Compute the gradient for each action (outer product of prob and s)
```

```
    grad = - np.outer(prob, s)
    # For the taken action a, add s to obtain (1 - pi(s,a))*s
    grad[a] += s
    return grad
```

Listing 1: Modified gradient_log_pi function

This code implements exactly the formula derived above.

3. Verification of the Gradient Implementation

To verify my implementation, I used the numerical approximation of the gradient in the function `gradient_log_pi_test`. I run the notebook cell to compare my analytical gradient with the numerical gradient for a range of random perturbations on the policy parameters. In all cases the analytical and numerical gradients agreed within the requested tolerance. This confirms that the derivation and implementation of `gradient_log_pi` are correct.

Also, here we present the graph showing the accumulated reward increasing over the number of episodes, which indicates that the policy is actually improving over time.

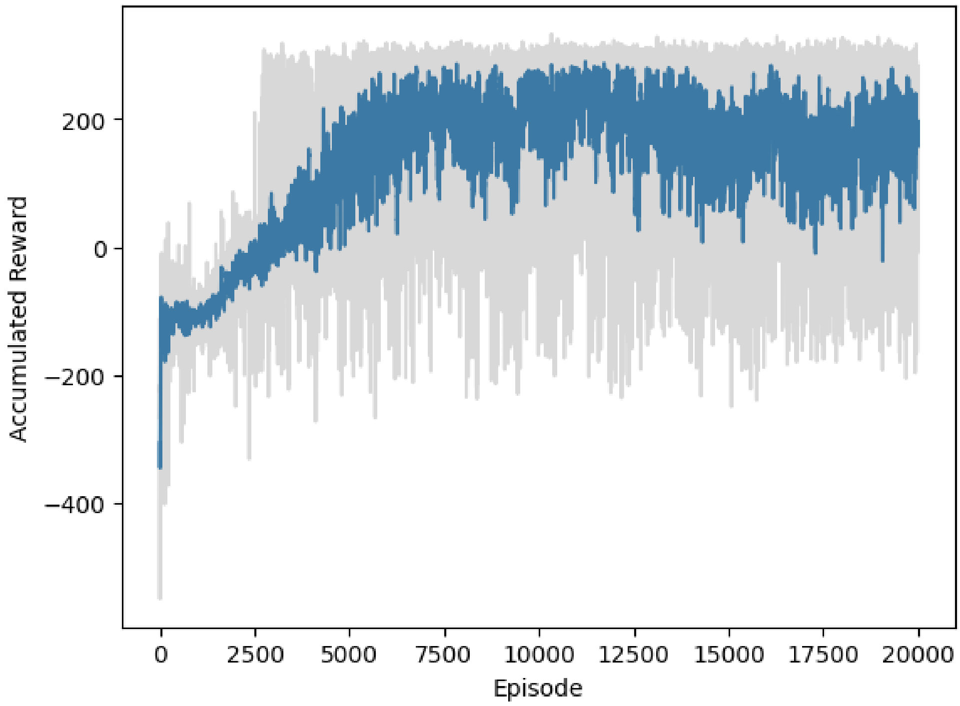

Figure 1: Accumulated reward over episodes.

2 Improved Parametrization of UCB1

(Optional)

3 Introduction of New Products

I focus on a scenario with:

- *Old product:* known success probability 0.5.
- *New product:* unknown success probability μ.

At each time step $t = 1, \ldots, T$, I must pick exactly one product to offer, aiming to maximize the total number of successful sales. Let $\Delta = 0.5 - \mu$:

$$\Delta > 0 \iff \mu < 0.5 \quad \text{(new product is worse)},$$
$$\Delta < 0 \iff \mu > 0.5 \quad \text{(new product is better)}.$$

I propose the following procedure:

Proposed Strategy

1) Initial n Exploratory Steps on New Product. Choose a small fixed integer n (for instance, $n = 10$). In the first n rounds, always offer the new product. Let S_n be the total number of successes during these n tries, so

$$\hat{\mu}_n = \frac{S_n}{n}$$

is an initial empirical estimate of the new product's success probability.

2) Bandit-style Indices (for $t > n$). From round $t = n + 1$ onward, treat the problem like a 2-armed bandit:

- **Arm 1 (Old Product)** has a known "reward" of 0.5, so its index is simply $\text{Index}_{\text{old}} = 0.5$.
- **Arm 2 (New Product)** is updated with an empirical mean and a confidence bonus. Specifically, if by round $t - 1$ I have tried the new product N_{t-1} times in total, with X_{t-1} successes, then

$$\hat{\mu}_{t-1} = \frac{X_{t-1}}{N_{t-1}}, \qquad \text{Index}_{\text{new}}(t) = \hat{\mu}_{t-1} + \sqrt{\frac{2\ln(t)}{2\,N_{t-1}}}.$$

At each step $t > n$, compare $\text{Index}_{\text{old}} = 0.5$ with $\text{Index}_{\text{new}}(t)$, and choose whichever is larger. Ties can be broken arbitrarily.

Pseudo-Regret Analysis

Denote by R_T the pseudo-regret up to time T, i.e. the difference between the expected number of successes of an optimal single choice (if μ were known) and that of my algorithm.

Case 1: $\mu > 0.5$ **(new product is better).** The best single choice is always picking the new product, with expected success μ each round. My algorithm invests n steps initially in the new product; that part is no problem if the new product is better, because I am effectively collecting reward near μ. After the first n rounds, the bandit step begins, but the new product's index is likely to exceed 0.5 fairly soon. Indeed, once the empirical mean $\hat{\mu}_{t-1}$ stabilizes around $\mu > 0.5$, the confidence bonus only makes the index bigger, ensuring that I select the new product almost every time. By standard bandit arguments, the number of times I might *fail* to choose the new product (e.g. if it momentarily loses to 0.5) is bounded by a constant (depending on $\mu - 0.5$). Hence the pseudo-regret R_T is $O(1)$ for $\mu > 0.5$.

Case 2: $\mu < 0.5$ **(new product is worse).** Now the best single choice is always the old product. Initially, I still do n test rounds with the new product, causing a regret on the order of $n\,(0.5 - \mu)$ from those forced tries. Then, in the subsequent bandit step, the UCB-based rule for the new product means I keep exploring it occasionally until I gather enough data to be confident that $\hat{\mu}_{t-1} + \sqrt{\frac{2\ln(t)}{2\,N_{t-1}}} < 0.5$. The standard analysis from *UCB* (or *LCB*) bandits says that the new product is pulled at most $O(\ln T)$ times if $\Delta = 0.5 - \mu$ is a positive gap. Summing the extra regret from each suboptimal pull yields $R_T = O(n + \ln T)$, but since n is a fixed constant, that is effectively $O(\ln T)$.

Conclusion. Hence:

$$R_T = \begin{cases} O(1), & \text{if } \mu > 0.5, \\ O(\ln T), & \text{if } \mu < 0.5, \end{cases}$$

which satisfies the requirement that the pseudo-regret is constant in the better-than-old case, and logarithmic in the worse-than-old case.

4 Empirical comparison of FTL and Hedge

4.1 (1) Influence of Different μ Values on the Regret

For each of the three values $\mu \in \{0.25, 0.375, 0.4375\}$, we generated i.i.d. Bernoulli(μ) sequences of length $T = 2000$ (with 10 runs). In Figure 2, we show the pseudo-regret over time for FTL and the different Hedge variants. The final average pseudo-regrets at $t = 2000$ are:

```
Final average pseudo-regret at t=2000 for p=0.25:
FTL: 0.200
Hedge_fixed(0.026327688477341595): 26.000
Hedge_fixed(0.05265537695468319): 13.400
Hedge_anytime(1.0): 4.600
Hedge_anytime(2.0): 1.400
```

```
Final average pseudo-regret at t=2000 for p=0.375:
FTL: 1.800
Hedge_fixed(0.026327688477341595): 26.800
Hedge_fixed(0.05265537695468319): 15.500
Hedge_anytime(1.0): 13.800
Hedge_anytime(2.0): 5.700

Final average pseudo-regret at t=2000 for p=0.4375:
FTL: 5.100
Hedge_fixed(0.026327688477341595): 29.700
Hedge_fixed(0.05265537695468319): 13.100
Hedge_anytime(1.0): 20.300
Hedge_anytime(2.0): 11.700
```

From these results, we see that smaller μ generally yields lower final pseudo-regret for FTL, while Hedge's performance depends heavily on its learning rate (η). Larger learning rates can hurt Hedge if μ differs significantly from 0.5, but the "anytime" versions tend to adapt better over time.

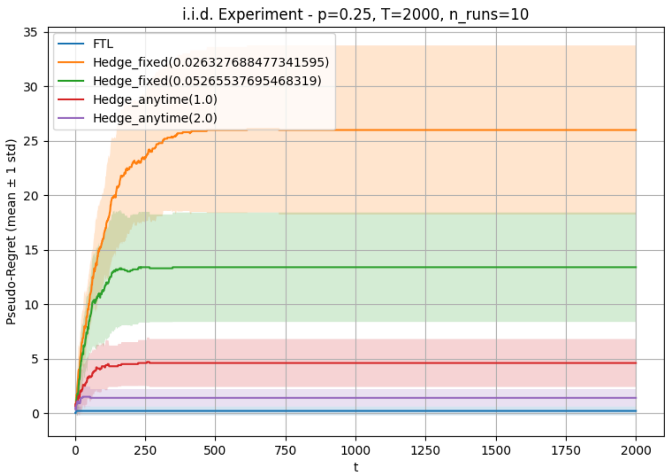

Figure 2: Example pseudo-regret curves for i.i.d. data with $\mu = 0.25$.

4.2 (2) Adversarial Sequence Design and Results

We designed an adversarial sequence by letting:

```
def simulate_adversarial_sequence(T):
    return np.array([t % 2 for t in range(1, T+1)])
```

Listing 2: Adversarial sequence alternating 0 and 1.

This sequence forces frequent switches if the algorithm tries to follow whichever label occurred more often so far. We ran the same 10 repetitions (the randomness comes from Hedge's sampling) and computed the *regret* (algorithm's loss minus the best constant expert's loss) at each time t. The final average regret at $t = 2000$ is:

```
Final average regret at t=2000 (adversarial):
FTL: 1000.000
Hedge_fixed(0.026327688477341595): -3.200
Hedge_fixed(0.05265537695468319): 29.900
Hedge_anytime(1.0): 16.200
Hedge_anytime(2.0): 13.500
```

We observe that FTL's regret is very large (around 1000), while some versions of Hedge achieve much smaller or even negative regret. This confirms that in certain adversarial designs, FTL can perform significantly worse than Hedge.

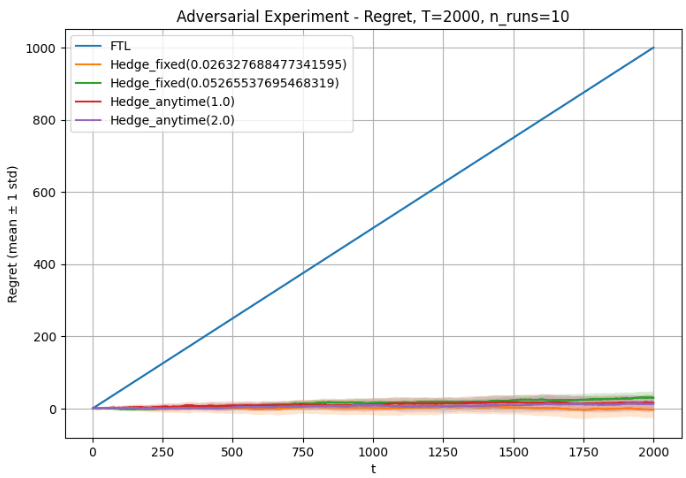

Figure 3: Regret over time on the adversarial sequence.

4.3 (3) Code Snippets and Observations

Below are the key code snippets for `FTL` and `Hedge` used in the experiments:

```
def ftl_predict(X):
    T = len(X)
    cum_loss = np.zeros(T)
    L0, L1 = 0, 0
```

```python
    mistakes = 0
    for t in range(T):
        if L0 < L1:
            pred = 0
        elif L1 < L0:
            pred = 1
        else:
            pred = 0
        if pred != X[t]:
            mistakes += 1
        if X[t] == 1:
            L0 += 1
        else:
            L1 += 1
        cum_loss[t] = mistakes
    return cum_loss
```

Listing 3: FTL implementation.

```python
def hedge_predict(X, schedule_type, param):
    T = len(X)
    L = np.zeros(2)
    cum_loss = np.zeros(T)
    mistakes = 0
    for t in range(1, T+1):
        if schedule_type == 'fixed':
            eta_t = param
        else:
            eta_t = param * np.sqrt(np.log(2)/t)
        L_min = np.min(L)
        w = np.exp(-eta_t * (L - L_min))
        p = w / np.sum(w)
        action = np.random.choice([0, 1], p=p)
        if action != X[t-1]:
            mistakes += 1
        loss0 = 1 if X[t-1] == 1 else 0
        loss1 = 1 if X[t-1] == 0 else 0
        L[0] += loss0
        L[1] += loss1
        cum_loss[t-1] = mistakes
    return cum_loss
```

Listing 4: Hedge implementation.

From these plots and results, we can see how the Hedge algorithm adapts better to adversarial changes, whereas FTL fails in the adversarial sequence. For the i.i.d. case, FTL often remains competitive, especially when μ is small, but Hedge can outperform it with properly tuned parameters.