Online and Reinforcement Learning (2025)
Home Assignment 4

Davide Marchi 777881

Contents

1 Policy Gradient Methods **2**
1.1 Baseline . . . 2
1.2 Lunar . . . 3

2 Improved Parametrization of UCB1 **5**

3 Introduction of New Products **6**

4 Empirical comparison of FTL and Hedge **8**

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

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

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:

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

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:

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

The first term is simply:

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

For the second term, using the chain rule,

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

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.

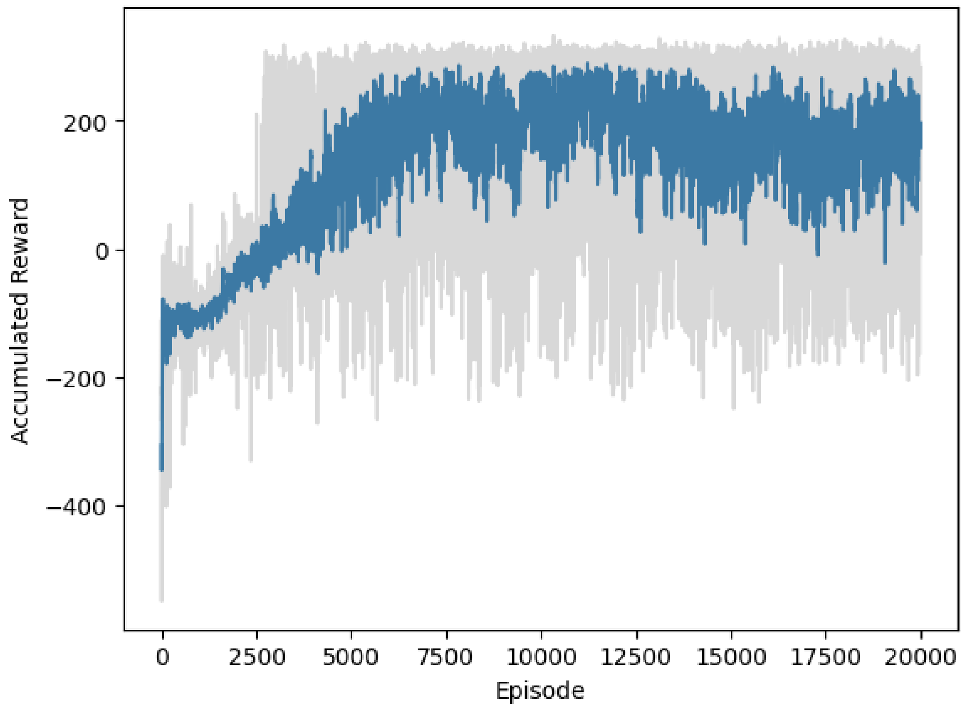

Figure 1: Accumulated reward over episodes.

2 Improved Parametrization of UCB1

(Optional)

3 Introduction of New Products

We consider a scenario with two products:

- **Old product:** Known success probability $p_{\text{old}} = 0.5$.
- **New product:** Unknown success probability μ.

Define $\Delta = 0.5 - \mu$. Our goal is to maximize the number of successes over T rounds by choosing, at each time t, one product to offer.

Algorithm Description

We adopt a two-phase strategy:

1. **Exploration:** In the first n rounds (with n a small constant), choose the new product to obtain an empirical estimate
$$\hat{\mu}_n = \frac{S_n}{n},$$
where S_n is the number of successes in these n trials.

2. **Exploitation:** For $t > n$, choose between the old and new products based on a UCB-type index. Define:
$$I_{\text{old}} = 0.5, \quad I_{\text{new}}(t) = \hat{\mu}_{t-1} + \sqrt{\frac{2 \ln t}{N_{\text{new}}(t-1)}},$$
where $N_{\text{new}}(t-1)$ is the number of times the new product has been chosen up to time $t-1$. At time t, select the product with the higher index.

Regret Definition

The pseudo-regret is defined as
$$R(T) = \mathbb{E}\left[T \max\{0.5, \mu\} - \sum_{t=1}^{T} r_t\right],$$
where r_t is the reward at time t. We analyze the two cases separately.

Case 1: $\mu < 0.5$ (New Product is Suboptimal)

In this case, the optimal product is the old one, and a regret of $\Delta = 0.5 - \mu$ is incurred each time the new product is selected. Let $N_{\text{new}}(T)$ be the total number of times the new product is chosen up to time T. Then,
$$R(T) = \Delta\,\mathbb{E}\left[N_{\text{new}}(T)\right].$$

With high probability, for all $t > n$, the number of times the new product is selected satisfies

$$\mathbb{E}\left[N_{\text{new}}(T)\right] \leq n + \frac{6 \ln T}{\Delta^2}.$$

Proof:

Using Hoeffding's inequality, for any $t > n$ we have:

$$\Pr\left(\hat{\mu}_{t-1} \geq \mu + \epsilon\right) \leq \exp\left(-2N_{\text{new}}(t-1)\epsilon^2\right).$$

By setting

$$\epsilon = \sqrt{\frac{2 \ln t}{N_{\text{new}}(t-1)}},$$

we ensure that, with high probability,

$$\hat{\mu}_{t-1} \leq \mu + \sqrt{\frac{2 \ln t}{N_{\text{new}}(t-1)}}.$$

Thus, the index for the new product satisfies

$$I_{\text{new}}(t) \leq \mu + 2\sqrt{\frac{2 \ln t}{N_{\text{new}}(t-1)}}.$$

The new product is selected only if

$$I_{\text{new}}(t) \geq 0.5,$$

which is equivalent to

$$\mu + 2\sqrt{\frac{2 \ln t}{N_{\text{new}}(t-1)}} \geq 0.5.$$

Rearranging and solving for $N_{\text{new}}(t-1)$ (and applying a union bound over t) yields

$$N_{\text{new}}(T) \leq n + \frac{6 \ln T}{\Delta^2},$$

in expectation.

Thus, the pseudo-regret is bounded by

$$R(T) \leq \Delta\left(n + \frac{6 \ln T}{\Delta^2}\right) = n\Delta + \frac{6 \ln T}{\Delta}.$$

Case 2: $\mu > 0.5$ (New Product is Optimal)

When $\mu > 0.5$, the new product is optimal. Regret is incurred only when the algorithm mistakenly selects the old product. Since the exploration phase forces the new product for n rounds and, after sufficient samples, the UCB index of the new product will, with high probability, exceed 0.5, the number of suboptimal choices is bounded by a constant independent of T. Therefore, the pseudo-regret is

$$R(T) = O(1).$$

Conclusion

Combining both cases, we conclude that the pseudo-regret satisfies

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

4 Empirical comparison of FTL and Hedge

In this exercise, we compare two expert algorithms, **Follow the Leader (FTL)** and **Hedge**, for predicting a binary sequence. We measure performance by the (pseudo)regret or regret against the best constant expert in hindsight. The code is contained in the file `FTL_and_Hedge.py`, and it tests both an i.i.d. Bernoulli setting (varying μ) and an adversarial setting designed to challenge FTL.

1) Implementation, Code Snippets, and i.i.d. Results

Key Code Snippets. Below are the core parts of the implementation. First, the **FTL** algorithm:

```
def ftl_predict(X):
    T = len(X)
    cum_loss = np.zeros(T)
    L0, L1 = 0, 0
    mistakes = 0
    for t in range(T):
        # Choose the leader (expert with fewer mistakes)
        if L0 < L1:
            pred = 0
        elif L1 < L0:
            pred = 1
        else:
            pred = 0  # tie-breaker

        # Check if we made a mistake
        if pred != X[t]:
            mistakes += 1

        # Update the experts' mistakes
        if X[t] == 1:
            L0 += 1
        else:
            L1 += 1

        cum_loss[t] = mistakes
    return cum_loss
```

Listing 2: FTL implementation.

Then, the **Hedge** algorithm with two experts (always predict 0 or always predict 1). The parameter η (learning rate) can be *fixed* or *anytime*:

```python
def hedge_predict(X, schedule_type, param):
    T = len(X)
    L = np.zeros(2)  # losses for expert0 and expert1
    cum_loss = np.zeros(T)
    mistakes = 0
    for t in range(1, T+1):
        # Define eta_t
        if schedule_type == 'fixed':
            eta_t = param
        else:  # 'anytime'
            eta_t = param * np.sqrt(np.log(2)/t)

        # Exponential weights
        L_min = np.min(L)
        w = np.exp(-eta_t * (L - L_min))
        p = w / np.sum(w)

        # Sample prediction
        action = np.random.choice([0,1], p=p)
        if action != X[t-1]:
            mistakes += 1

        # Update experts' losses
        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 3: Hedge implementation.

Finally, for each i.i.d. experiment, we run these algorithms on 10 independent Bernoulli(μ) sequences of length $T = 2000$, compute each algorithm's cumulative loss, and subtract the best constant expert's loss to obtain the *pseudo-regret*.

Numeric Values of Hedge Parameters. In the output, there are names like `Hedge_fixed(0.0263)`. The number in parentheses is the numeric value of η. We use the following formulas (with $T = 2000$ and $\ln(2) \approx 0.693$):

- $\eta = \sqrt{\frac{2\ln(2)}{T}} \approx 0.0263$
- $\eta = \sqrt{\frac{8\ln(2)}{T}} \approx 0.0527$
- For anytime Hedge: $\eta_t = c\sqrt{\frac{\ln(2)}{t}}$ with $c \in \{1, 2\}$.

Algorithm Name	**Formula**	**Value for $T = 2000$**
`Hedge_fixed(0.0263)`	$\sqrt{2\ln(2)/T}$	0.0263
`Hedge_fixed(0.0527)`	$\sqrt{8\ln(2)/T}$	0.0527
`Hedge_anytime(1.0)`	$1.0\sqrt{\ln(2)/t}$	*varies with t*
`Hedge_anytime(2.0)`	$2.0\sqrt{\ln(2)/t}$	*varies with t*

Table 1: Correspondence between Hedge formulas and the displayed numeric parameters.

Outputs and Plots (i.i.d. Setting). Below are the final average pseudo-regrets for $\mu = 0.25, 0.375, 0.4375$. Each block corresponds to 10 runs, each of length $T = 2000$:

```
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
```

Figures 2, 3, and 4 show the pseudo-regret curves over time for the three values of μ.

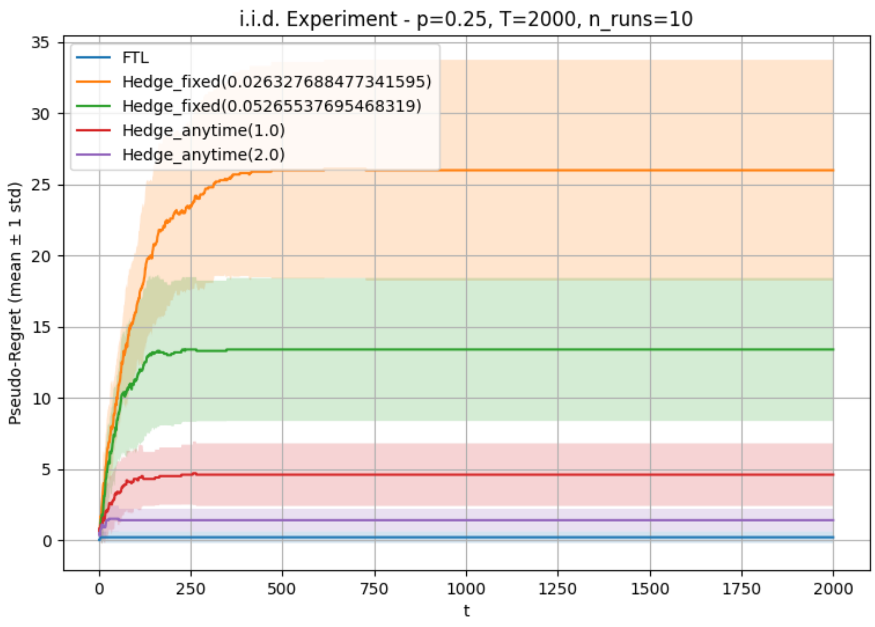

Figure 2: Pseudo-regret for $\mu = 0.25$.

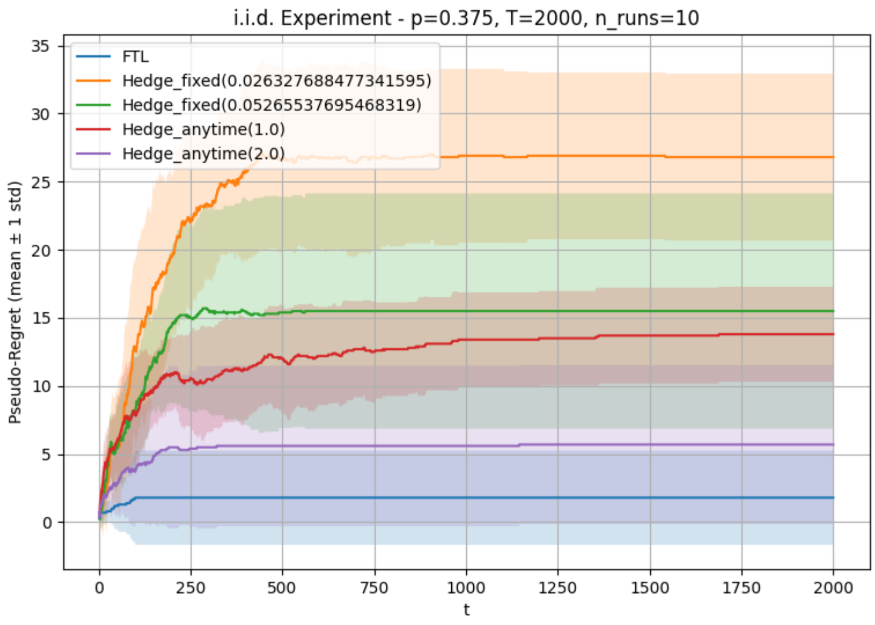

Figure 3: Pseudo-regret for $\mu = 0.375$.

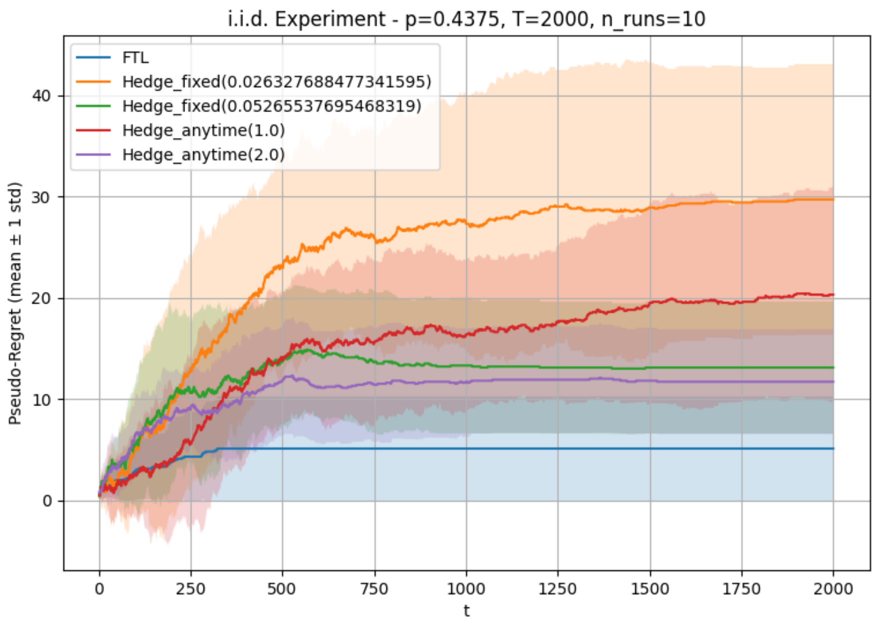

Figure 4: Pseudo-regret for $\mu = 0.4375$.

2) Influence of μ and Performance Over Time

Which values of μ lead to higher regret? The results indicate that when μ is closer to 0.5, the pseudo-regret tends to be higher. This is because when the distribution is more balanced, the difference between the mistakes of the constant experts (always predicting 0 or 1) becomes smaller, making it harder for the algorithms to quickly identify a clear majority, which in turn delays convergence and results in higher cumulative pseudo-regret.

Does the relative performance evolve with time? Yes. Over time, all algorithms tend to slow down the growth of their pseudo-regret as they collect more samples and better estimate the underlying distribution. In particular, once a sufficient number of observations have been collected, the algorithms' pseudo-regret curves start to plateau. Moreover, for smaller μ (i.e., when the distribution is more skewed), this plateau is reached earlier since the dominant class becomes apparent sooner. Conversely, when μ is closer to 0.5, it takes longer for the algorithms to settle, resulting in a slower reduction of the pseudo-regret growth rate.

3) Adversarial Case and Its Effect on FTL

We now consider an adversarial sequence. We define:

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

Listing 4: Adversarial sequence alternating 0 and 1.

In this case, we measure *regret* (algorithm's cumulative loss minus the best constant expert's loss). The final average regret at $t = 2000$ over 10 runs 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
```

Figure 5 shows the regret over time. We observe that **FTL** suffers heavily, reaching a regret of around 1000 by $t = 2000$. Meanwhile, some Hedge variants maintain very low or even negative regret (meaning they do better than the best constant expert).

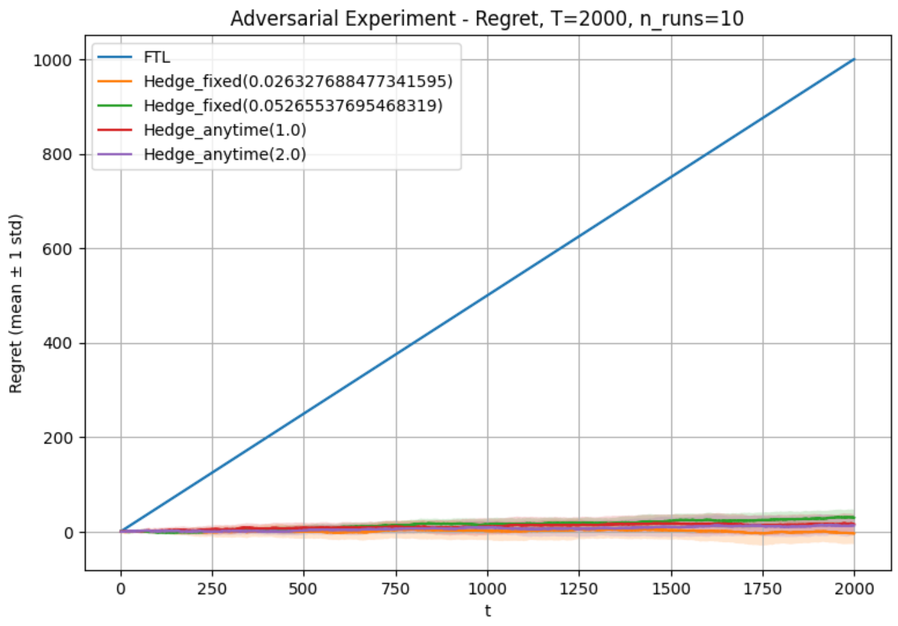

Figure 5: Regret over time on the adversarial sequence (alternating 0,1).

These results highlight that FTL can be outperformed by Hedge in adversarial scenarios. The exact extent depends on the learning rate choice, but `Hedge_anytime(1.0)` and `Hedge_anytime(2.0)` generally avoid large regret growth.

Note: All code and further details can be found in `FTL_and_Hedge.py`. That can be run to replicate these experiments.