Reinforcement Learning

Lab #1: Prediction and Control

Daniil Arapov

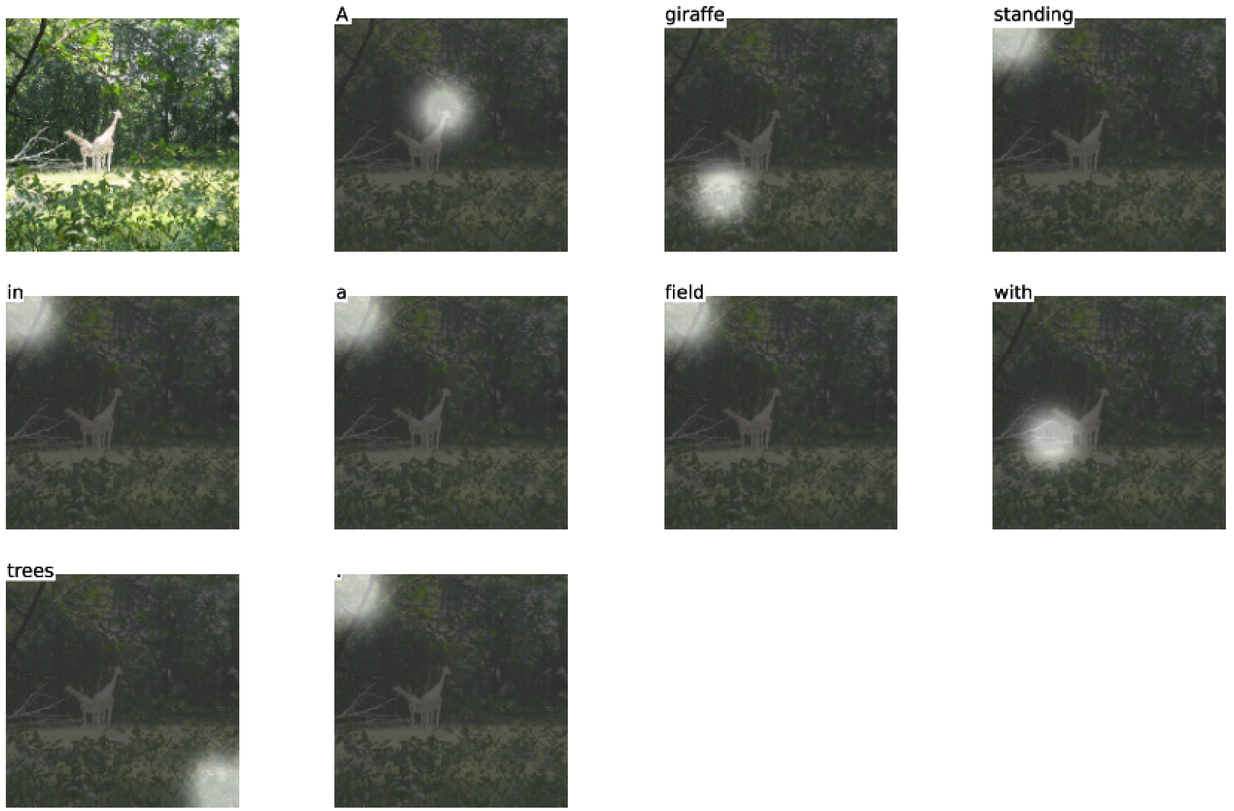
Partially based on DeepMind lecture #2

Test questions for today:

1. What hypothesis lying in a heart of RL is a Gödel Problem?
2. What is a policy?
3. What is a value function?
4. Describe the difference between v_π and $v_{\pi*}$,
   given π and $\pi*$ - arbitrary policies.
5. What are *exploration* and *exploitation*?
6. What are *prediction* and *control*?

More RL examples

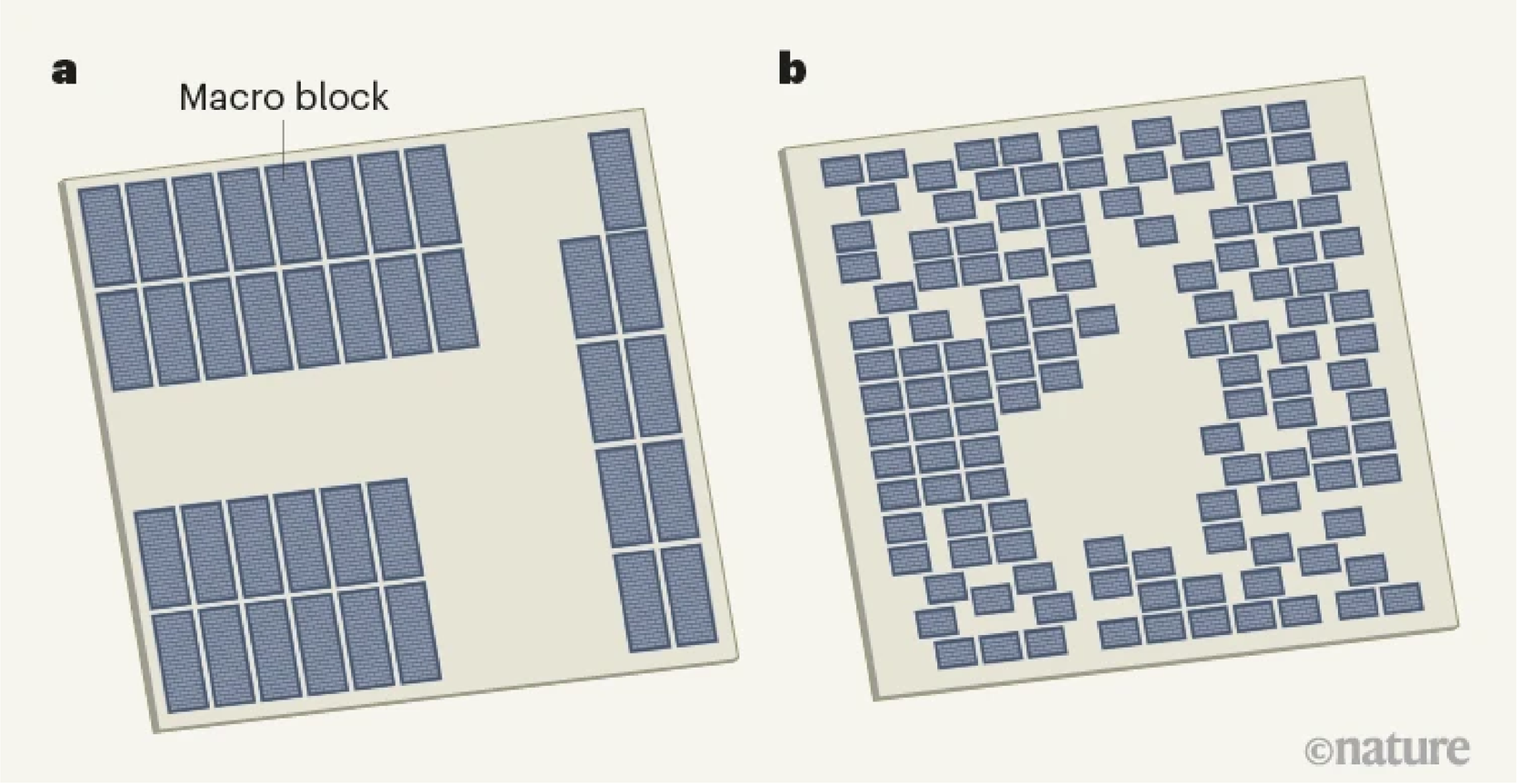
Non-differentiable functions



(a) A giraffe standing in the field with trees.

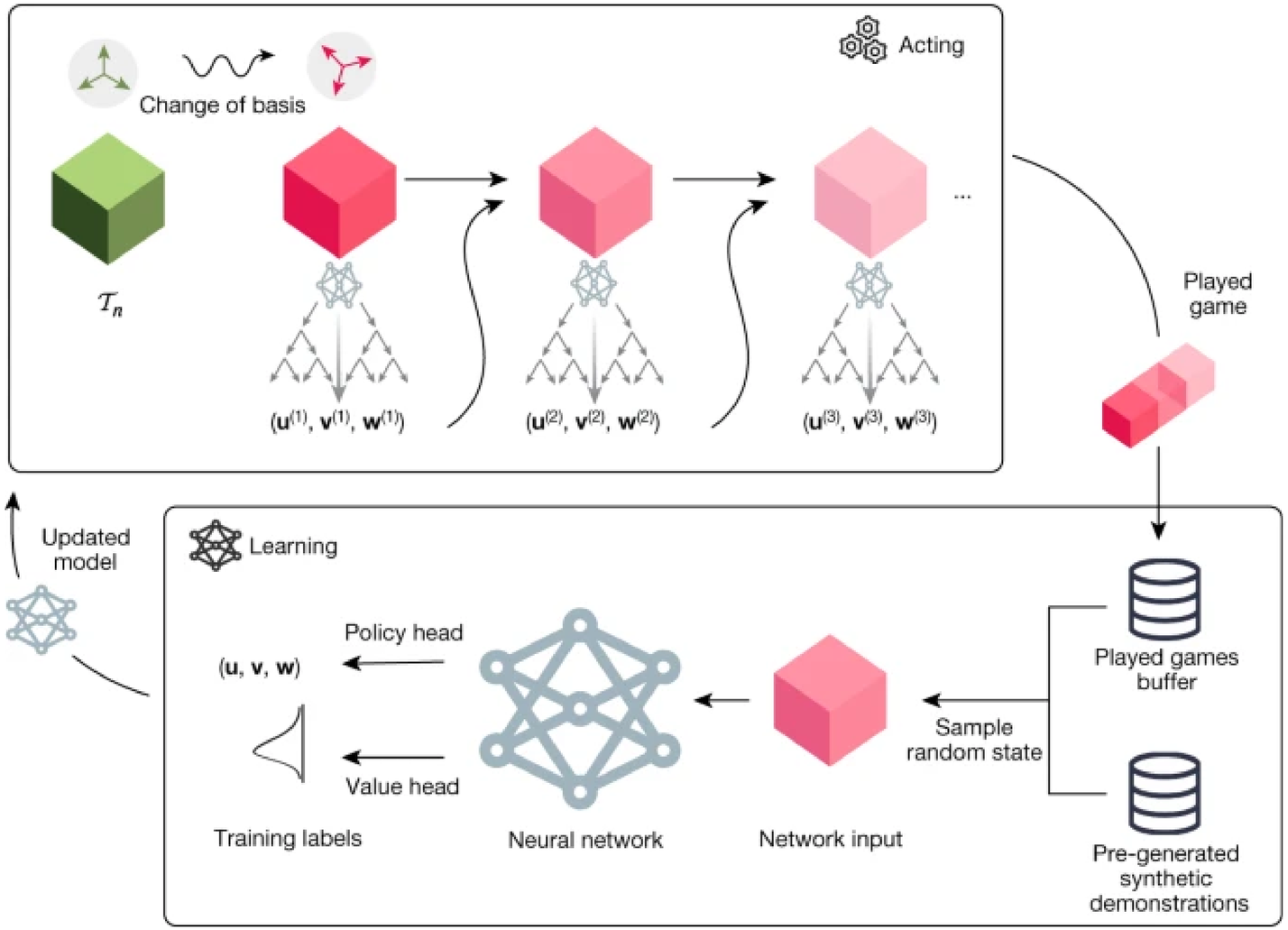
More RL examples

Specific optimization (i.e., designing chips)



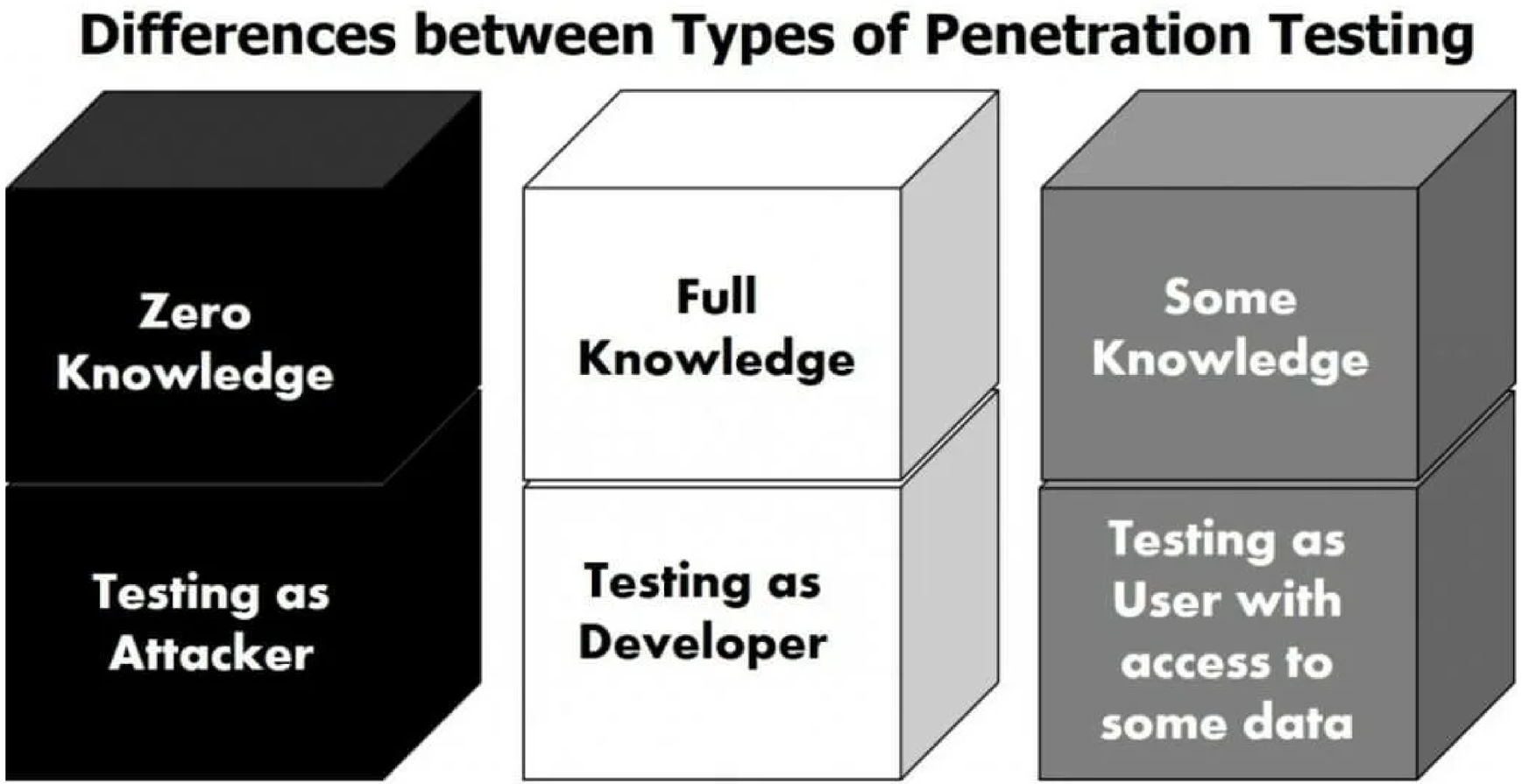
More RL examples

New matrix multiplication methods!



More RL examples

Information security applications?



https://blogs.getcertifiedgetahead.com/wp-content/uploads/2016/01/black-box-testing-and-more-1024x545.jpg

Prediction and Control

Prediction: Understanding what is going to happen.

Control: Changing behaviour to improve the results.

Examples:

- If you keep sleeping in lectures, your lab grade will be low
  -> You should be attentive
- If you keep smoking, your health will degrade
  -> You should smoke less
- If you keep undersleeping, your productivity will be worsening
  -> You should sleep more

Multi-armed bandit problem statement

- A multi-armed bandit is a set of distributions $\{\mathcal{R}_a | a \in \mathcal{A}\}$
- $\mathcal{A}$ is a (known) set of actions (or “arms")
- $\mathcal{R}_a$ is a distribution on rewards, given action a
- At each step t the agent selects an action $A_t \in \mathcal{A}$
- The environment generates a reward $R_t \sim \mathcal{R}_{A_t}$
- The goal is to maximise cumulative reward $\sum_{i=1}^{t} R_i$
- We do this by learning a **policy**: a distribution on $\mathcal{A}$

Multi-armed bandit environment

```
class UniformMultiArmedBandit:

    class Arm:

        def __init__(self, cost, distribution):
            self.cost = cost
            self.distribution = distribution

        def trigger(self):
            return self.distribution.rvs(1)[0] - self.cost
```

Multi-armed bandit environment

```
def __init__(self, n_arms, costs = None, distributions = None):

    if not (distributions is None):
        assert len(distributions) == n_arms, "Distributions size should be equal to n_arms"
    else:
        a_list = np.randint(1, 5, n_arms)
        b_list = np.randint(5, 10, n_arms)
        distributions = [stats.randint(a, b) for a, b in zip(a_list, b_list)]

    if not (costs is None):
        assert len(costs) == n_arms, "Costs size should be equal to n_arms"
    else:
        costs = np.randint(3, 8, n_arms)

    self.n_arms = n_arms

    self.arms = []
    for i in range(n_arms):
        new_arm = self.Arm(costs[i], distributions[i])
        self.arms.append(new_arm)

def trigger(self, idx):
    return self.arms[idx].trigger()
```

One arm example

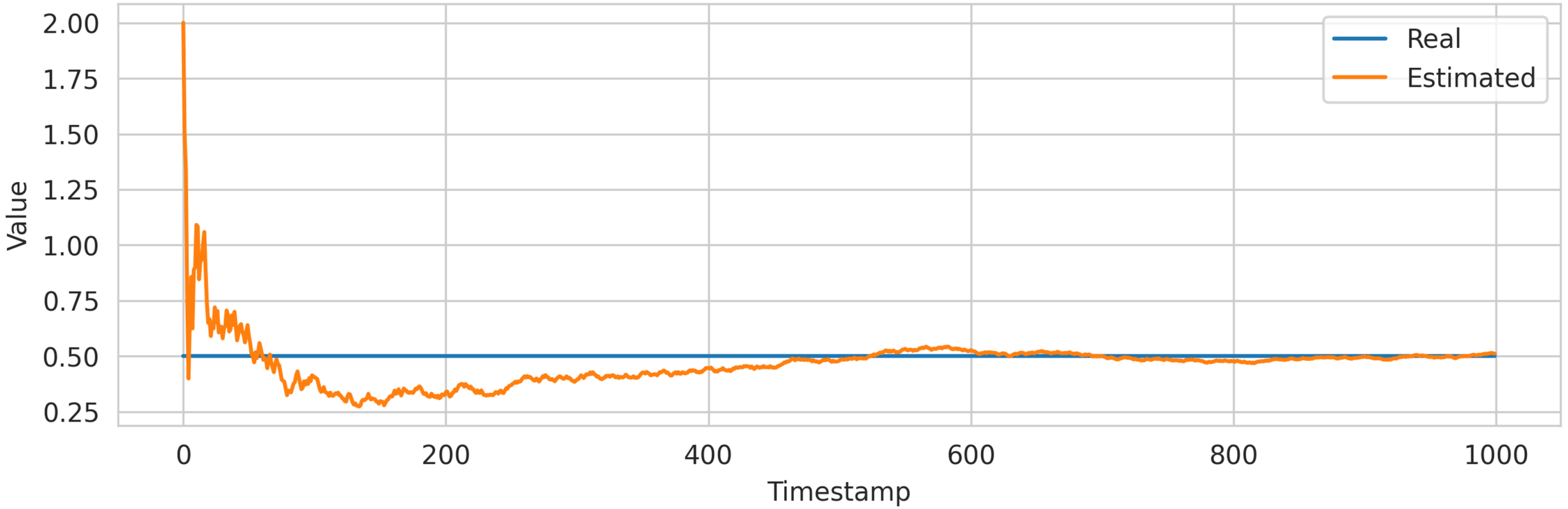
```
cost = 3
distribution = stats.randint(1, 7)

bandit = UniformMultiArmedBandit(1, [cost], [distribution])
```

```
real_exp = distribution.mean() - cost
print(real_exp)
```

```
0.5
```

```
N = 1000
records = []
means = []

for i in range(N):
    records.append(bandit.trigger(0))
    means.append(np.mean(records))
```

One arm example



$N \to \infty$ results in exact estimation

Having multiple arms, when one should switch an arm?

Depends on the policy...

Greedy policy

```
n_arms = 4
bandit = UniformMultiArmedBandit(n_arms)
```

```
rewards = [[bandit.trigger(i)] for i in range(n_arms)]
```

```
rewards
```

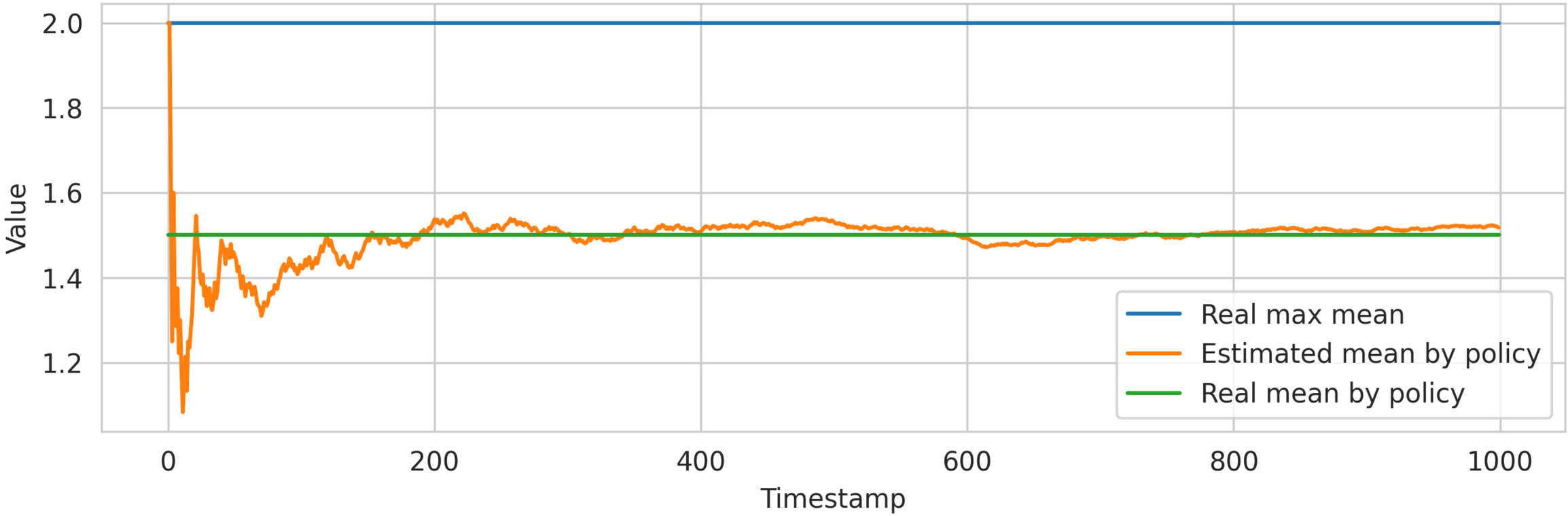
```
[[0], [0], [0], [2]]
```

```
def greedy_policy(rewards):
    means = []
    # mean for each action
    for rew_list in rewards:
        means.append(np.mean(rew_list))

    return np.argmax(means)
```

Greedy policy

```
# mean reward for each arm
actual_means = [arm.distribution.mean() - arm.cost for arm in bandit.arms]
# max mean reward
real_max_exp = max(actual_means)

estimated_mean_policy = []
real_mean_policy = []

N = 1000

for i in range(N):
    # choosing an action
    action = greedy_policy(rewards)

    # recording the results
    estimated_mean_policy.append(np.mean(rewards[action]))
    rewards[action].append(bandit.trigger(action))
    real_mean_policy.append(actual_means[action])
```

Greedy policy - local minima!



Epsilon-greedy policy

Almost the same as greedy, but epsilon chance to act randomly.

```
def epsilon_greedy_policy(rewards, n_arms, epsilon):
    means = []
    # mean for each action
    for rew_list in rewards:
        means.append(np.mean(rew_list))
    main_action = np.argmax(means)

    probas = [epsilon / n_arms] * n_arms
    probas[main_action] += (1 - epsilon)
    return np.array(probas)
```

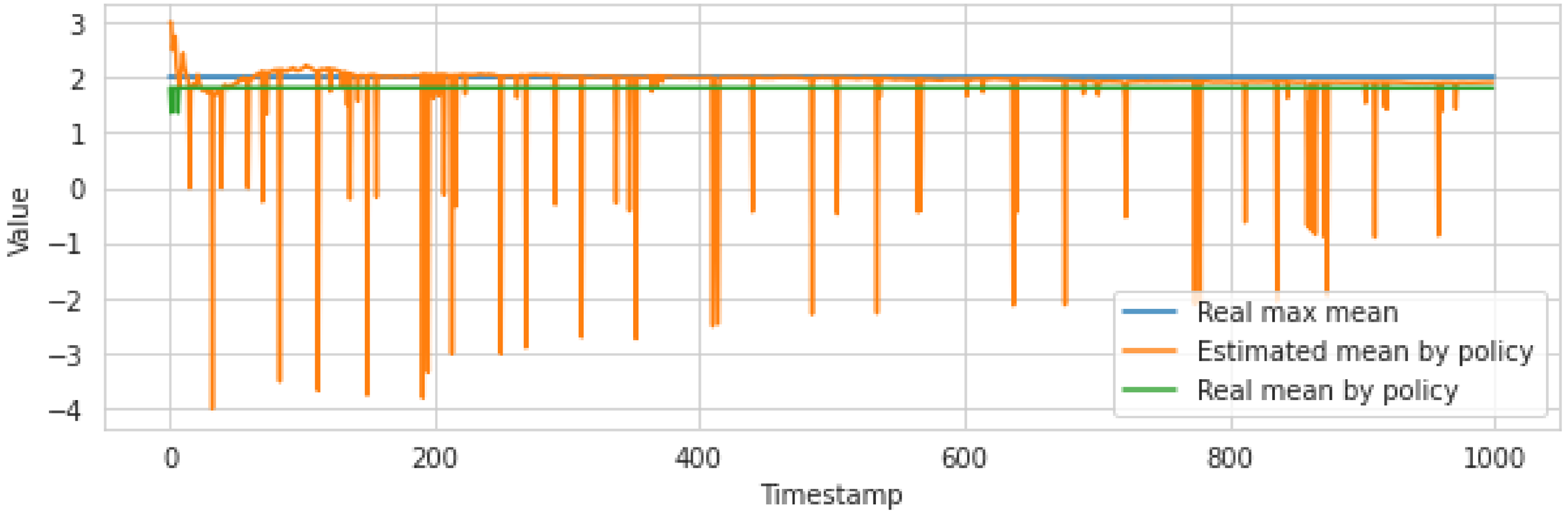
Epsilon-greedy policy

What are the differences from just a greedy policy?

```
for i in range(N):
    # choosing an action
    action_probas = epsilon_greedy_policy(rewards, n_arms, epsilon = 0.1)
    action = np.random.choice(n_arms, size = 1, p = action_probas)[0]

    # recording the results
    estimated_mean_policy.append(np.mean(rewards[action]))
    rewards[action].append(bandit.trigger(action))
    real_mean_policy.append((actual_means * action_probas).mean())
```

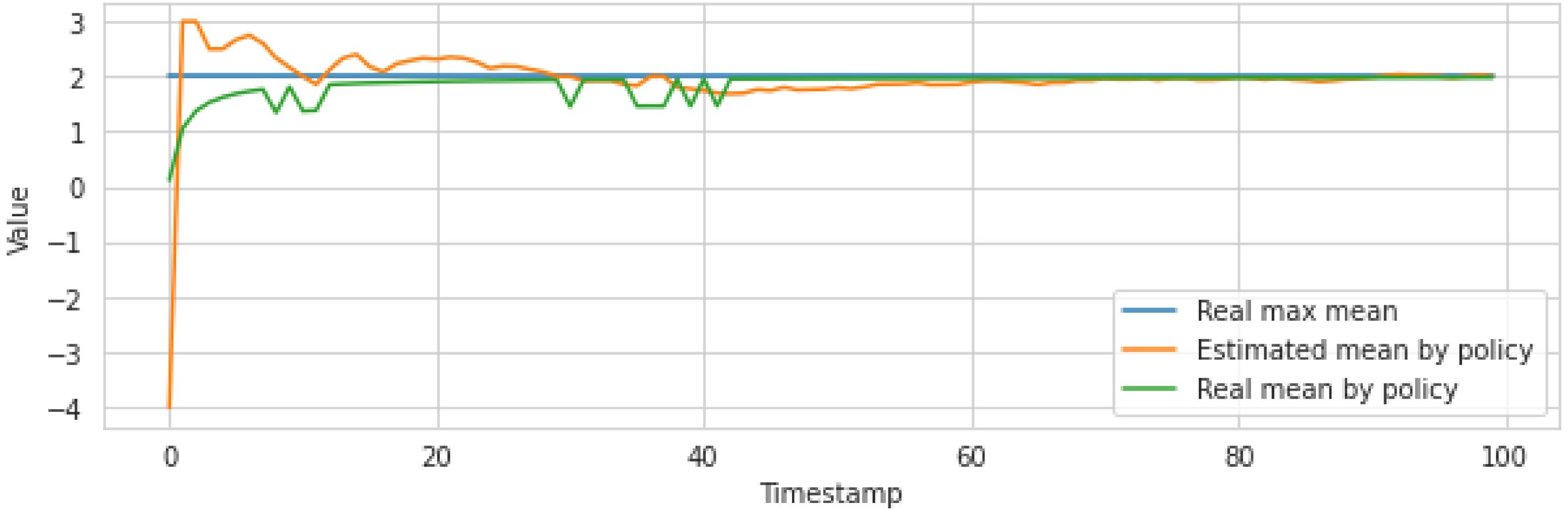
Epsilon-greedy policy plots

Whaaaaat?



Epsilon-greedy policy plots (decreasing epsilon)

Increasing exploitation over the time

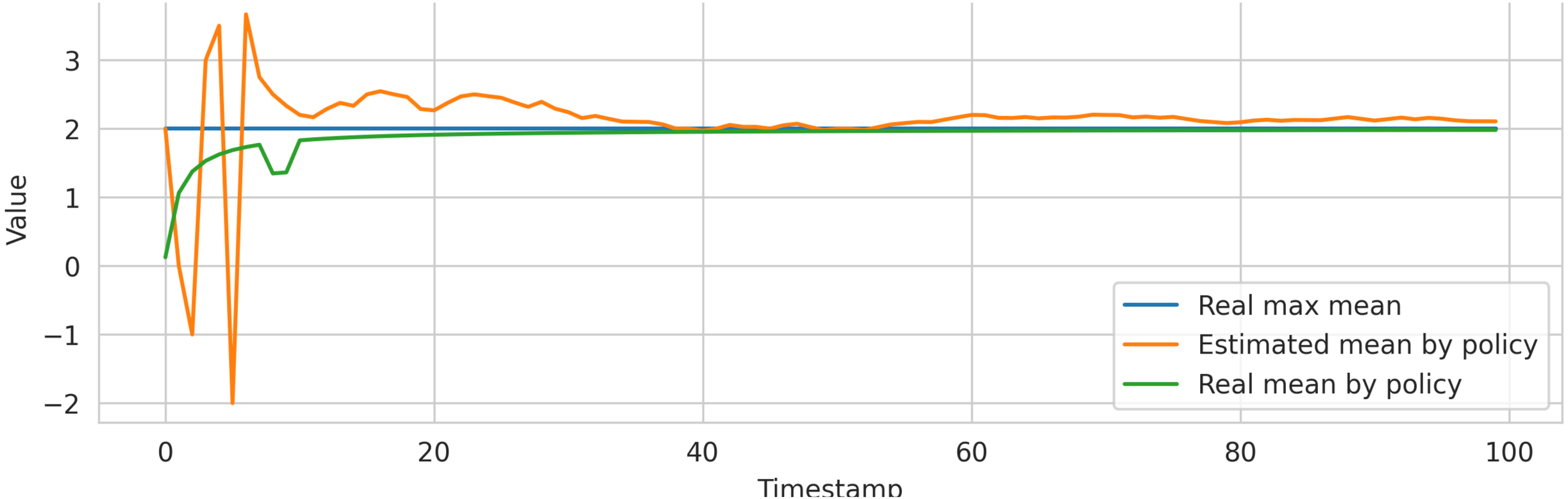


Softmax-based policy

Tend to explore values close to maximum

```
def softmax_policy(rewards):
    means = []
    # mean for each action
    for rew_list in rewards:
        means.append(np.mean(rew_list))

    return np.softmax(means)
```

Softmax-based policy



Softmax-based policy

What if two mean values are close to each other? ***Constant exploration.***

Is there a way to improve softmax so that having two close values we will tend to prefer the greater one? **Exploration / exploitation** in softmax.

So, overfitting is not always bad?

Policy gradients

Instead of computing softmax over the estimated mean values, one may try to learn parameters for each of the actions:

- Can we learn policies $\pi(a)$ directly, instead of learning values?
- For instance, define **action preferences** $H_t(a)$ and a policy

$$\pi(a) = \frac{e^{H_t(a)}}{\sum_b e^{H_t(b)}}$$

- The preferences are not values: they are just learnable policy parameters
- Goal: learn by optimising the preferences
- In the bandit case, we want to update:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \mathbb{E}[R_t | \pi_{\theta_t}],$$

Uncertainty

Previous two methods do not tend to consider non-explored actions granted low reward. This is the problem because greatest-mean solution may have randomly generated low-value point. So, we have to explore all the actions ***enough*** to be more or less sure that there are no ”surprises” due to underexplored actions.

How to estimate how much knowledge do we have?
Bayesian statistics will help us! It is similar to the way our brain works.

Bayesian

Having some prior knowledge and specific data, we draw the conclusions.

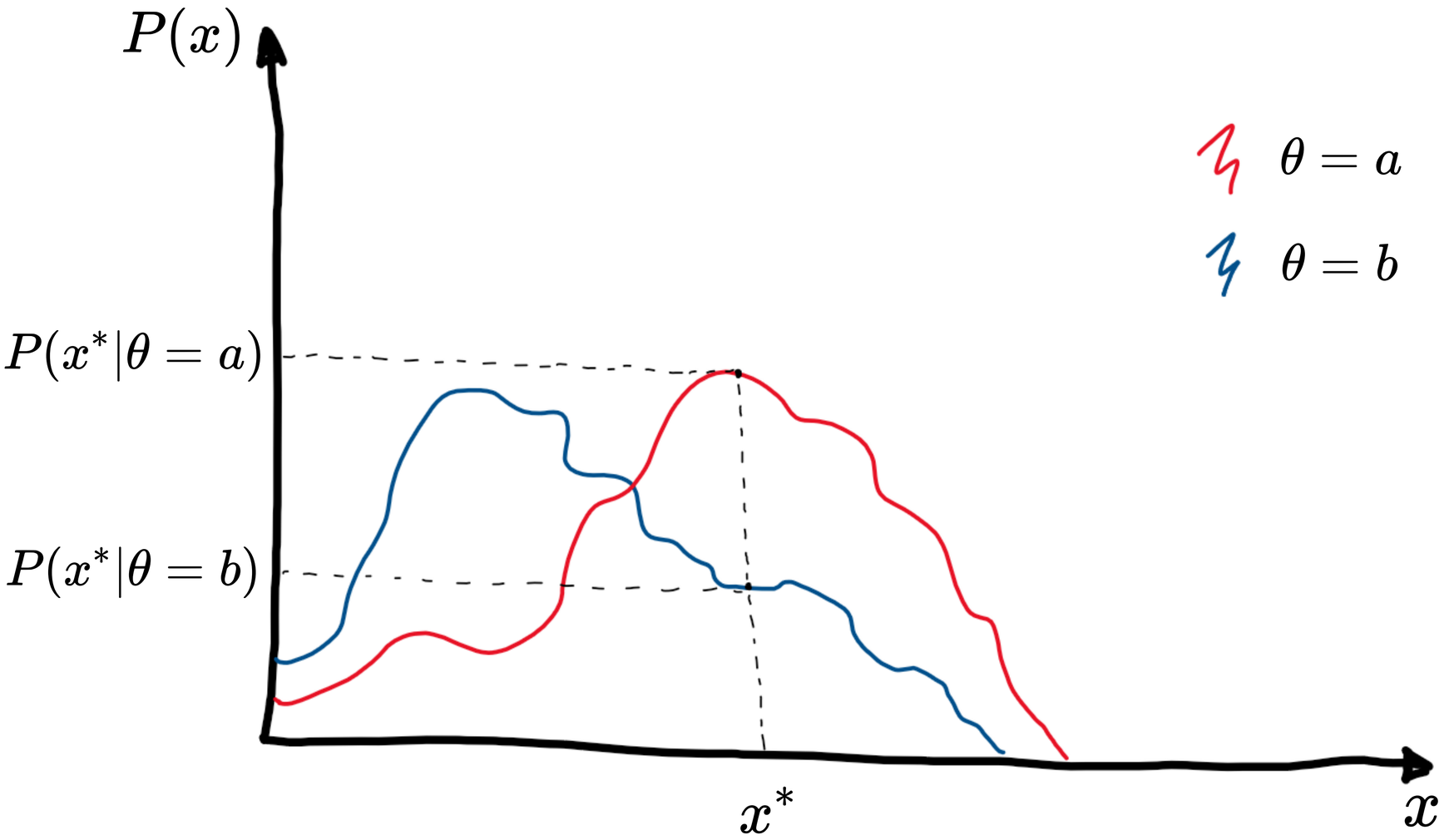
For example, we have prior knowledge that in 70% of Russian families there is just one child. Specific data - some family **X** has 15 pairs of child shoes. Having that additional information we may assume that family **X** has one child with “low” probability rather than having two or three children. However, we need additional information. If additional info does not contradict to our thoughts, we will be more confident. Otherwise, having contradictory data (i.e. 15 pairs of child shoes and 1 child hat), we will need more info to draw final conclusion. Bayesian statistics works similarly.

Bayesian

Speaking of Bayesian Statistics we are usually interested to determine distribution out of which we have sampled our points. If we know the exact distribution of values in arm, we know the mean of the arm.

Getting more and more samples, we will become much more confident out of which distribution our data has commen. Also, we may introduce uncertainty.

Comparing distribution probabilities



$\theta = a$

$\theta = b$

What distribution generated x*?

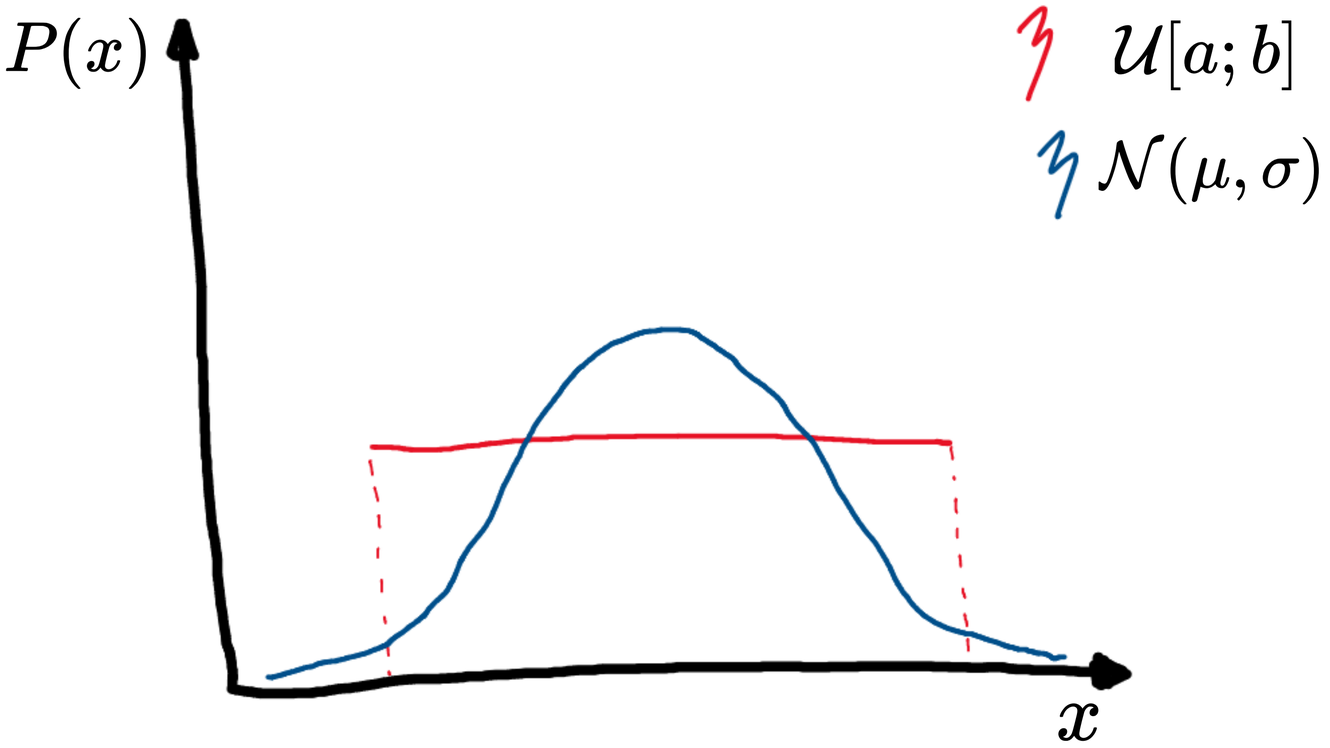
Consider relative probability:

$$P_{rel}(x^*) = \frac{P(x^*|\theta = a)}{P(x^*|\theta = b)}$$

Having several data points:

$$P_{rel}(x_1, \ldots, x_n) = \prod_{i=1}^{n} \frac{P(x_i|\theta = a)}{P(x_i|\theta = b)}$$

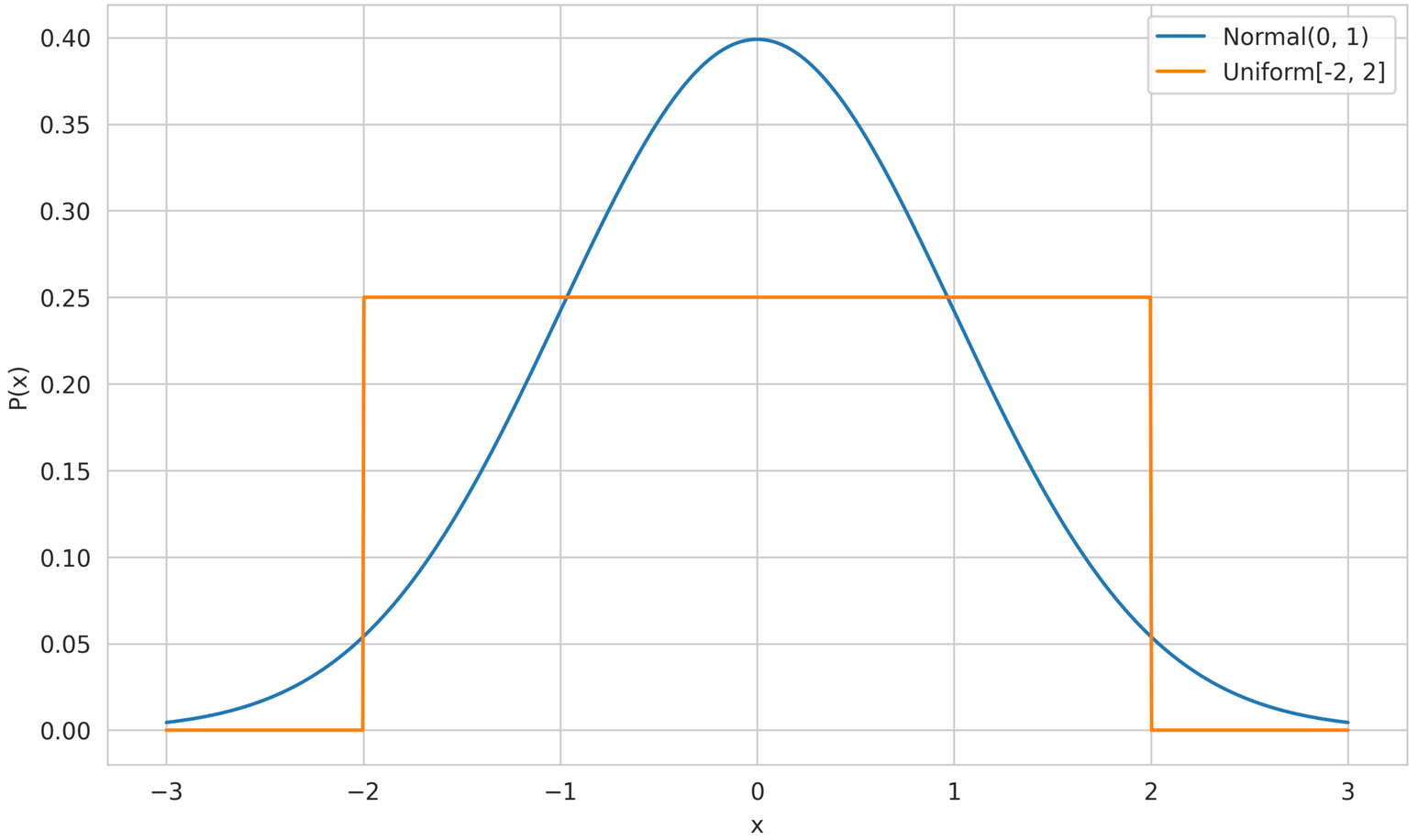
Normal vs Continuous Uniform distributions



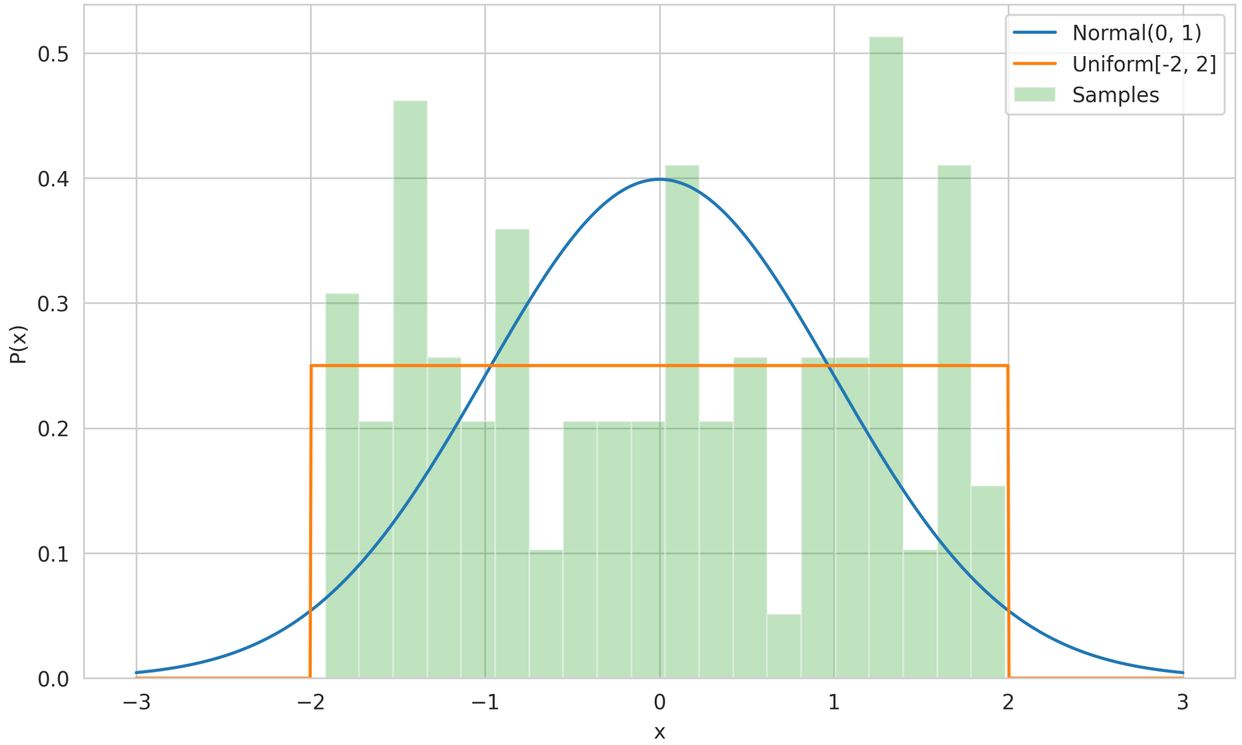
Considering relative probability,
what happens if original distribution is:

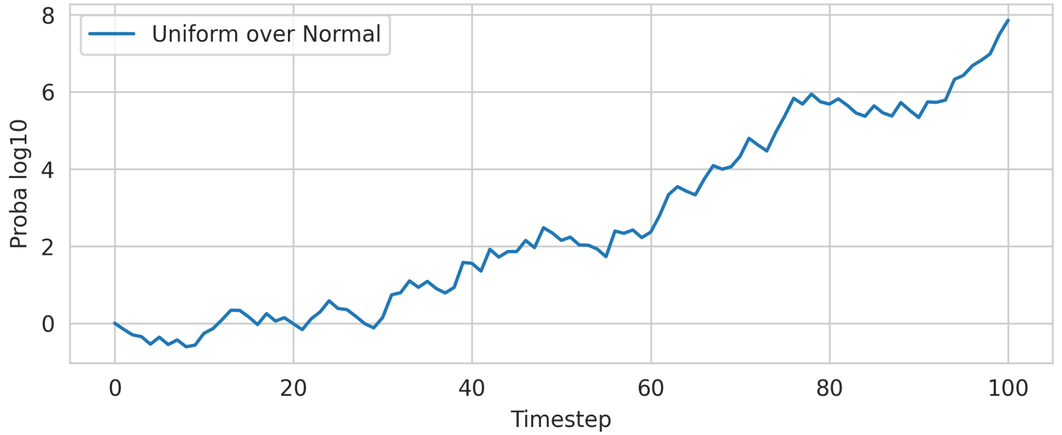
- Continuous Uniform distribution?
- Normal distribution?

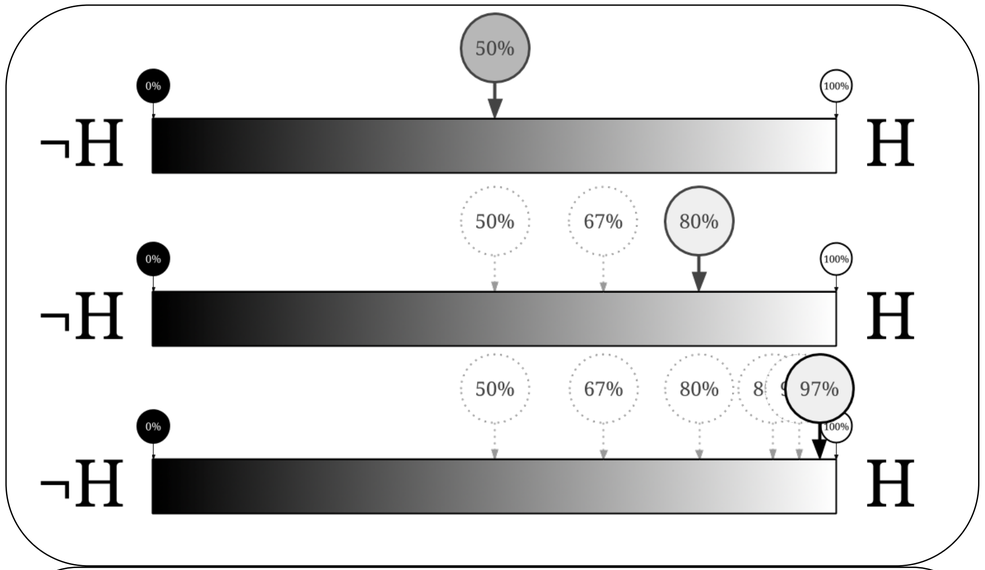
Normal vs Continuous Uniform distributions (in code)



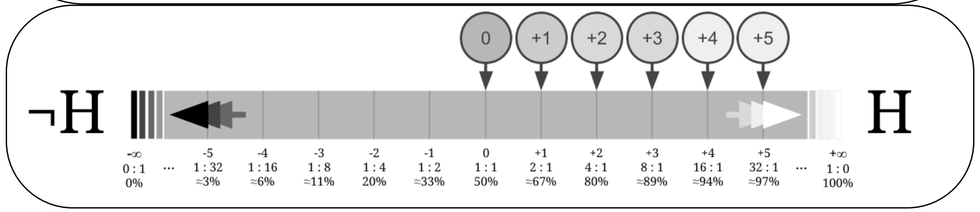
Normal vs Continuous Uniform distributions (Uniform)



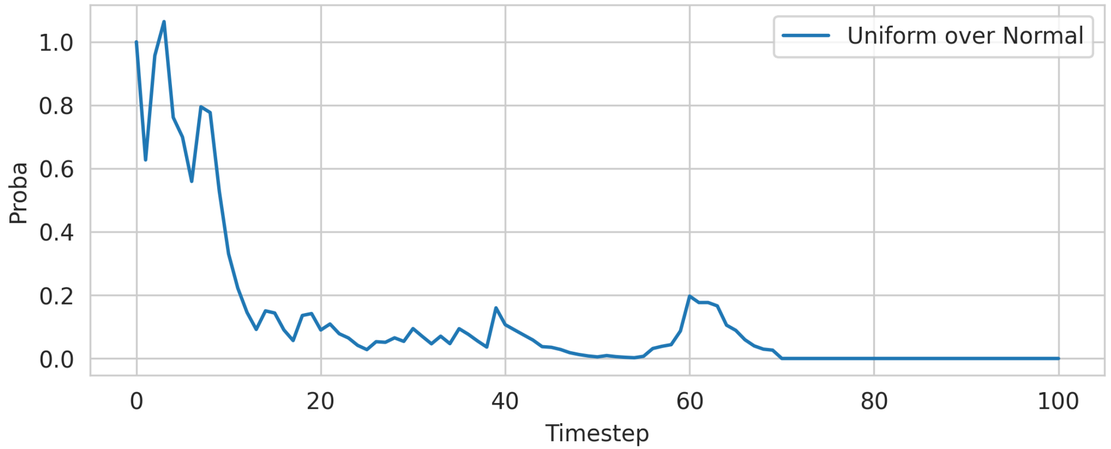




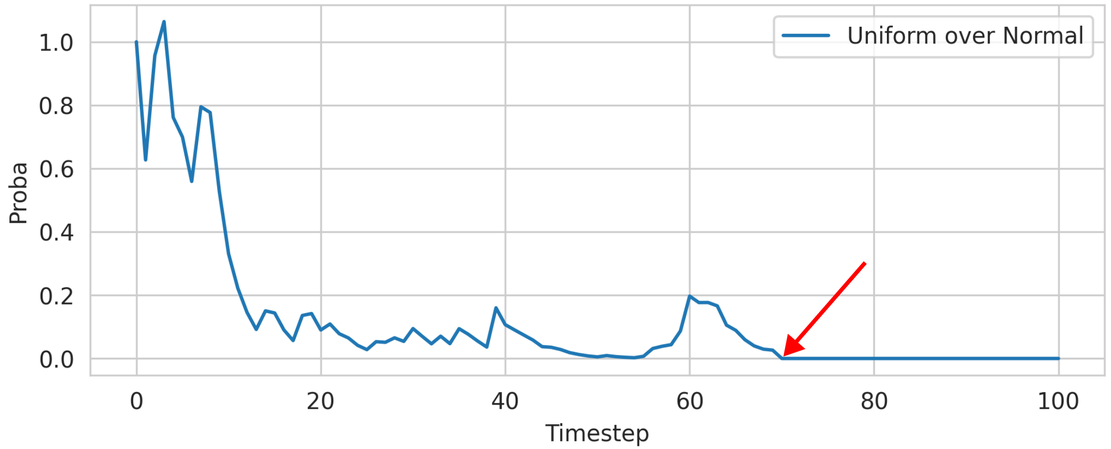
https://arbital.com/p/bayes_log_odds/

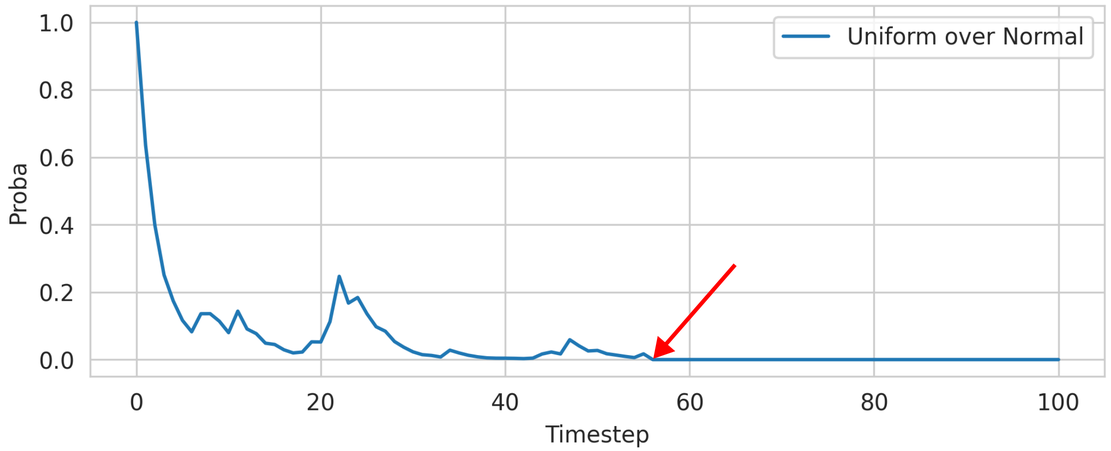


Normal vs Continuous Uniform distributions (Normal)

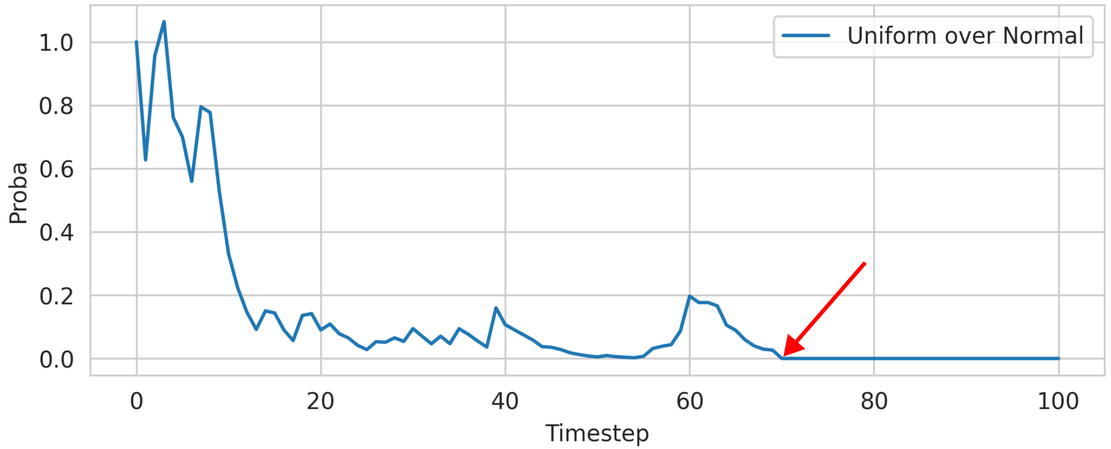


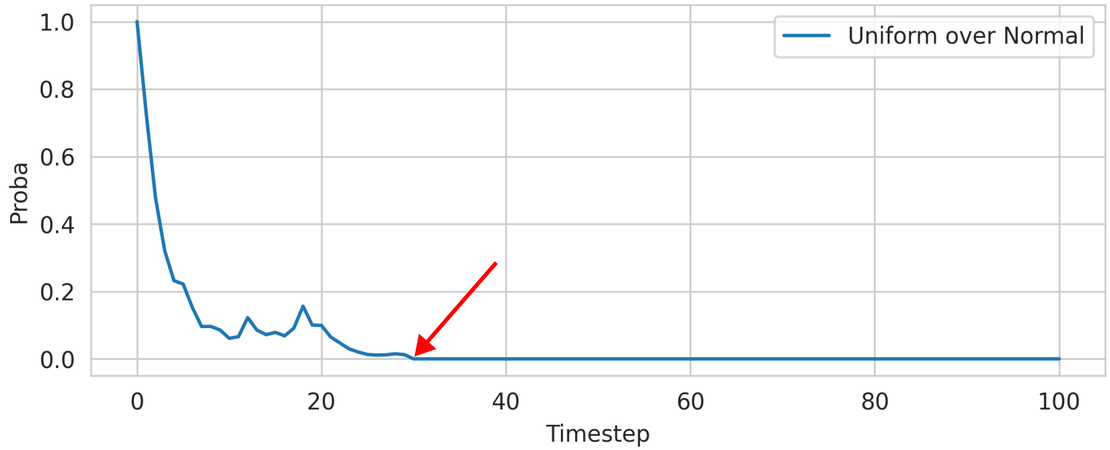
Normal vs Continuous Uniform distributions (Normal)

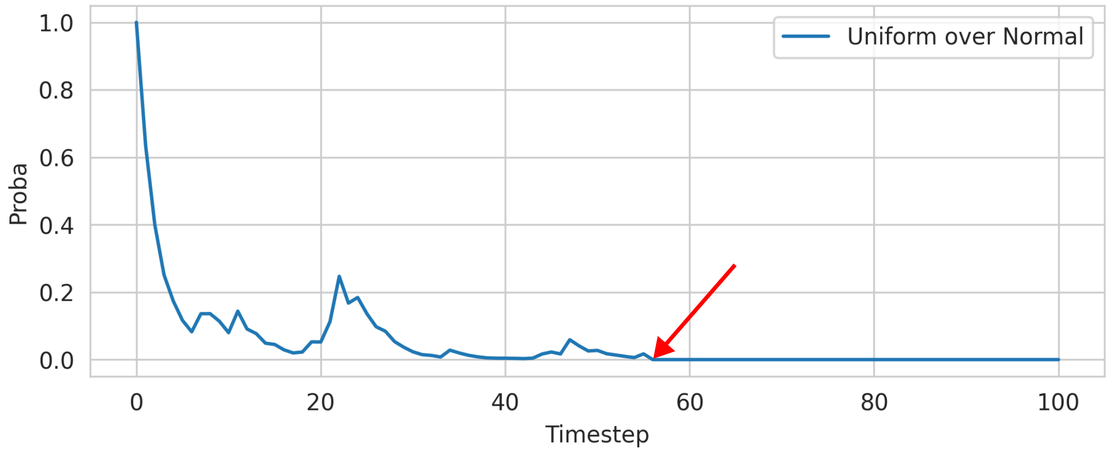




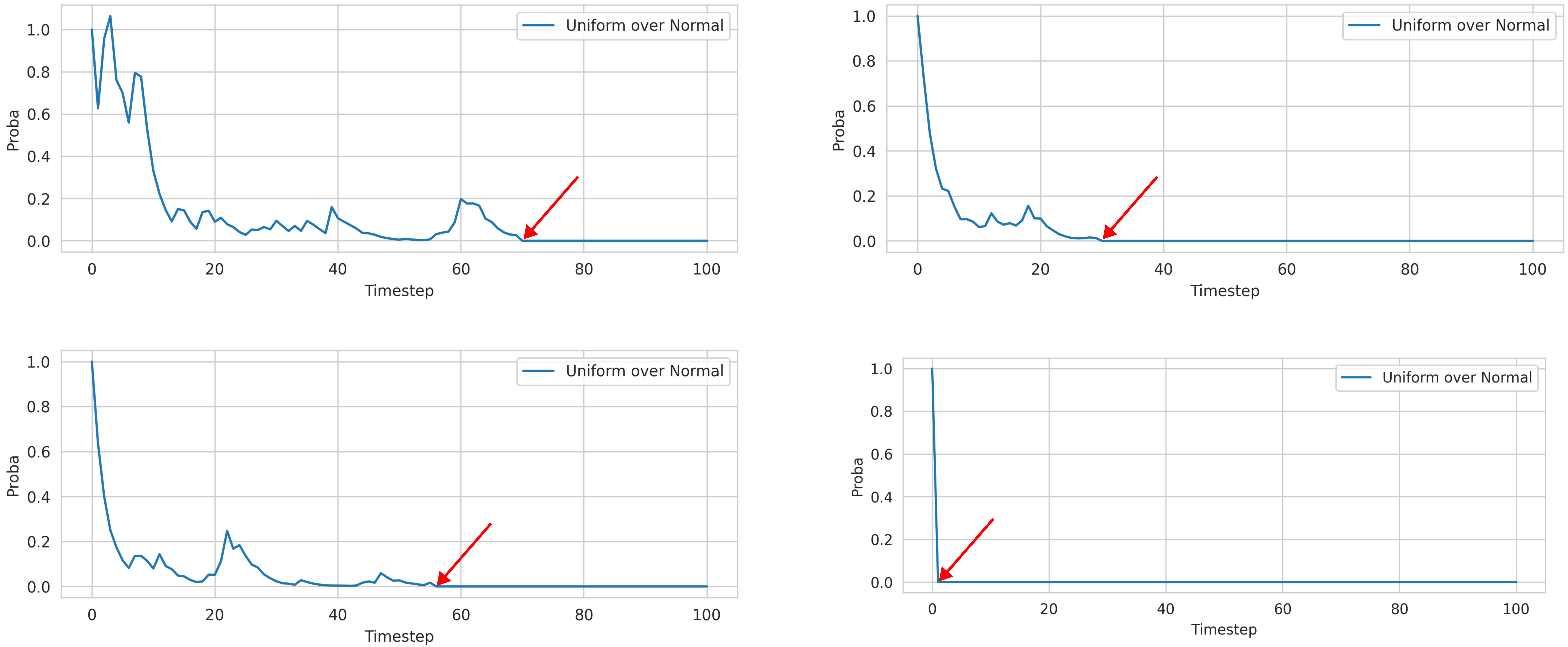
Normal vs Continuous Uniform distributions (Normal)





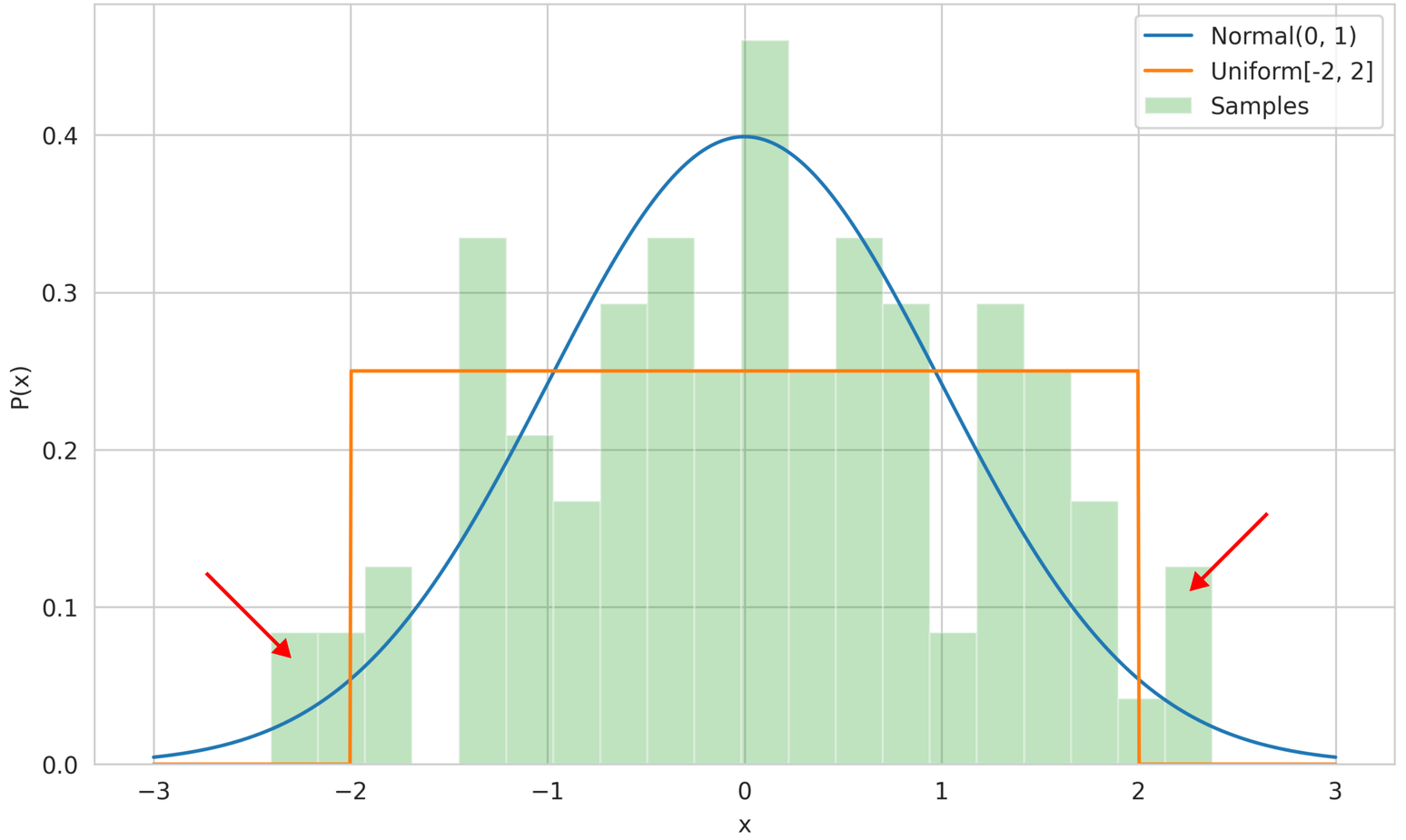


Normal vs Continuous Uniform distributions (Normal)



Why is it exactly zero?

Normal vs Continuous Uniform distributions (Normal, explanation)



Coming back to the probability space

Consider all the possible outcomes:

$$P(\theta_i|x^*) = \frac{P(x^*|\theta_i)}{\sum\limits_{\theta_j \in \Theta} P(x^*|\theta_j)}$$

$$P(\theta^*|x^*) = \frac{P(x^*|\theta^*)}{\int\limits_{\Theta} P(x^*|\theta)\, d\theta}$$

Transition to distribution of distributions' statistics is straightforward.

Discrete uniform distribution bayesian multi-armed bandit (0 samples)

Let us parametrize Uniform distribution as **loc**, **scale**:

- 0 <= **loc** <= 1000;
- 0 <= **scale** <= 1000;
- **loc** + **scale** <= 1000.

We have ~500’000 possible pairs of **loc** and **scale**.

Discrete uniform distribution bayesian multi-armed bandit (1 sample)

Having first sample **s1**, we observe the following:

- some pairs have probability of 0 (***example?***)
- **loc** = **s1**, **scale** = 0 - proba of **s1** = 1
- what is proba formula for (**loc***, **scale***) given **s1** lies in it?

Discrete uniform distribution bayesian multi-armed bandit (1 sample)

```
sample = bandit.trigger(0) + cost
print("Returned value:", sample)
```

```
Returned value: 2
```

```
from tqdm.auto import tqdm

probabilities = np.ones((1000, 1000))

for loc in tqdm(range(1000)):
    for scale in range(1000):
        probabilities[loc, scale] *= stats.randint.pmf(sample, loc, loc + 1 + scale)
```


```
probabilities[sample, 0]
```

```
1.0
```

```
probabilities[sample - 1, 1]
```

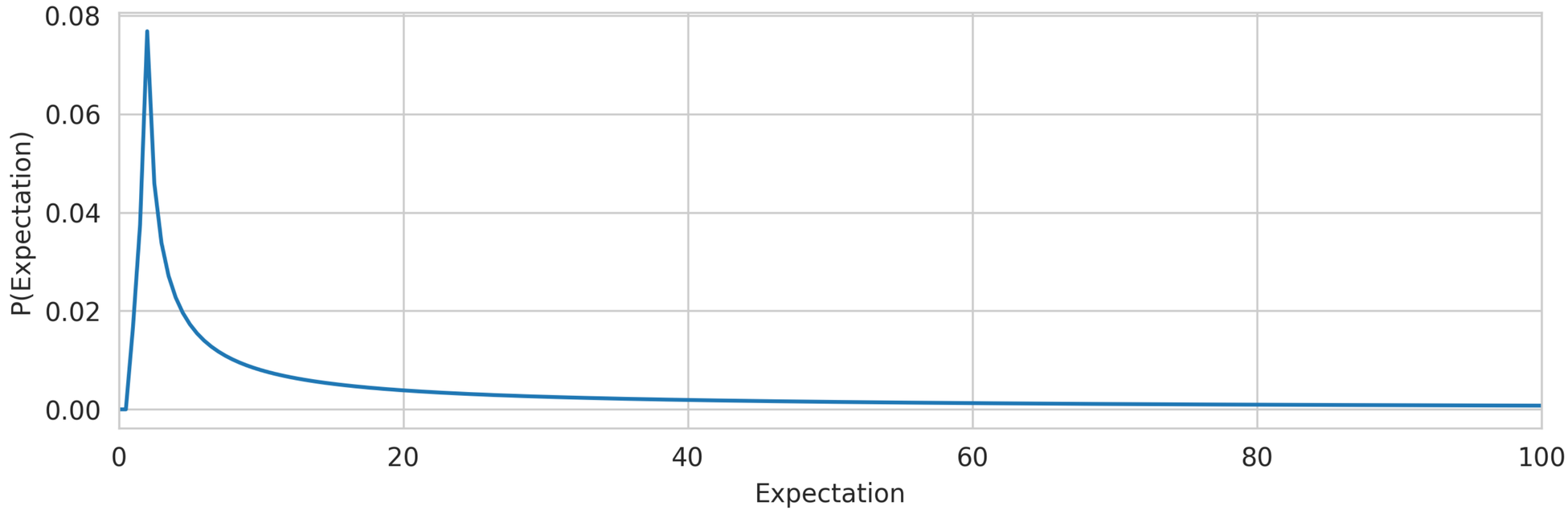
```
0.5
```

```
probabilities[sample, 1]
```

```
0.5
```

Discrete uniform distribution bayesian multi-armed bandit (1 sample)

```python
real_probas = probabilities / probabilities.sum()

expected_mean_dict = {}

for loc in range(1000):
    for scale in range(1000):
        mean = loc + scale / 2
        # in case met first time
        current_proba = expected_mean_dict.get(mean, 0)
        expected_mean_dict[mean] = current_proba + real_probas[loc, scale]
```



Expectation of expectation = ***75.6638***

Discrete uniform distribution bayesian multi-armed bandit (2 samples)

```
sample2 = bandit.trigger(0) + cost
print("Returned value:", sample2)

for loc in tqdm(range(1000)):
    for scale in range(1000):
        # not wasting time on 0
        if probabilities[loc, scale]:
            probabilities[loc, scale] *= stats.randint.pmf(sample2, loc, loc + 1 + scale)
```

```
Returned value: 4
```


```
max_sample = max(sample, sample2)
probabilities[max_sample:, :].sum()
```

```
0.0
```

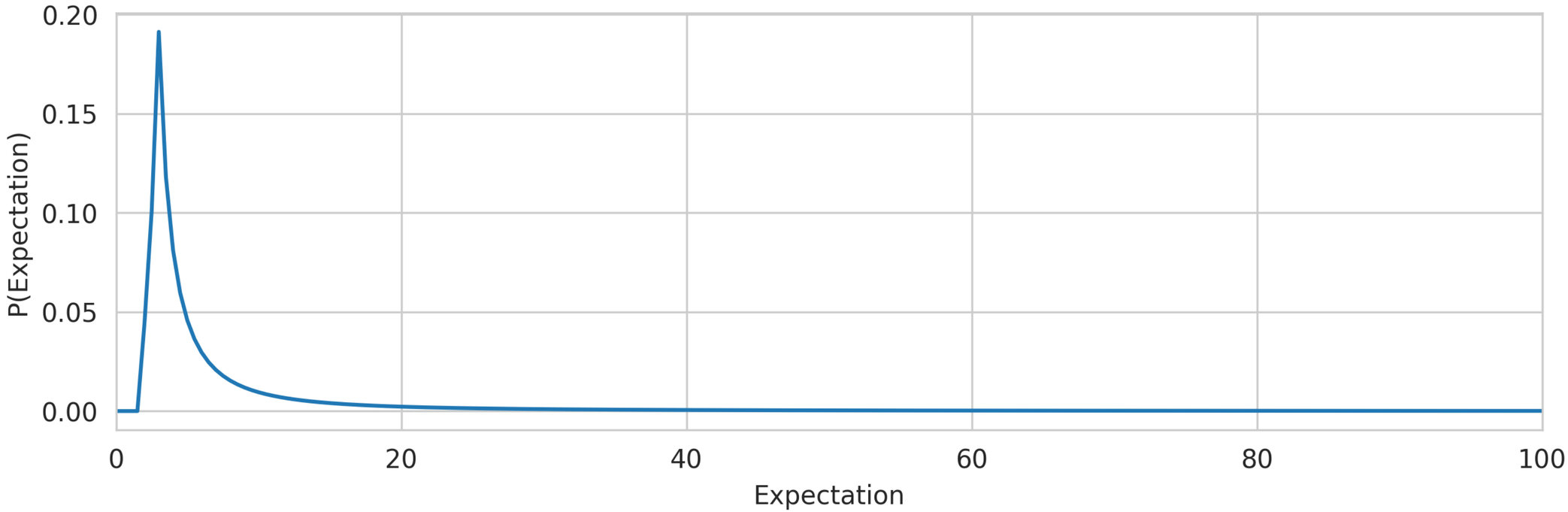
```
dist = abs(sample - sample2)
probabilities[:, :dist].sum()
```

```
0.0
```

```
probabilities.astype('bool').sum()
```

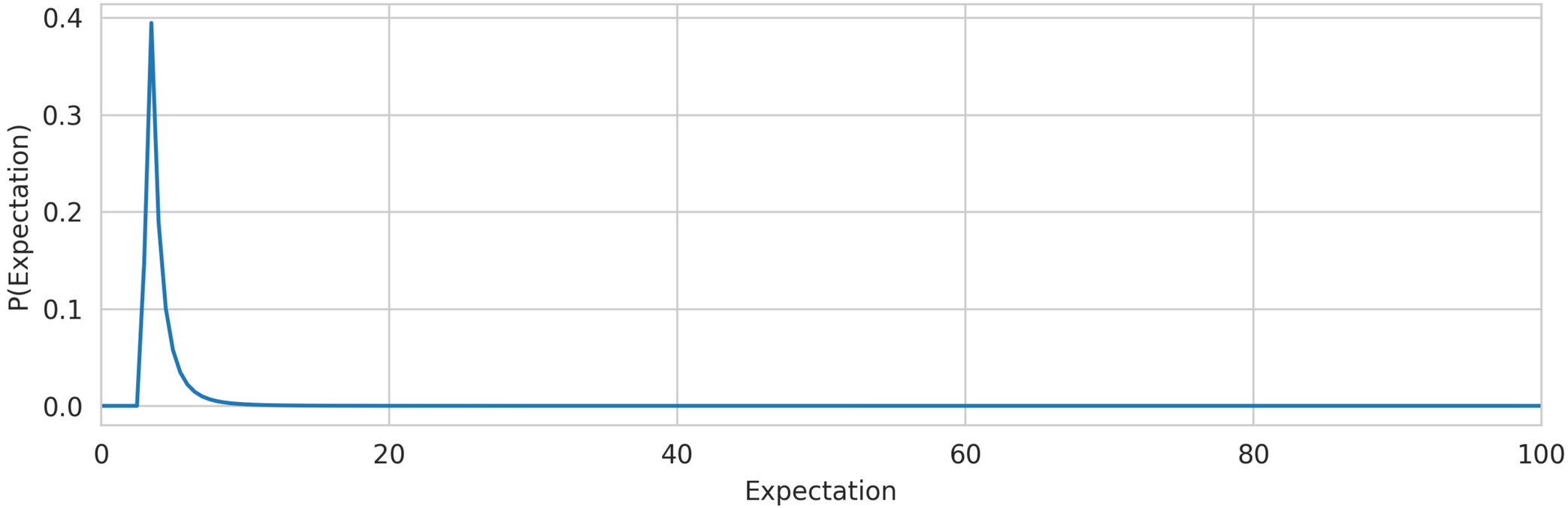
```
2991
```

Discrete uniform distribution bayesian multi-armed bandit (2 samples)



Expectation of expectation = ***10.1909***

Discrete uniform distribution bayesian multi-armed bandit (5 samples)



Expectation of expectation = ***4.0904***

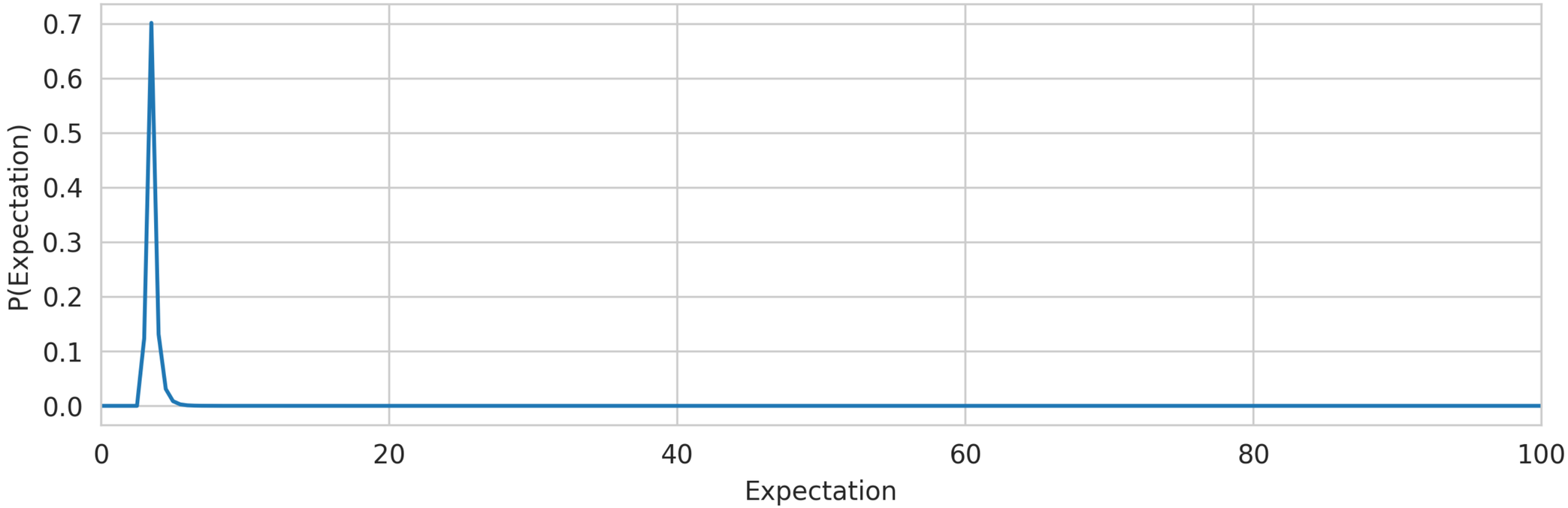
Adding prior information to our model

Does casino owner really want to give us lots of money?

Let us assume that proba for **loc** is reverse of its value
(*the greater **loc**, the lesser probability*)

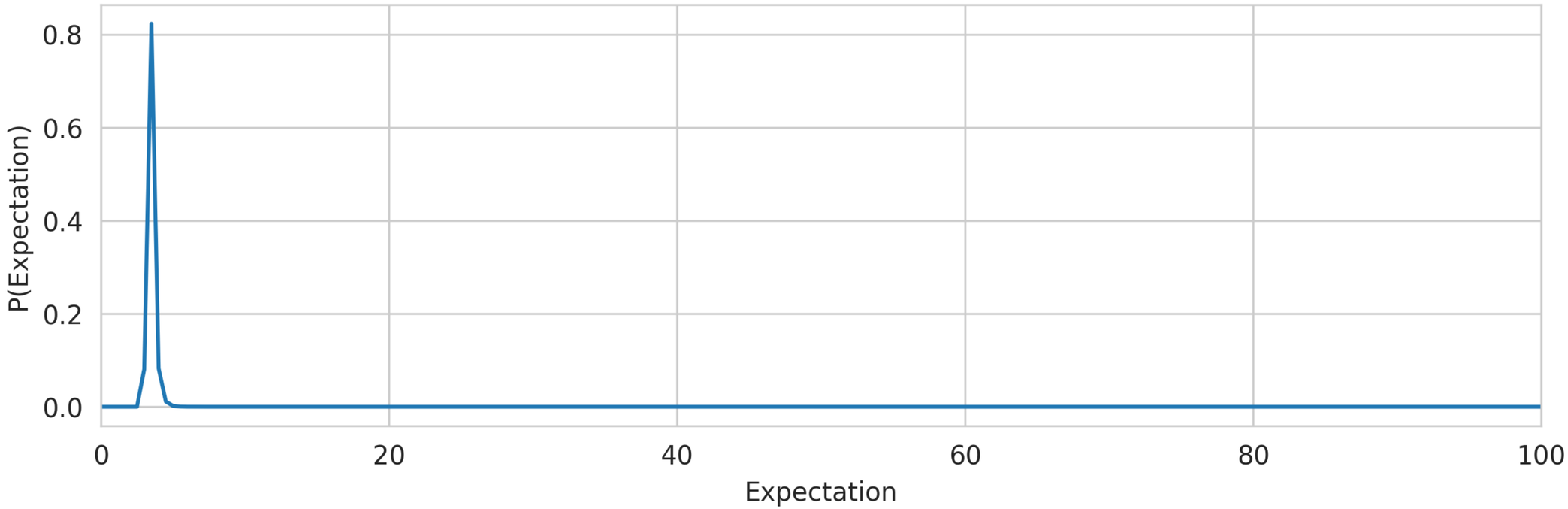
```
prior_probabilities = np.zeros((1000, 1000))
for i in range(1000):
    prior_probabilities[i, :] = 1 / (i+1)
```

Adding prior information to our model (1 sample)



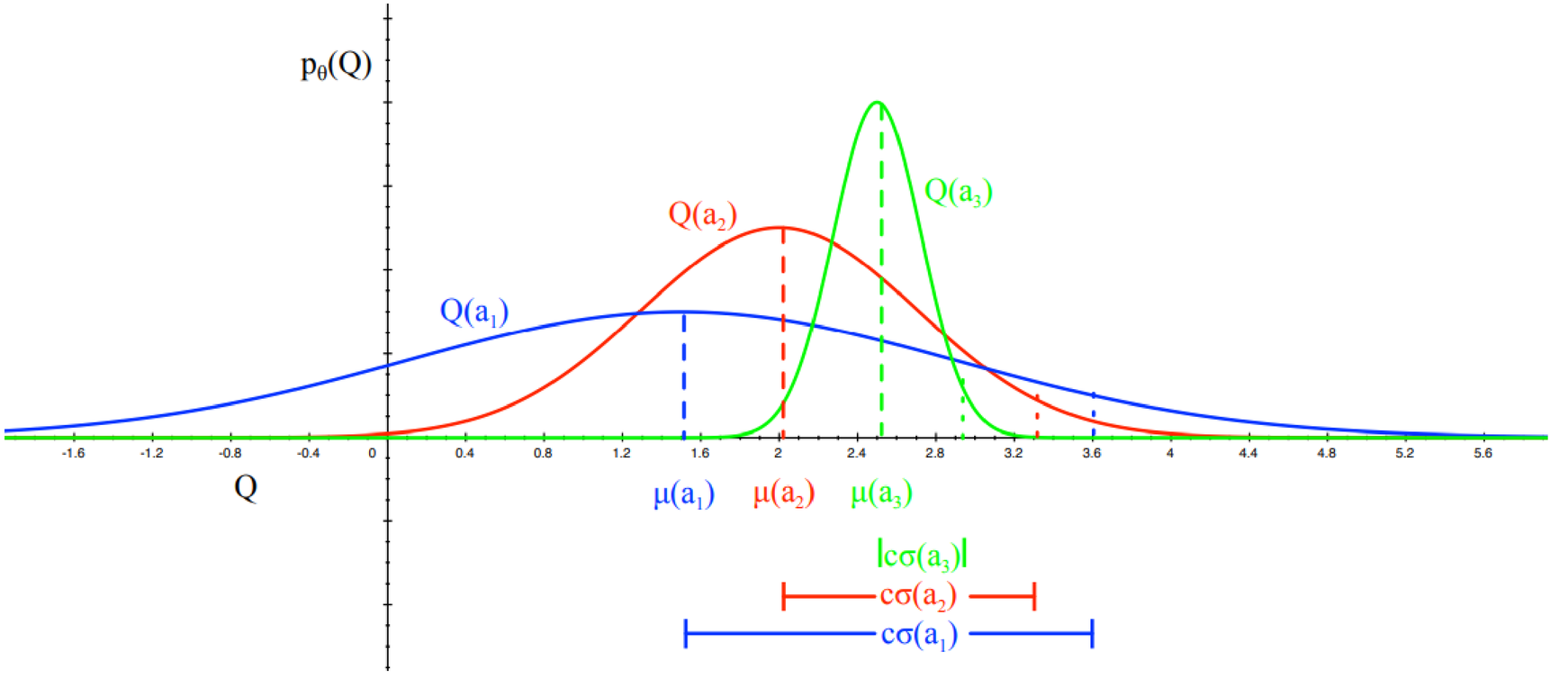
Expectation of expectation = ***3.5581***

Adding prior information to our model (5 samples)



Expectation of expectation = ***3.5163***

Multi-arm with Bayesian



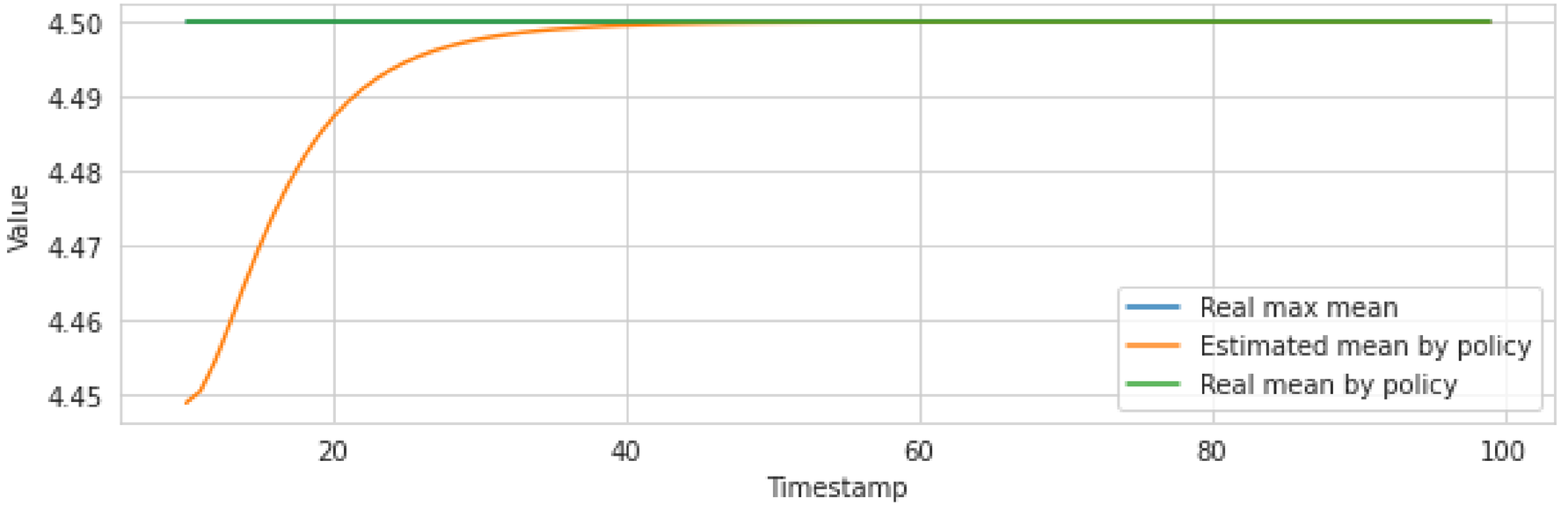
- We can estimate upper confidences from the posterior
  - e.g., $U_t(a) = c\sigma_t(a)$ where $\sigma(a)$ is std dev of $p_t(q(a))$
- Then, pick an action that maximises $Q_t(a) + c\sigma(a)$

Multi-arm with Bayesian

```python
def bayesian_policy(probas, n_arms = 4, c = 1):
    upper_bounds = []
    mean_expectations = []
    # computing mean expectation and std
    for i in range(n_arms):
        # computing proba for each mean
        real_probas = probas[i] / probas[i].sum()

        # mean expectation
        mean_expectation = (real_probas * mean_matrix).sum()
        # expectation of square - square of expectation
        variance = mean_matrix**2 * real_probas - mean_expectation**2
        std = variance ** 0.5

        mean_expectations.append(mean_expectation)
        upper_bounds.append(mean_expectation + c * std)

    return np.argmax(upper_bounds), mean_expectations[np.argmax(upper_bounds)]
```

Multi-arm with Bayesian

```python
def update_prediction(probas, arm_id, sample):
    for loc in range(1000):
        for scale in range(1000):
            # not wasting time on 0
            if probas[arm_id, loc, scale]:
                probas[arm_id, loc, scale] *= stats.randint.pmf(sample, loc, loc + 1 + scale)
```



Additional complexities

- Too great parameter space (several degree greater)
- Non-discrete parameter space (i.e. Poisson)
- Unknown distribution
- Mix of distributions
- Reward changes over time
- ...