4165446 Leander Zimmermann
6009977 Julian Zimmerlin

PML sheet 7

Theory, Part 2

a) $p(x_{t+1}=1 \mid x_t=0) = (1-\phi) \cdot \min\{1, \frac{1-p}{p}\}$

b) $p(x_{t+1}=0 \mid x_t=1) = (1-\phi) \cdot \min\{1, \frac{p}{1-p}\}$

c) $T = \begin{bmatrix} 1-p(x_{t+1}=1 \mid x_t=0) & p(x_{t+1}=0 \mid x_t=1) \\ p(x_{t+1}=1 \mid x_t=0) & 1-p(x_{t+1}=0 \mid x_t=1) \end{bmatrix}$

d) $T = \begin{bmatrix} 0.9 & 0.0\overline{6} \\ 0.1 & 0.9\overline{3} \end{bmatrix}$

$T^2 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.8167 \\ 0.1833 \end{bmatrix}; T^5 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.6412 \\ 0.3588 \end{bmatrix}$

$T^{10} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.4970 \\ 0.5030 \end{bmatrix}; T^{20} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.4158 \\ 0.5842 \end{bmatrix}; T^{100} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.4001 \\ 0.5999 \end{bmatrix}$

We can see that as the number of steps grows, the state distribution converges to that of the unfair coin toss, so we can use MCMC sampling to sample from an unfair coin toss using only samples from Uniform([0,1]).

Theory, Part 1

(a) Fair: $q(x) = \begin{cases} 1/2, & \text{if } x=0 \\ 1/2, & \text{if } x=1 \end{cases}$

(b) Unfair: $p(x) = \begin{cases} p, & \text{if } x=0 \\ 1-p, & \text{if } x=1 \end{cases}$

(c) $c q(x) \overset{!}{\geq} p(x) \; \forall x \Leftrightarrow \frac{c}{2} > p \wedge \frac{c}{2} > 1-p$

$\Leftrightarrow c > 2p \wedge c > 2(1-p) \Leftrightarrow c > 2\max\{p, 1-p\}$

Assuming $p < 0.5$: $\Leftrightarrow c > 2(1-p)$

(d) Define $m = \max\{p, 1-p\}$. Then $c = 2m$.

Rejection sampling: Sample from $q(x)$, then $u \sim \text{Uniform}[0, c\,q(x)]$, reject if $u > p(x)$.

$\hookrightarrow$ Rej. rate $= \sum_{x \in \{0,1\}} q(x) \frac{c\,q(x) - p(x)}{c\,q(x)}$

$= \frac{1}{2} \cdot \frac{m-p}{m} + \frac{1}{2} \cdot \frac{m-(1-p)}{m}$

$= \frac{m - p + m - 1 + p}{2m} = \frac{2m-1}{2m} = 1 - \frac{1}{2m}$

$\rightarrow$ For $p = 0.5$: Rej rate $= 1 - \frac{1}{2 \cdot \frac{1}{2}} = 0$

$\rightarrow$ For $p = 1$: Rej rate $= 1 - \frac{1}{2 \cdot 1} = \frac{1}{2}$

Probabilistic Machine Learning

Machine Learning in Science, University of Tübingen, Summer Semester 2022

Leander Zimmermann 4165446

Julian Zimmerlin 6009977

Exercise 07

hand in before 17.06.2022, 12:00 p.m. (noon)

In the lecture you learned about different methods that allow sampling from complex distributions. In this programming exercise, you will explore importance sampling in order to refine laplace approximations and implement a Gibbs-sampler for an Ising model.

Outline

1) Refining Laplace approximations with importance sampling.

2) Gibbs-sampling for Ising model.

Refining Laplace approximations with importance sampling

In [1]:

```
import torch
import matplotlib.pyplot as plt
from torch.autograd.functional import hessian
from torch.distributions import Normal, Uniform, Bernoulli
from copy import deepcopy
%matplotlib inline
```

In the first part of the exercise, we will refine a laplace approximation with importance sampling. Our goal will be to estimate the posterior mean $\mathbb{E}[p(y|x)]$ with Monte-Carlo sampling. We will show that the laplace approximation can fail to estimate this mean accurately, and that importance sampling can improve the estimate. First, we define a simple model $p(y|x)$ and a prior $p(x)$. The likelihood is $p(y|x) = \mathcal{N}(\mu = 30x + \frac{1}{x}, \sigma = 0.5)$. The prior is Uniform $p(x) = U(0.01, 1.0)$.

In [2]:

```
noise_dist = Normal(0, 0.5)

def log_likelihood(x):
    simulation = 30*x + 1/(x)
    difference = simulation - y_o
    return noise_dist.log_prob(difference)

prior = Uniform(0.01, 1.0)
```

Let's visualize this model by evaluating the probabilities on a grid:

In [3]:

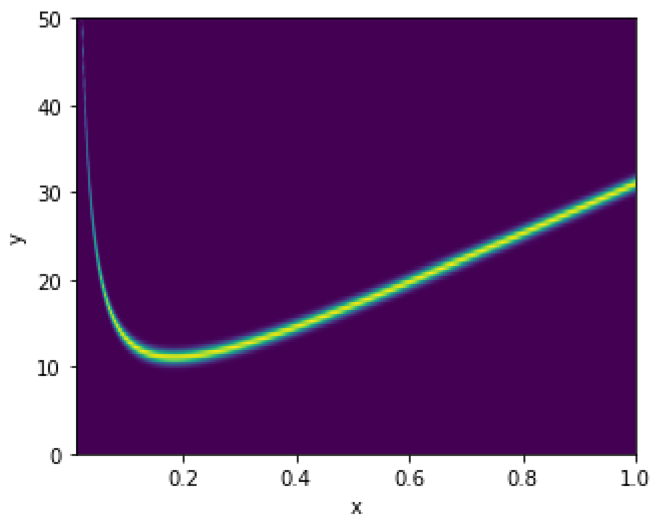
```
x_grid = torch.linspace(0.01, 1, 1000)
y_grid = torch.linspace(0, 50, 100)

all_likelihoods = []
for y_ in y_grid:
    y_o = y_
    likelihoods = torch.exp(log_likelihood(x_grid))
    all_likelihoods.append(likelihoods)
all_likelihoods = torch.stack(all_likelihoods)

fig, ax = plt.subplots(1, 1, figsize=(5, 4))
_ = ax.set_xlabel("x")
_ = ax.set_ylabel("y")
plt.imshow(all_likelihoods.numpy(), origin="lower", extent=[0.01, 1, 0, 50], aspect="auto")
```

Out[3]:

```
<matplotlib.image.AxesImage at 0x1addc742eb0>
```



Assume the observed data is $y_o = 11.2$.

In [4]:

```
y_o = 11.2
```

Then, given likelihood and prior, we can compute the unnormalized posterior $p(x|y_o) \propto p(y_o|x)p(x)$. Implement such a function:

In [5]:

```
def unnormalized_log_posterior(x):
    return log_likelihood(x) + torch.log(torch.tensor(1/(1-0.01)))
# If implemented correctly, this should return:
# unnormalized_log_posterior(torch.tensor([0.5])) -> tensor([-67.4958])
# unnormalized_log_posterior(torch.tensor([0.6, 0.8])) -> tensor([-143.5846, -395.0208])
```

In [6]:

```
unnormalized_log_posterior(torch.tensor([0.6, 0.8]))
```

Out[6]:

```
tensor([-143.5846, -395.0208])
```

In [7]:

```
def unnormalized_posterior(x):
    return torch.exp(unnormalized_log_posterior(x))
```

Establishing a ground truth

In this simple model, we can even compute the normalization constant by numerical integration (on a grid). We will use this to obtain a ground truth for the posterior later on. Note that, in more challenging real-world scenarios, this will not usually be possible. There are no tasks for you to do in this section, but please make sure that you understand the code.

In [8]:

```
dx = 0.99 / 999
l = unnormalized_posterior(x_grid)
Z_true = torch.sum(l) * dx
```

After having estimated the normalization constant, we can write a function that allows us to evaluate the posterior:

In [9]:

```
def posterior(x):
    return unnormalized_posterior(x) / Z_true
```

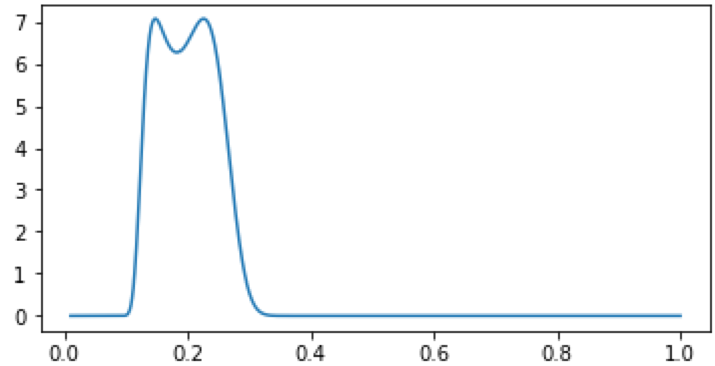
Let's visualize this:

In [10]:

```
x_grid = torch.linspace(0.01, 1, 1000)
posterior_values_on_grid = posterior(x_grid)
```

In [11]:

```
fig, ax = plt.subplots(1, 1, figsize=(6,3))
_ = ax.plot(x_grid, posterior_values_on_grid.detach().numpy())
```



In order to get a ground-truth value for the posterior mean, we compute the expected value of this posterior:

In [12]:

```
func_val = x_grid * posterior(x_grid)
mean_true = torch.sum(func_val) * dx
print("Mean when integrating on grid: ", mean_true.detach().item())
```

```
Mean when integrating on grid:  0.19864849746227264
```

Laplace approximation

In almost any real-world scenario, we can not compute the normalization constant of the posterior and we, therefore, can not evaluate the posterior mean as done above. Instead, we have to rely on other methods to approximate the posterior or generate (approximate) samples from it.

We will now explore a laplace-approximation to the posterior. Write a function to obtain the Maximum-likelihood-estimate $x_{\mathrm{MLE}} = \min_x - \log p(x|y)$ with gradient descent. As starting point, use $x = 0.99$.

In [13]:

```
x_mle = torch.nn.parameter.Parameter(torch.tensor(0.99))
optim = torch.optim.SGD([x_mle], lr=1e-4)

for _ in range(500):
    #print(x_mle)
    optim.zero_grad()
    loss = -log_likelihood(x_mle)
    loss.backward()
    optim.step()

# If implemented correctly, the x_mle should be
# Parameter containing:
# tensor([0.2270], requires_grad=True)
loss = -log_likelihood(x_mle)
print(f'Our MLE: {x_mle:.4f} has NLL {loss:.4f}')
loss_suggested = float(-log_likelihood(torch.tensor([0.2270])))
print(f'And the loss of the suggested value 0.2270: {loss_suggested:.4f} -> it is higher so
```

```
Our MLE: 0.2255 has NLL 0.2258
And the loss of the suggested value 0.2270: 0.2263 -> it is higher so optimi
zation routine is correct
```

Compute the hessian of the unnormalized log-posterior around the maximum likelihood estimate. Use the pytorch method `hessian` .

In [14]:

```
H = hessian(unnormalized_log_posterior, x_mle)
print(H)
# If implemented correctly, the hessian should be:
# tensor([[-458.7795]])
```

```
tensor(-427.7756)
```

...and construct the laplace approximation $\mathcal{N}(\mu = x_{\mathrm{MLE}}, \sigma = \sqrt{-H^{-1}})$:

In [15]:

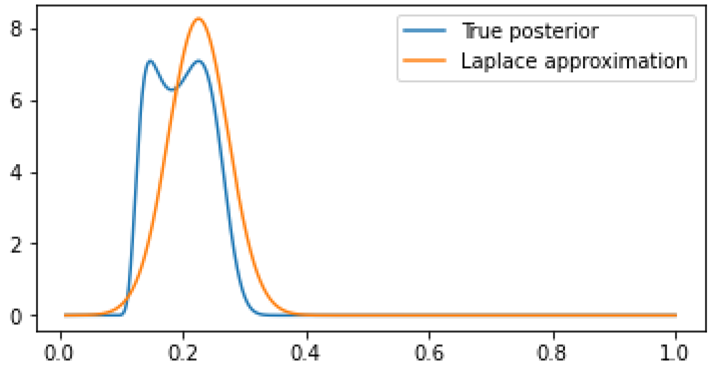
```
laplace_approximation = torch.distributions.Normal(x_mle.detach(), torch.sqrt(-1/H).detach(
```

We can again visualize this distribution and compare it to the (in this simple case available) ground truth posterior:

In [16]:

```
laplace_approximation_values_on_grid = torch.exp(laplace_approximation.log_prob(x_grid)).sq

fig, ax = plt.subplots(1, 1, figsize=(6,3))
_ = ax.plot(x_grid, posterior_values_on_grid.detach().numpy())
_ = ax.plot(x_grid, laplace_approximation_values_on_grid.detach().numpy())
_ = ax.legend(["True posterior", "Laplace approximation"])
```



From the above plot, we can already guess that the mean of the laplace approximation will be higher than the true posterior mean. Since the Laplace approximation is a gaussian, it's mean is the maximum-likelihood estimate:

In [17]:

```
laplace_mean = laplace_approximation.mean
print("Mean as estimated by laplace: ", laplace_mean.detach().item())
```

```
Mean as estimated by laplace:  0.22553983330726624
```

Refine with importance sampling

The laplace approximation overestimated the posterior mean. Let's see if we can fix this with importance sampling. We will use the Laplace approximation as proposal distribution $q(x)$. We then compute:

$$\mathbb{E}[p(x|y)] = \int p(x|y)\ x\ dx = \int \frac{p(x|y)}{q(x)}\ x\ q(x)\ dx \approx \frac{1}{N} \sum_i \frac{p(x_i|y)}{q(x_i)} x_i\,.$$

However, in the current scenario, the we can only estimate the posterior distribution **up to proportionality**. Therefore, we first have to estimate the normalization constant Z of the unnormalized posterior $\tilde{p}(x|y)$. We again do so with importance sampling:

$Z = \int \tilde{p}(x|y) dx = \int \frac{\tilde{p}(x|y)}{q(x)} q(x) dx \approx \frac{1}{N} \sum_i \frac{\tilde{p}(x_i|y)}{q(x_i)}$. Let's first estimate Z:

In [18]:

```
N = 10000000
laplace_samples = laplace_approximation.sample((N,))
laplace_log_probs = laplace_approximation.log_prob(laplace_samples)
laplace_probs = torch.exp(laplace_log_probs)

unnormalized_posterior_probs = unnormalized_posterior(laplace_samples)
Z = 1/N * (unnormalized_posterior_probs / laplace_probs).sum() # Estimate Z with importance

# If implemented correctly, Z should be approximately: tensor(0.1154)
print(Z)
```

```
tensor(0.1139)
```

Then we can estimate the posterior mean with importance sampling:

In [19]:

```
importance_sampling_mean = 1/Z/N * (laplace_samples * \
    unnormalized_posterior_probs / laplace_probs).sum() # Estimate the mean with importance
```

In [20]:

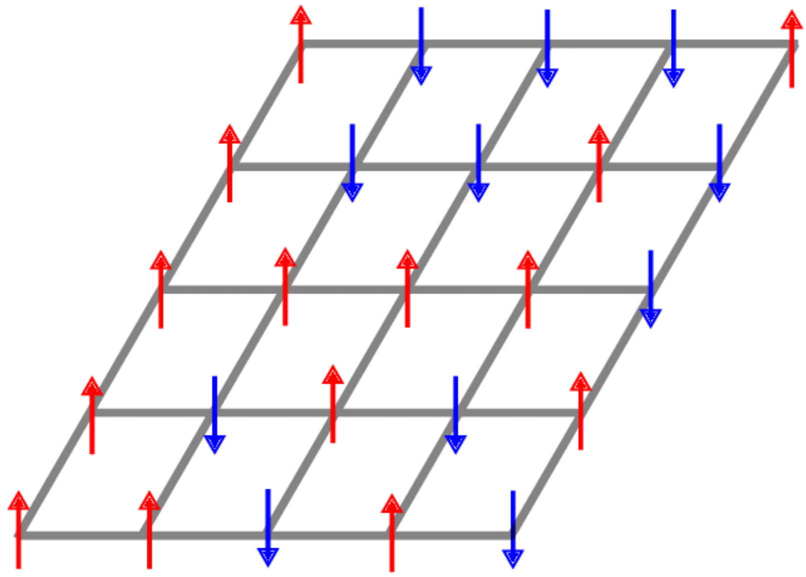
```
print("mean as estimated by laplace + importance sampling: ", importance_sampling_mean.deta
```

```
mean as estimated by laplace + importance sampling:  0.1986648291349411
```

This is a very good estimate of the true posterior mean. Congratualtions, you are done with the first part of the programming exercise!

Gibbs sampling for the Ising model

The Ising model is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables that represent magnetic dipole moments of atomic "spins" that can be in one of two states (+1 or −1). The spins are arranged in a lattice, allowing each spin to interact with its neighbors. Neighboring spins that agree have a lower energy than those that disagree; the system tends to the lowest energy but heat disturbs this tendency, thus creating the possibility of different structural phases. The image illustrates an ising model on a 2D-lattice:



[Image source (https://www.researchgate.net/figure/Schematic-representation-of-a-configuration-of-the-2D-Ising-model-on-a-square-lattice_fig2_321920877)](https://www.researchgate.net/figure/Schematic-representation-of-a-configuration-of-the-2D-Ising-model-on-a-square-lattice_fig2_321920877)

The probability of a particular state σ (i.e., a set of spins) in this system is given by:

$p(\sigma) = \frac{\exp(-\beta H(\sigma))}{Z}$.

The normlization constant Z is computationally expensive to evaluate. $H(\sigma)$ is the hamiltonian defined as:

$H(\sigma) = -\sum_{\langle i,j\rangle} J_{i,j}\sigma_i\sigma_j - \mu\sum_j h_j\sigma_j$

The notation $\langle i,j\rangle$ indicates that i, j are neighbours. J models the interactions between neighbours. We will assume that the magnetic moment μ is zero and that J is one:

$H(\sigma) = -\sum_{\langle i,j\rangle} \sigma_i\sigma_j$

This model (in a circular version, so last spin's right neighbour is the first spin) and a 1D-lattice is implemented below:

In [21]:

```
num_dim = 10

J = 1. # energy due to local field from nearby spins --> corresponds to diffusion term in H
h = 0. # energy due to external field --> corresponds to drift term in Hamiltonian

def hamiltonian(sigma):
#     Hamiltonian = H = -J * (Sum over pairs of nearest neighbour spins) - h (sum over spin
#     Assume 1D lattice, so nearest neighbours are on the right and left of each spin.
#     Assume that lattice is circular, so last spin's right neighbour is the first spin.
    assert sigma.dim() == 1
    nn_left = torch.roll(sigma, -1) # roll lattice to get all left neighbours
    nn_right = torch.roll(sigma, 1) # roll lattice to get all right neighbours
    term1 = nn_left.dot(sigma)
    term2 = nn_right.dot(sigma)

    term3 = sigma.sum()
    return -J * (term1 + term2) - h * term3
```

In [22]:

```
def ising_prob(sigma, beta):
    return torch.exp(-beta * hamiltonian(sigma))
```

In this tutorial, we will investigate how one can sample states σ given the unnormalized density function $\tilde{p}(\sigma) \propto \exp(-\beta H(\sigma))$. We will do so with Gibbs-sampling.

Each value of the initial state is sampled from a discrete distribution that has $p(1) = 0.5$ and $p(-1) = 0.5$.

In [23]:

```
initial = torch.as_tensor(torch.randint(2, (num_dim,)) * 2 - 1, dtype=torch.float32)
```

We then need a function that evaluates the (unnormalized) conditional distribution $\tilde{p}(\sigma_i | \sigma_{j \neq i})$. Write a function that evaluates $\tilde{p}(\sigma_i = 1 | \sigma_{j \neq i})$ and $\tilde{p}(\sigma_i = -1 | \sigma_{j \neq i})$:

In [24]:

```
# Note: I assume by "unnormalized conditional distribution" they mean
# the unnormalized joint p^~(sigma_i, sigma_{j!=i}). This is confusing
# but otherwise it doesn't make sense because you cant compute the UNNORMALIZED
# p^~(sigma_i | sigma_{j!=i}) easily and also it makes more sense
# with the next cell. Also it matches with the given results.

def conditional_dist_at_plus_1(sigma, dim, beta):
    sigma_plus1 = sigma.clone()
    sigma_plus1[dim] = 1
    return ising_prob(sigma_plus1, beta)# / (ising_prob(sig_plus1, beta) + ising_prob(sig_m

def conditional_dist_at_minus_1(sigma, dim, beta):
    sigma_minus1 = sigma.clone()
    sigma_minus1[dim] = -1
    return ising_prob(sigma_minus1, beta)

# If implemented correctly, the results should be approximately:
# conditional_dist_at_plus_1(torch.ones(10), 0, 0.1) -> tensor(7.3891)
# conditional_dist_at_minus_1(torch.ones(10), 0, 0.1) -> tensor(3.3201)
print(conditional_dist_at_plus_1(torch.ones(10), 0, 0.1))
print(conditional_dist_at_minus_1(torch.ones(10), 0, 0.1))
```

```
tensor(7.3891)
tensor(3.3201)
```

Given $\tilde{p}(\sigma_i = 1 | \sigma_{j \neq i})$ and $\tilde{p}(\sigma_i = -1 | \sigma_{j \neq i})$, we then write a function that samples the (normalized!) conditional distribution $\tilde{p}(\sigma_i | \sigma_{j \neq i})$.

In [25]:

```
def sample_conditional(sigma, dim, beta):
    p_plus1 = conditional_dist_at_plus_1(sigma, dim, beta) / \
    (conditional_dist_at_plus_1(sigma, dim, beta) + conditional_dist_at_minus_1(sigma, dim,
    conditional_samples = torch.bernoulli(p_plus1)*2-1
    return conditional_samples # Should have shape (1,)

# E.g. sample_conditional(torch.ones(10), 1, 0.5) -> tensor([1.])
sample_conditional(torch.ones(10), 1, 0.5)
```

Out[25]:

```
tensor(1.)
```

Finally, write a function that loops over every dimension and samples each conditional once:

In [26]:

```
def gibbs(sigma, beta):
    result = torch.zeros_like(sigma)
    for dim in range(len(sigma)):
        result[dim] = sample_conditional(sigma, dim, beta)
    return result
```

Done! We have implemented the Gibbs-sampler for the Ising model. We can now simulate the Ising model for different values of β. We will start with low temperatures, i.e., high values of β.

In [27]:

```
beta = 0.5
```

Run the gibbs sampler for N steps. Visualize the states as an $[N, D]$ matrix, with D being the number of states of the system:

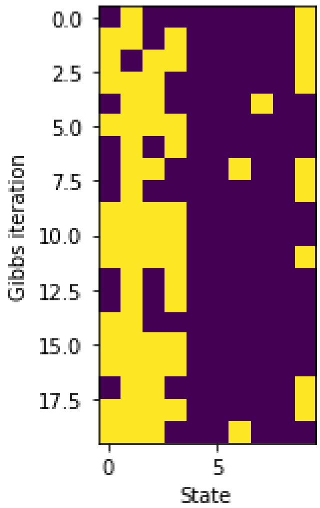
In [28]:

```
all_samples = []
for _ in range(20):
    s = gibbs(initial, beta=beta)
    all_samples.append(deepcopy(s))
all_samples = torch.stack(all_samples)

fig, ax = plt.subplots(1, 1, figsize=(6, 4))
_ = ax.imshow(all_samples.numpy())
ax.set_xlabel("State")
ax.set_ylabel("Gibbs iteration")
```

Out[28]:

```
Text(0, 0.5, 'Gibbs iteration')
```



We can check whether samples returned by the gibbs sampler indeed produce states that have high probability. Compute the average probability of states sampled by the Gibbs sampler and compare them to sampling states independently and uniformly.

In [29]:

```
average_prob_gibbs = torch.mean(torch.stack([ising_prob(s, beta=beta) for s in all_samples]
print("Average (unnormalized) probability of gibbs samples:", average_prob_gibbs)
```

```
Average (unnormalized) probability of gibbs samples: tensor(185.2443)
```

In [30]:

```
random_samples = torch.as_tensor(torch.randint(2, (50, num_dim)) * 2 - 1, dtype=torch.float
average_prob_random = torch.mean(torch.stack([ising_prob(s, beta=beta) for s in random_samp
print("Average (unnormalized) probability of random samples:", average_prob_random)
```

```
Average (unnormalized) probability of random samples: tensor(52.0070)
```

Next, we will run the sampler for high temperatures:

In [31]:

```
beta = 0.001
```

In [32]:

```
all_samples = []
for _ in range(20):
    s = gibbs(initial, beta=beta)
    all_samples.append(deepcopy(s))
all_samples = torch.stack(all_samples)

fig, ax = plt.subplots(1, 1, figsize=(6, 4))
_ = ax.imshow(all_samples.numpy())
_ = ax.set_xlabel("State")
_ = ax.set_ylabel("Gibbs iteration")
```