CS179 Homework 6

Due: Friday, June 6 2025 (11:59 PM)

Instructions

This homework contains quite a lot of starter code and templates, so it is being issued in past years' format: as a jupyter notebook that you can complete and convert to PDF for upload to GradeScope. Unlike previous homeworks, this will require that you assign page numbers to each problem in the assignment.

Please submit a **single PDF file** that contains everything to Gradescope. This includes any text you wish to include to describe your results, the complete code snippets of how you attempted each problem, any figures that were generated, and scans of any work on paper that you wish to include. It is important that you include enough detail that we know how you solved the problem, since otherwise we will be unable to grade it.

You are encouraged to use this starter Jupyter notebooks to complete your assignment and to write your report. This will help you not only ensure that all of the code for the solutions is included, but also will provide an easy way to export your results to a PDF file (for example, doing *print preview* and *printing to pdf*). I recommend liberal use of Markdown cells to create headers for each problem and sub-problem, explaining your implementation/answers, and including any mathematical equations. For parts of the homework you do on paper, scan it in such that it is legible (there are a number of free Android/iOS scanning apps, if you do not have access to a scanner), and include it as an image in the Jupyter notebook.

Double check that all of your answers are legible on Gradescope, e.g. make sure any figures or text you have written does not get cut off.

If you have any questions/concerns about using Jupyter notebooks, ask us on EdD. If you decide not to use Jupyter notebooks, but go with Microsoft Word or LaTeX to create your PDF file, make sure that all of the answers can be generated from the code snippets included in the document.

Summary of Assignment: 100 total points

- Part 0: Game Support Functions
- Part 1: Model Set-Up (20 points)
- Part 2: Policy Gradient (20 points)
- Part 3: Analysis (25 points)
- Part 4: Adaptation (30 points)
- Statement of Collaboration (5 points)

```
!pip install pyGMs pyro-ppl
import matplotlib.pyplot as plt

import pyGMs as gm
import numpy as np
import torch
import random

import requests                                 # reading data
from io import StringIO
import time

import pyro
import pyro.infer
import pyro.optim
import pyro.distributions as dist
import torch.distributions.constraints as constraints
import pyro.poutine as poutine

from IPython.display import display, clear_output  # for iterative plotting

seed = 123
random.seed(seed)
pyro.set_rng_seed(seed)
```

```
Collecting pyGMs
  Downloading pygms-0.3.0-py3-none-any.whl.metadata (1.4 kB)
Collecting pyro-ppl
  Downloading pyro_ppl-1.9.1-py3-none-any.whl.metadata (7.8 kB)
Requirement already satisfied: numpy>=1.7 in /usr/local/lib/python3.11/dist-packages (from pyro-ppl) (2.0.2)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.11/dist-packages (from pyro-ppl) (3.4.0)
Collecting pyro-api>=0.1.1 (from pyro-ppl)
  Downloading pyro_api-0.1.2-py3-none-any.whl.metadata (2.5 kB)
Requirement already satisfied: torch>=2.0 in /usr/local/lib/python3.11/dist-packages (from pyro-ppl) (2.6.0+cu124)
Requirement already satisfied: tqdm>=4.36 in /usr/local/lib/python3.11/dist-packages (from pyro-ppl) (4.67.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (3.18.0)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (4.14.0)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (2025.3.2)
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (12.4.127)
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch>=2.0->pyro-ppl)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0->pyro-ppl) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch>=2.0->pyro-ppl) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch>=2.0->pyro-ppl) (3.0.2)
Downloading pygms-0.3.0-py3-none-any.whl (133 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 133.7/133.7 kB 4.2 MB/s eta 0:00:00
Downloading pyro_ppl-1.9.1-py3-none-any.whl (755 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 756.0/756.0 kB 20.7 MB/s eta 0:00:00
Downloading pyro_api-0.1.2-py3-none-any.whl (11 kB)
Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl (363.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 3.4 MB/s eta 0:00:00
Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (13.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.8/13.8 MB 90.4 MB/s eta 0:00:00
```

```
Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (24.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.6/24.6 MB 81.3 MB/s eta 0:00:00
Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-manylinux2014_x86_64.whl (883 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 883.7/883.7 kB 47.7 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl (664.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 1.7 MB/s eta 0:00:00
Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl (211.5 MB)
```

In this homework, we will Pyro's optimization to perform policy gradient optimization for a simple Markov decision process representing a game of blackjack.

Our simplified game of blackjack works as follows:

- The player and dealer each draw a card (visible to both)
- The player can continue to draw cards ("hit"), eventually stopping ("hold")
- The player's score is the sum of their card values, with face cards counting as 10, and aces counting as either one or eleven (player's choice)
- If the player's score goes over 21, they "bust" (lose)
- After stopping, the dealer draws cards until either they have a higher score than the player (player loses) or the dealer busts (player wins().
- Winning is worth +1 dollar; losing worth -1 dollar

Set-up: Game support functions

First we define a few useful functions for summarizing the state of the game (from the player's point of view):

```
def val(cards):
    """Value of a hand of cards"""
    s = sum([min(c,10) for c in cards])             # face cards (J,Q,K) count as 10
    ace = 1*(s<12 and any([c==1 for c in cards]))   # usable ace: have an ace and making it 11 doesn't bust
    if ace: s += 10                                 # (if so, treat it as an 11)
    return s,ace

def state(player,dealer):  # player = list of cards; dealer = list with single card showing
    """State representation (triple of ints) summarizing visible state"""
    s,a = val(player)
    d,_ = val(dealer[:1])
    return int(max(min(s,22)-11,0)),int(d-2), a
    # State triple is:  (player score, dealer score showing, player-usable-ace)
    # player scores <= 11 are combined (should always hit), and >21 (already busted)
    #   and shifted to the range 0...11  (0=11 or less, 11=22 or more)

nstates = ((22-11)),10,2      # policy size (max # values of state triple during play)

# SOME MORE USEFUL FUNCTIONS

def score(cards):        # The score of a given hand (sum of card values)
    s,ace = val(cards)
    return s

def bust(cards):         # bust if score > 21
    return score(cards)>21

def hand(cards):         # string representation of a hand
    names = ['Ace','2','3','4','5','6','7','8','9','10','J','Q','K']
    st = ','.join([names[c-1] for c in cards])
    st += ' (bust)' if bust(cards) else f' ({score(cards)})'
    return st
```

Part 1: Defining the model

We will define the Markov process using Pyro, as a sequence of actions (selected by the policy) and state changes (card draws), followed by the payout (performed by sampling the dealer cards and comparing).

We represent the policy as logistic values, to avoid needing to represent the probability simplex constrants (probability of "hit" between 0 and 1); so: `P(hit | state) = expit(log-policy[state])`.

```
pyro.clear_param_store()
pyro.set_rng_seed(seed)

def model():
    player = []
    actions = []
    dealer = []
    policy = torch.special.expit( pyro.param( 'log-policy', torch.rand(nstates)-.5 ) )  # initial random per-state policy

    # Start with one card each to player and dealer
    player.append( pyro.sample(f'P0', dist.Categorical(torch.ones(13)/13))+1 )
    dealer.append( pyro.sample(f'D0', dist.Categorical(torch.ones(13)/13))+1 )

    # Player draws cards until they decide to stop (hold) or bust (>21)
    hold = False
    while not (hold or bust(player)):
        actions.append( pyro.sample(f'A{len(player)}', dist.Bernoulli(policy[state(player,dealer)])) )
        if actions[-1]:  # if hit:
            player.append( pyro.sample(f'P{len(player)}', dist.Categorical(torch.ones(13)/13))+1 )
        else:
            hold=True

    # now, dealer draws until they win or bust:
    while not (bust(player) or bust(dealer) or (score(dealer)>score(player))):
        dealer.append( pyro.sample(f'D{len(dealer)}', dist.Categorical(torch.ones(13)/13))+1 )

    payout = 1 if bust(dealer) else -1

    return payout,player,dealer   # need payout value; return others for convenience

########

_ = model()  # force init if not yet done

params = [pyro.param( 'log-policy')]   # only one parameter array in our model
```

(There is nothing to do here, except look over the code and make sure you understand what it's doing.)

Part 2: Training the policy

We would like to optimize the expected value, $\mathbb{E}_p[u(x)]$, over the parameters of the distribution p. We can estimate this value using a Monte Carlo estimate, by drawing samples of trajectories (gameplays). Unfortunately, in some models like this one, the sampled utility $u(x)$ is not a differentiable function of the parameters of the distribution p; here, for example, x is discrete.

To resolve this issue, we will use a "score function" estimator, obtained by rewriting

$$\nabla_\theta \int u(x)p(x;\theta) = \int u(x)\nabla_\theta p(x;\theta) = \int u(x)p(x;\theta)\nabla_\theta \log\, p(x;\theta) = \mathbb{E}_p[u(x)\nabla_\theta \log\, p(x;\theta)]$$

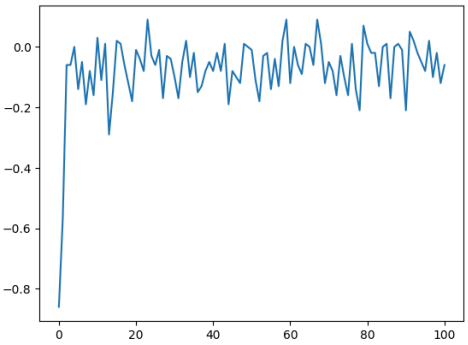
We then use our non-differentiable samples to estimate the expectation $\mathbb{E}_p[u(x)\log\, p(x;\theta)]$ and use auto-differentiation to optimize with respect to θ.

```
%%time
steps = 10000
batchsize = 10
optimizer = torch.optim.Adam(params, lr=1e-1)
payouts = []

for step in range(steps+1):
    optimizer.zero_grad()             # call this before computing the gradient

    # draw sample(s) from model
    model_traces = [poutine.trace(model).get_trace() for _ in range(batchsize)]
    # use "score function" loss: U(x) * log p(x;theta)
    loss = - sum([t.nodes['_RETURN']['value'][0] * t.log_prob_sum() for t in model_traces])
    loss.backward()                   # compute the gradient
    optimizer.step()                  # & update

    if step % 100 == 0:
        payouts.append( np.mean([model()[0] for _ in range(200)]) )
        clear_output(wait=True)
        plt.plot(payouts); plt.show()
```




```
CPU times: user 9min 5s, sys: 3.46 s, total: 9min 9s
Wall time: 9min 6s
```

Train your model. You may want to adjust the initial step size and number of gradient steps, and/or the number of games used per gradient update (batchsize, above), and see how well you can do. Note that there is some noise in the evaluation values as well.

Part 3: Analysis

- Display (as two array of probabilities, one for "ace=False" and the other for "ace=True") the optimized probability of taking another card under your policy.

```
log_policy = pyro.param('log-policy').detach()
policy_array = torch.special.expit(log_policy)

print("===== ROUNDED =====")
print()
print(f"ace=False: \n\n{np.round(policy_array[:, :, 0].detach().cpu().numpy(), 3)}\n")
print(f"ace=True: \n\n{np.round(policy_array[:, :, 1].detach().cpu().numpy(), 3)}\n")

print("===== RAW =====")
print()
print(f"ace=False: \n\n{np.round(policy_array[:, :, 0].detach().cpu().numpy(), 3)}\n")
print(f"ace=True: \n\n{np.round(policy_array[:, :, 1].detach().cpu().numpy(), 3)}\n")
```

```
===== ROUNDED =====

ace=False:

[[1.    1.    1.    1.    1.    1.    1.    1.    1.    1.   ]
 [0.999 0.999 1.    1.    1.    1.    1.    0.844 1.    0.998]
 [0.974 0.002 0.999 0.996 0.983 1.    1.    0.956 1.    0.   ]
 [0.989 1.    0.999 0.998 0.993 1.    1.    1.    1.    0.999]
 [0.997 0.    0.004 1.    0.002 0.004 0.997 0.935 1.    0.999]
 [0.    0.    0.001 0.    0.    0.03  0.993 0.996 0.993 0.972]
 [0.001 0.001 0.01  0.    0.003 0.001 0.    0.    0.    0.999]
 [0.    0.    0.    0.    0.    0.    0.    0.005 0.    0.001]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.001]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.   ]]

ace=True:

[[1.    1.    1.    1.    1.    1.    1.    1.    1.    0.999]
 [0.999 0.999 0.998 0.953 0.997 0.993 0.998 0.999 0.999 0.996]
 [0.974 0.999 0.999 0.998 0.99  1.    1.    0.999 0.999 0.97 ]
 [0.999 0.992 0.997 1.    0.003 0.007 1.    1.    0.999 0.012]
 [0.995 0.999 0.907 0.998 0.992 0.999 0.985 0.973 0.945 0.001]
 [0.02  0.001 1.    0.971 0.999 0.998 1.    0.938 0.995 0.981]
 [1.    0.999 0.03  0.999 0.002 0.977 0.999 0.991 1.    0.056]
 [0.99  0.998 0.    0.001 1.    0.005 0.012 0.996 0.    0.996]
 [0.    0.    0.    0.003 0.    0.    0.    0.    0.856 0.019]
 [0.    0.003 0.002 0.    0.    0.001 0.    0.001 0.001 0.009]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.   ]]
```

```
===== RAW =====

ace=False:

[[1.    1.    1.    1.    1.    1.    1.    1.    1.    1.   ]
 [0.999 0.999 1.    1.    1.    1.    1.    0.844 1.    0.998]
 [0.974 0.002 0.999 0.996 0.983 1.    1.    0.956 1.    0.   ]
 [0.989 1.    0.999 0.998 0.993 1.    1.    1.    1.    0.999]
 [0.997 0.    0.004 1.    0.002 0.004 0.997 0.935 1.    0.999]
 [0.    0.    0.001 0.    0.    0.03  0.993 0.996 0.993 0.972]
 [0.001 0.001 0.01  0.    0.003 0.001 0.    0.    0.    0.999]
 [0.    0.    0.    0.    0.    0.    0.    0.005 0.    0.001]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.001]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.    0.   ]]

ace=True:

[[1.    1.    1.    1.    1.    1.    1.    1.    1.    0.999]
 [0.999 0.999 0.998 0.953 0.997 0.993 0.998 0.999 0.999 0.996]
 [0.974 0.999 0.999 0.998 0.99  1.    1.    0.999 0.999 0.97 ]
 [0.999 0.992 0.997 1.    0.003 0.007 1.    1.    0.999 0.012]
 [0.995 0.999 0.907 0.998 0.992 0.999 0.985 0.973 0.945 0.001]
 [0.02  0.001 1.    0.971 0.999 0.998 1.    0.938 0.995 0.981]
 [1.    0.999 0.03  0.999 0.002 0.977 0.999 0.991 1.    0.056]
 [0.99  0.998 0.    0.001 1.    0.005 0.012 0.996 0.    0.996]
 [0.    0.    0.    0.003 0.    0.    0.    0.    0.856 0.019]
 [0.    0.003 0.002 0.    0.    0.001 0.    0.001 0.001 0.009]
```

- Sample and display five games following your policy. Comment on whether the player's actions appear reasonable.

```
for i in range(5):
    player = []
    dealer = []
    player.append(int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1)
    dealer.append(int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1)

    print(f"\nGame {i}")
    print(f"Initial Player: {player}, Dealer shows: {dealer[0]}")

    while not (bust(player)):
        pb, db, af = state(player, dealer)
        prob_hit = float(policy_array[pb, db, af].item())
        action = int(torch.bernoulli(torch.tensor(prob_hit)).item())
        action_str = "HIT" if action == 1 else "HOLD"
        print(f"Player hand: {player} (score {score(player)}), usable_ace: {af} => prob_hit: {prob_hit:.3f} → {action_str}")
        if action == 1:
            card = int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1
            player.append(card)
            if bust(player):
                print(f"Player draws {card} → bust with {player} (score {score(player)})")
                break
        else:
            break

    if not bust(player):
        print(f"Player holds at {player} (score {score(player)})")
        while not (bust(player) or bust(dealer) or (score(dealer) > score(player))):
            card = int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1
            dealer.append(card)
        if bust(dealer):
            print(f"Dealer draws {dealer[-1]} => bust with {dealer} (score {score(dealer)}) → Player WINS")
        else:
            print(f"Dealer final: {dealer} (score {score(dealer)}) => Dealer WINS, Player LOSES")
    else:
        print(f"Player busted => Player LOSES")
```

```
Game 0
Initial Player: [13], Dealer shows: 4
Player hand: [13] (score 10), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [13, 9] (score 19), usable_ace: 0 => prob_hit: 0.000 → HOLD
Player holds at [13, 9] (score 19)
Dealer draws 9 => bust with [4, 4, 9, 9] (score 26) → Player WINS

Game 1
Initial Player: [4], Dealer shows: 11
Player hand: [4] (score 4), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [4, 1] (score 15), usable_ace: 1 => prob_hit: 0.945 → HIT
Player hand: [4, 1, 13] (score 15), usable_ace: 0 => prob_hit: 1.000 → HIT
Player draws 13 → bust with [4, 1, 13, 13] (score 25)
Player busted => Player LOSES

Game 2
Initial Player: [2], Dealer shows: 9
Player hand: [2] (score 2), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [2, 10] (score 12), usable_ace: 0 => prob_hit: 0.844 → HIT
Player hand: [2, 10, 5] (score 17), usable_ace: 0 => prob_hit: 0.000 → HOLD
Player holds at [2, 10, 5] (score 17)
Dealer final: [9, 1] (score 20) => Dealer WINS, Player LOSES

Game 3
Initial Player: [2], Dealer shows: 9
Player hand: [2] (score 2), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [2, 3] (score 5), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [2, 3, 13] (score 15), usable_ace: 0 => prob_hit: 0.935 → HIT
Player hand: [2, 3, 13, 6] (score 21), usable_ace: 0 => prob_hit: 0.000 → HOLD
Player holds at [2, 3, 13, 6] (score 21)
Dealer draws 6 => bust with [9, 1, 7, 2, 6] (score 25) → Player WINS

Game 4
Initial Player: [2], Dealer shows: 10
Player hand: [2] (score 2), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [2, 3] (score 5), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [2, 3, 10] (score 15), usable_ace: 0 => prob_hit: 1.000 → HIT
Player hand: [2, 3, 10, 6] (score 21), usable_ace: 0 => prob_hit: 0.000 → HOLD
Player holds at [2, 3, 10, 6] (score 21)
Dealer draws 12 => bust with [10, 2, 2, 4, 2, 12] (score 30) → Player WINS
```

The behavior of the player does seem rather reasonable and consistently follows rather textbook strategies - it always hits when the overall score is relatively low, holds when the score is high, and plays soft aces aggresively. This can be seen in Game 1, where the player continues to play even with a score of 15 with the usable ace. However, overall, since the expectated payout over 1000s of games is negative, it may not be the best possible behavior.

- Estimate (using a few thousand samples) the expected payout of your optimized policy.

```
payouts = []

for _ in range(10000):
    player = [int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1]
    dealer = [int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1]
```

```
    while not bust(player):
        pb, db, af = state(player, dealer)
        prob_hit = float(policy_array[pb, db, af].item())
        if torch.bernoulli(torch.tensor(prob_hit)).item() == 1:
            player.append(int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1)
        else:
            break

    while not (bust(player) or bust(dealer) or score(dealer) > score(player)):
        dealer.append(int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1)

    payouts.append(1.0 if bust(dealer) else -1.0)

print(f"Estimated expected payout over {10000} games: {np.mean(payouts):.4f}")
```

```
Estimated expected payout over 10000 games: -0.0622
```

Double-click (or enter) to edit

Part 4: Adaptation

- Change the rules of the game so that face cards (J,Q,K) count as *more* than 10 (i.e., count as 11,12,13). You will need to adjust the state definition and number of states to compensate for this update.

```
def val(cards):
    s = sum([c if c != 1 else 1 for c in cards])
    has_ace = any(c == 1 for c in cards)
    if has_ace and (s + 10) <= 21:
        s += 10
        usable_ace = True
    else:
        usable_ace = False
    return s, usable_ace


def state(player, dealer):
    s, usable_ace = val(player)
    d, _ = val(dealer[:1])
    player_idx = int(max(min(s, 22) - 11, 0))
    dealer_idx = int(d - 2)
    ace_flag = 1 if usable_ace else 0

    return player_idx, dealer_idx, ace_flag

nstates = 12, 12, 2


pyro.clear_param_store()
pyro.set_rng_seed(seed)

def model():
    player = []
    actions = []
    dealer = []
    policy = torch.special.expit( pyro.param( 'log-policy', torch.rand(nstates)-.5 ) )  # initial random per-state policy

    # Start with one card each to player and dealer
    player.append( pyro.sample(f'P0', dist.Categorical(torch.ones(13)/13))+1 )
    dealer.append( pyro.sample(f'D0', dist.Categorical(torch.ones(13)/13))+1 )

    # Player draws cards until they decide to stop (hold) or bust (>21)
    hold = False
    while not (hold or bust(player)):
        actions.append( pyro.sample(f'A{len(player)}', dist.Bernoulli(policy[state(player,dealer)])) )
        if actions[-1]:  # if hit:
            player.append( pyro.sample(f'P{len(player)}', dist.Categorical(torch.ones(13)/13))+1 )
        else:
            hold=True

    # now, dealer draws until they win or bust:
    while not (bust(player) or bust(dealer) or (score(dealer)>score(player))):
        dealer.append( pyro.sample(f'D{len(dealer)}', dist.Categorical(torch.ones(13)/13))+1 )

    payout = 1 if bust(dealer) else -1

    return payout,player,dealer   # need payout value; return others for convenience

########

_ = model()  # force init if not yet done

params = [pyro.param( 'log-policy')]   # only one parameter array in our model
```

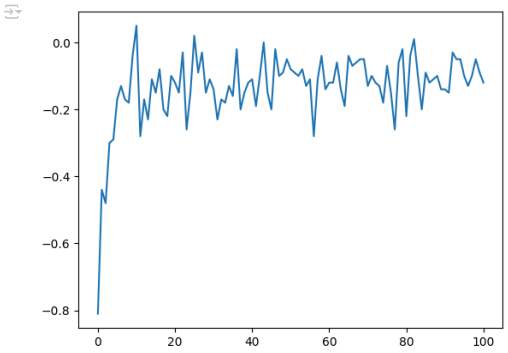
- Train a policy for your new game. Does the new game appear to be easier (for the player to win), harder or about the same as the original blackjack?

```
%%time
steps = 10000
batchsize = 10
optimizer = torch.optim.Adam(params, lr=1e-1)
payouts = []

for step in range(steps+1):
    optimizer.zero_grad()             # call this before computing the gradient

    # draw sample(s) from model
    model_traces = [poutine.trace(model).get_trace() for _ in range(batchsize)]
    # use "score function" loss: U(x) * log p(x;theta)
    loss = - sum([t.nodes['_RETURN']['value'][0] * t.log_prob_sum() for t in model_traces])
    loss.backward()                   # compute the gradient
    optimizer.step()                  # & update

    if step % 100 == 0:
        payouts.append( np.mean([model()[0] for _ in range(200)]) )
        clear_output(wait=True)
        plt.plot(payouts); plt.show()
```




```
CPU times: user 8min 23s, sys: 2.72 s, total: 8min 26s
Wall time: 8min 22s
```

```
payouts = []

for _ in range(10000):
    player = [int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1]
    dealer = [int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1]

    while not bust(player):
        pb, db, af = state(player, dealer)
        af = int(af)
        pb = min(max(pb, 0), policy_array.shape[0]-1)
        db = min(max(db, 0), policy_array.shape[1]-1)
        prob_hit = float(policy_array[pb, db, af])
        if torch.bernoulli(torch.tensor(prob_hit)).item() == 1:
            player.append(int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1)
        else:
            break

    while not (bust(player) or bust(dealer) or score(dealer) > score(player)):
        dealer.append(int(torch.multinomial(torch.ones(13) / 13, 1).item()) + 1)

    payouts.append(1.0 if bust(dealer) else -1.0)

print(f"Estimated expected payout over {10000} games: {np.mean(payouts):.4f}")
```

```
Estimated expected payout over 10000 games: -0.1560
```

The curve looks relatively similar to the original curve, but it takes the player a little longer to increase - however, according to the expected payout over 10000 games, the payout for this game is actually a little bit lower, with an EP of -0.156 compared to the original -0.0622.

Statement of Collaboration (5 points)

It is **mandatory** to include a Statement of Collaboration in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using EdD) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content before they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to EdD, etc.). Especially after you have started working on the assignment, try to restrict the discussion to EdD as much as possible, so that there is no doubt as to the extent of your collaboration.

I did not work with anyone.