Emanuel Budulai
1004070159
February 22, 2023

# Assignment 2

STA414: Statistical Methods for Machine Learning II

*Belief Propogation, Message Passing, Markov Chain Monte Carlo, TrueSkill model*

UNIVERSITY OF TORONTO

## Problem 1: Image Denoising (Belief Propogation, Message Passing)

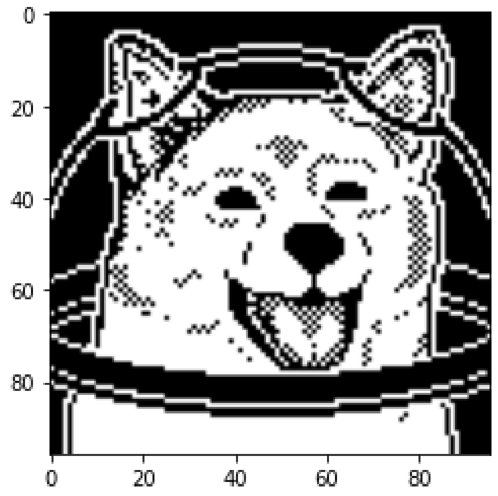
In this problem, we will implement the sum-product Loopy **belief propagation** (Loopy-BP) method for denoising binary images which you have seen in tutorial 4. We will consider images as matrices of size $\sqrt{n} \times \sqrt{n}$. Each element of the matrix can be either 1 or $-1$, with 1 representing white pixels and $-1$ representing black pixels. This is different from the 0/1 representation commonly used for other CV tasks. This notation will be more convenient when multiplying with pixel values.

### Data Preparation

Below we provide you with code for loading and preparing the image data.

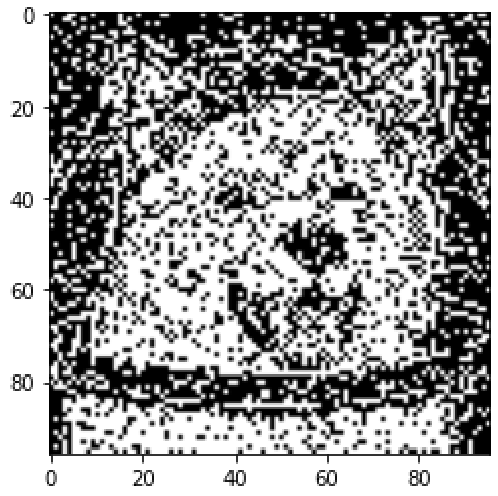
First, we load a black and white image of Laika and convert it into a binary matrix of 1 and -1. So that white pixels have value 1 and black pixels have value -1.

```
!pip install wget

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import PIL.Image as Image
from os.path import exists
from wget import download
from tqdm import tqdm
filename, url = "3vaef0cog4f61.png", "https://i.redd.it/3vaef0cog4f61.png"

def load_img():
    if not exists(filename):
        download(url)

    with open(filename, 'rb') as fp:
        img2 = Image.open(fp).convert('L')
        img2 = np.array(img2)
    return (img2[:96,11:107] > 120) * 2.0 - 1

img_true = load_img()
plt.imshow(img_true, cmap='gray')
```



To introduce noise into the image, for each pixel, swap its value between 1 and -1 with rate 0.2.

```
def gen_noisyimg(img, noise=.05):
    swap = np.random.binomial(1, noise, size=img.shape)
    return img * (2 * swap - 1)

noise = 0.2
img_noisy = gen_noisyimg(img_true, noise)
plt.imshow(-1 * img_noisy, cmap='gray')
```



## The Loopy BP algorithm

Recall from lecture and tutorial, the Loopy-BP algorithm iteratively updates the messages of each node through a sum-product operation. **The sum-product** operation computes the joint inbound message through multiplication, and then marginalizes the factors through summation. This is in contrast to the **max-product** BP, which computes the maximum a-posteriori value for each variable through taking the maximum over variables.

**Initialization:**
For discrete node $x_j$ with 2 possible states, $m_{i \to j}$ can be written as a 2 dimensional real vector $\mathrm{m}_{i,j}$ with $m_{i \to j}(x_j) = \mathrm{m}_{i,j}[index(x_j)]$. We initialize them uniformly to $m_{i \to j}(x_j) = 1/2$.
(Aside: for continuous cases, $m_{i \to j}(x_j)$ is a real valued function of $x_j$. We only need to deal with the discrete case here.)

**For a number of iterations:**
For node $x_j$ in $\{x_s\}_{s=1}^n$:

1. Compute the product of inbound messages from neighbours of $x_j$:

$$\prod_{k \in N(j) \neq i} m_{k \to j}(x_j)$$

2. Compute potentials $\psi_j(x_j) = \exp(\beta x_j y_j)$ and $\psi_{ij}(x_i, x_j) = \exp(J x_i x_j)$. This expression specifically holds when $x \in \{-1, +1\}$.

3. Marginalize over $x_j = \{-1, +1\}$ to get $m_{j \to i}(x_i)$:

$$m_{j \to i}(x_i) = \sum_{x_j} \psi_j(x_j) \psi_{ij}(x_i, x_j) \prod_{k \in N(j) \neq i} m_{k \to j}(x_j)$$

4. Normalize messages for stability $m_{j \to i}(x_i) = m_{j \to i}(x_i) / \sum_{x_i} m_{j \to i}(x_i)$.

**Compute beliefs after message passing is done:**

$$b(x_i) \propto \psi_i(x_i) \prod_{j \in \mathcal{N}(i)} m_{j \to i}(x_i).$$

You'll be tasked to perform steps 1-3 in the iterations and computing the beliefs. We will provide you with helper functions for initialization, finding neighbours, and normalization.

**Initialization**
Initialize the message between neighbor pixels uniformly as $m_{jßi}(x_i) = 1/k$. Since each pixel can only be 1 or -1, message has two values $m_{j \to i}(1)$ and $m_{j \to i}(-1)$. We also initialize hyperparameters $J$, $\beta$.

```
y = img_noisy.reshape([img_true.size, ])
num_nodes = len(y)
init_message = np.zeros([2, num_nodes, num_nodes]) + .5
J = 1.0
beta = 1.0
```

Find the neighboring pixels around a given pixel, which will be used for BP updates

```
def get_neighbors_of(node):
    """
    arguments:
     int node:  in [0,num_nodes) index of node to query
    globals:
     int num_nodes: number of nodes
    return: set(int) indices of neighbors of queried node
    """
    neighbors = []
    m = int(np.sqrt(num_nodes))
    if (node + 1) % m != 0:
        neighbors += [node + 1]
    if node % m != 0:
        neighbors += [node - 1]
    if node + m < num_nodes:
        neighbors += [node + m]
    if node - m >= 0:
        neighbors += [node - m]

    return set(neighbors)
```

1. **(20 Points) Implement message passing in BP**
2. **(10 Points) Computing belief from messages**
3. **Momentum in belief propagation**
   - **3.1. (5 Points) Implement `test_trajectory`**
   - **3.2. (5 Points) Plot a series of images.**
4. **Noise level, $\beta$ and overfitting**
   - **4.1. (5 Points) Generate and display images with noise of $0.05, 0.3$**
   - **4.2. (5 Points) Perform image denoising on images with different noise and $\beta$ levels and plot the denoised images.**

## 1. Implement message passing in BP

Implement the function `get_message()` that computes the message passed from node $j$ to node $i$:

$$m_{j\to i}(x_i) = \sum_{x_j} \psi_j(x_j)\psi_{ij}(x_i, x_j) \prod_{k\in N(j)\neq i} m_{k\to j}(x_j)$$

`get_message()` will be used by (provided below) `step_bp()` to perform one iteration of loopy-BP: it first normalizes the returned message from `get_message()`, and then updates the message with momentum `1.0 - step`.

***Solution.***

```
def get_message(node_from, node_to, messages):
    """
    arguments:
     int node_from: in [0,num_nodes) index of source node
     int node_from: in [0,num_nodes) index of target node
     float array messages: (2, num_nodes, num_nodes), messages[:,j,i] is message
                           from node j to node i
    reads globals:
     float array y: (num_nodes,) observed pixel values
     float J: clique coupling strength constant
     float beta: observation to true pixel coupling strength constant
    return: array(float) of shape (2,) un-normalized message from node_from to
    node_to
    """
    #TODO: implement your function here
    # 1. Compute the product of inbound messages from neighbours of x_j:
    inbound_messages_nodes = get_neighbors_of(node_from).difference(set([node_to]))
    # Use log to reduce numerical error
    log_neighbor_messages = 0.0
    for k in inbound_messages_nodes:
        log_neighbor_messages += np.log(messages[:, k, node_from])

    # 2. Compute potentials (and)
    # 3. Bring together into one array of shape (2, ) to marginalize later
    x_rv = np.array([1.0, -1.0])
    message_1 = np.sum(np.exp((beta * y[node_from] + J) * x_rv \
                              + log_neighbor_messages))
    message_neg1 = np.sum(np.exp((beta * y[node_from] - J) * x_rv \
                             + log_neighbor_messages))

    message = np.array([message_1, message_neg1])

    return message


def step_bp(step, messages):
    """
    arguments:
     float step: step size to update messages
    return
     float array messages: (2, num_nodes, num_nodes), messages[:,j,i] is message
                           from node j to node i
    """
    for node_from in range(num_nodes):
        for node_to in get_neighbors_of(node_from):
            m_new = get_message(node_from, node_to, messages)
            # normalize
            m_new = m_new / np.sum(m_new)

            messages[:, node_from, node_to] = step * m_new + (1. - step) * \
                messages[:, node_from, node_to]
    return messages
```

Then, run loopy BP update for 10 iterations:

***Solution.***

```python
num_iter = 10
step = 0.5
for it in range(num_iter):
    init_message = step_bp(step, init_message)
    print(it + 1,'/',num_iter)
```

which outputs

```
1 / 10
2 / 10
3 / 10
4 / 10
5 / 10
6 / 10
7 / 10
8 / 10
9 / 10
10 / 10
```

**2. Computing belief from messages**

Now, calculate the unnormalized belief for each pixel

$$\tilde{b}(x_i) = \psi_i(x_i) \prod_{j \in N(i)} m_{j \to i}(x_i),$$

and normalize the belief across all pixels

$$b(x_i) = \frac{\tilde{b}(x_i)}{\sum_{x_j} \tilde{b}(x_j)}.$$

***Solution.***

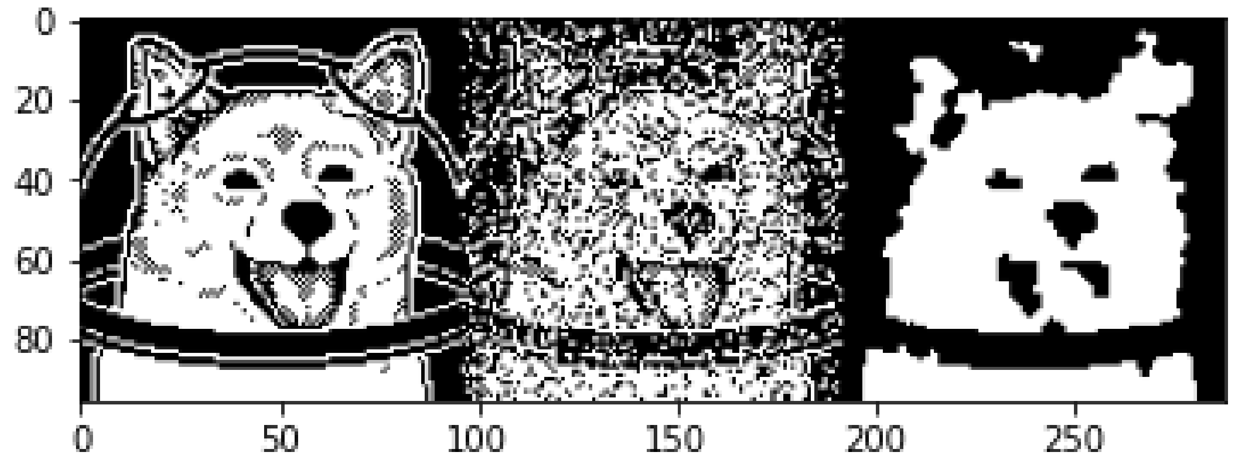
```
def update_beliefs(messages):
    """
    arguments:
    float array messages: (2, num_nodes, num_nodes), messages[:,j,i] is message
                           from node j to node i
    reads globals:
     float beta: observation to true pixel coupling strength constant
     float array y: (num_nodes,) observed pixel values
    returns:
     float array beliefs: (2, num_nodes), beliefs[:,i] is the belief of node i
    """
    x_rv = np.array([1.0, -1.0])
    beliefs = np.zeros([2, num_nodes])
    for node in range(num_nodes):
        #TODO: implement belief calculation here
        # 1. Calculate the unnormalized belief for each pixel
        neighbors = get_neighbors_of(node)
        # Again use log to reduce numerical error
        log_neighbor_messages = 0.0
        for j in neighbors:
            log_neighbor_messages += np.log(messages[:, j, node])

        belief = np.exp(beta * y[node] * x_rv + log_neighbor_messages)

        # 2. Normalize the belief across all pixels
        norm_belief = belief / np.sum(belief)
        beliefs[:, node] = norm_belief

    return beliefs

# call update_beliefs() once
beliefs = update_beliefs(init_message)
```

Finally, to get the denoised image, we use 0.5 as the threshold and consider pixel with belief less than threshold as black while others as white.

```
pred = 2. * ((beliefs[1, :] > .5) + .0) - 1.
img_out = pred.reshape(img_true.shape)

plt.imshow(np.hstack([img_true, -1*img_noisy, img_out]), cmap='gray')
```

which outputs the images:

***Solution.***



## 3. Momentum in belief propagation

In the sample code provided above, we performed message update with a momentum parameter `step`. In this question, you will experimentally investigate how momentum affects the characteristics of convergence.

**3.1.** Complete the function `test_trajectory` below to obtain predicted image after each step of message passing. Return predicted images as list.

***Solution.***

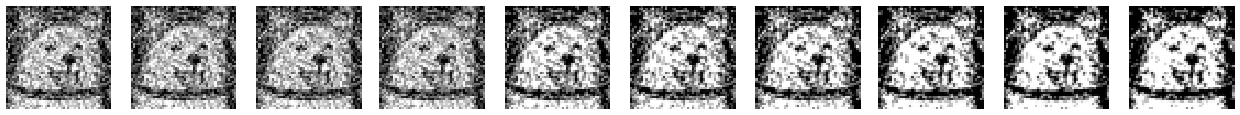
```python
def test_trajectory(step_size, max_step=10):
    """
    step_size: step_size to update messages in each iteration
    max_step: number of steps
    """
    # re-initialize each time
    messages = np.zeros([2, num_nodes, num_nodes]) + .5
    images = []

    for i in range(0, max_step):
        messages = step_bp(step_size, messages)
        updated_beliefs = update_beliefs(messages)
        new_pred = updated_beliefs[1] > updated_beliefs[0]
        img_out = pred.reshape(img_true.shape)
        images.append(img_out)

    return images
```

**3.2.** Use test trajectory to create image serieses for step size 0.1, 0.3, and 1.0, each with 10 steps. Display these images with `plot_series` provided below.

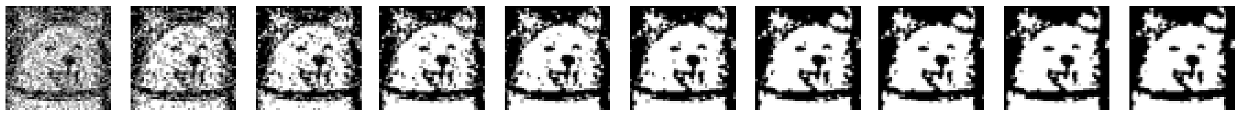
***Solution.***

```python
def plot_series(images):

    n = len(images)
    fig, ax = plt.subplots(1, n)
    for i in range(n):
        ax[i].imshow(images[i], cmap='gray')
        ax[i].set_axis_off()
    fig.set_figwidth(10)
    fig.show()

#Solution:
step_size_lst = [0.1, 0.3, 1.0]
for size in step_size_lst:
    preds = test_trajectory(size)
    plot_series(preds)
```

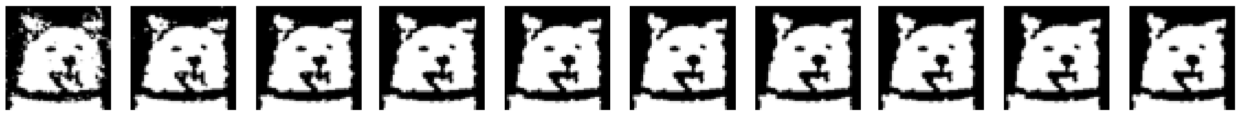
Which gives the following output for the trajectory of step size of 0.1:



step size of 0.3:



step size of 1.0:



In the textbox below, 1. comment on what happens when a large step size is used for too many iterations. 2. how would you adjust other hyperparameters to counteract this effect?

***Solution.***
1. When a large step size is used for too many iterations, then more and more of the pixels will be changed from white to black (i.e. from $-1$ to 1). While this may remove more noise, it may also remove more details from the original image (this can be seen when the step size is 1.0: by the time the 10th iteration is reached, lots of detail has been lost from the dog's ears and eyes). This is due to the fact that the `step_bp` function will more aggressively update the message from one node to another, resulting in far less refinement in the denoising.
2. If the step size is desired to be large, then the number of iterations should be adjusted to be relatively small in order to preserve more of the details in the image. However, the limiting factor is usually the time complexity, so typically it will be the number of iterations that limits the algorithm (especially for more complex images/worse hardware). This means that the minimum possible step size should be chosen according the max possible iterations that could be run on the hardware while

## 4. Noise level, *beta* and overfitting.

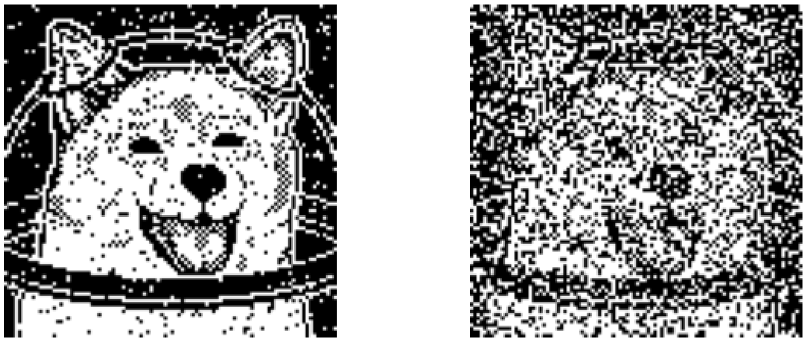
In this question, we will study how the level of noise in the image influences our choice in the hyperparameter $\beta$.

**4.1.** First, generate and display images with noise of 0.05, 0.3.

***Solution.***

```
# Solution
noise_lst = [0.05, 0.3]
images = []
for noise in noise_lst:
    img_noisy = gen_noisyimg(img_true, noise)
    images.append(−1 * img_noisy)

plot_series(images)
```

which outputs the two noised images



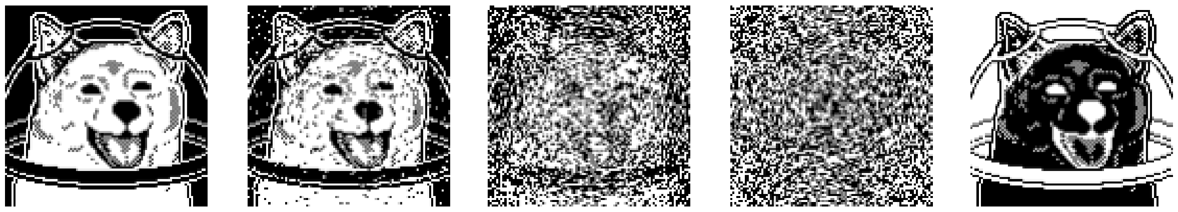
with a noise level of 0.05 and 0.5, respectively.

In the text box below, comment on what would happen if noise was set to 0.5 and 1.0.

***Solution.***

```
noise_lst = [0.05, 0.3, 0.5, 1.0]
images = [img_true]
for noise in noise_lst:
    img_noisy = gen_noisyimg(img_true, noise)
    images.append(−1 * img_noisy)

plot_series(images)
```

giving:



If the noise was set to 0.5 then the image will be completely obscured, and none of the original image will remain (on average). This will make it very hard for the algorithm to establish a belief as to what the original image looked like, since each pixel has a 50 chance of being swapped.

If the noise was set to 1.0, then based on the definition of the `gen_noisying` function, every single pixel in the original image would be swapped since the `noise` parameter supplied to the `np.random.binomial` function is the "probability of success", meaning that when `noise`=1.0 is supplied and a sample size that is the same size of the image is taken from this distribution, each sampled pixel will be a "success" (swapped), thus inverting the image. In other words, all white pixels (-1) will be black (1), and all black pixels (1) will be white (-1), so in effect no noise was generated.

**4.2.** Now, perform image denoising on images with noise levels 0.05 and 0.3 using $\beta = 0.5$, $\beta = 1.0$, $\beta = 2.5$, and $\beta = 5.0$. Set step size to 0.8 and `max_step` to 5. Plot the denoised images (if reusing `test_trajectory`, you should plot 8 image serieses).

***Solution.***

```python
# Solution
step_size = 0.8
max_step = 5

noise_lst = [0.05, 0.3]
beta_lst = [0.5, 1.0, 2.5, 5.0]
for noise in noise_lst:
    img_noisy = gen_noisyimg(img_true, noise)
    for beta in beta_lst:
        preds = test_trajectory(step_size, max_step=5)
        plot_series(preds)
```

Image trajectory for `noise` $= 0.05$, $\beta = 0.5$:

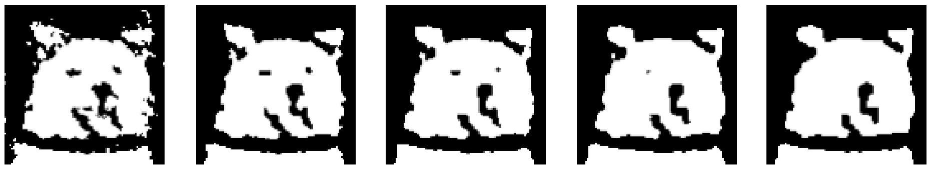


Image trajectory for `noise` $= 0.05$, $\beta = 1.0$:

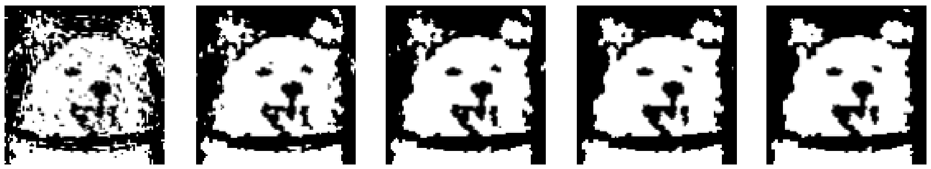


Image trajectory for `noise` $= 0.05$, $\beta = 2.5$:

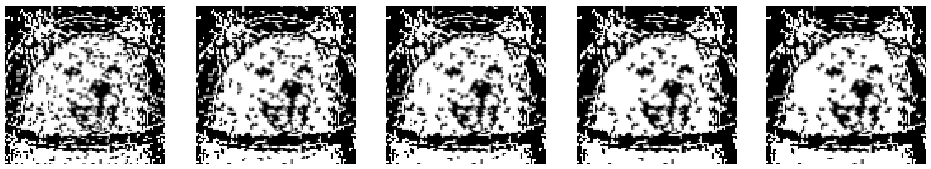


Image trajectory for `noise` $= 0.05$, $\beta = 5.1$:

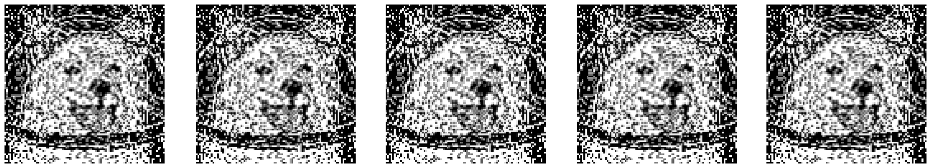


Image trajectory for `noise` $= 0.3$, $\beta = 0.5$:

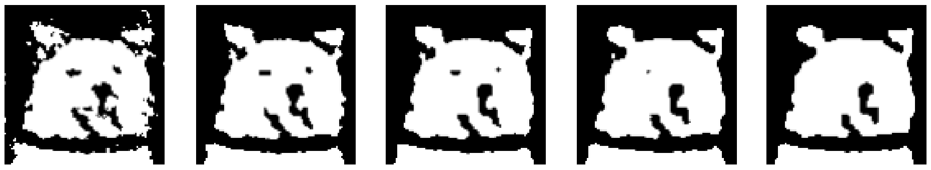


Image trajectory for `noise`$= 0.3$, $\beta = 1.0$:

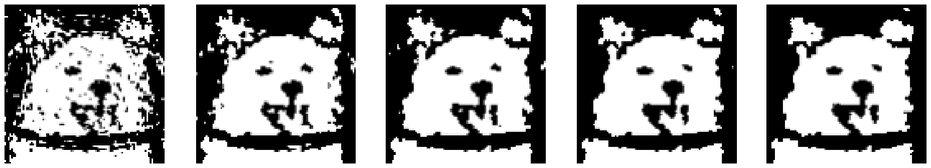


Image trajectory for `noise`$= 0.3$, $\beta = 2.5$:

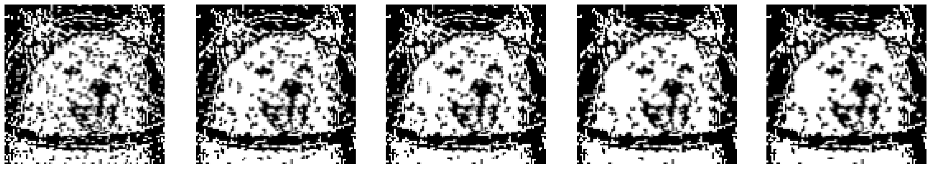
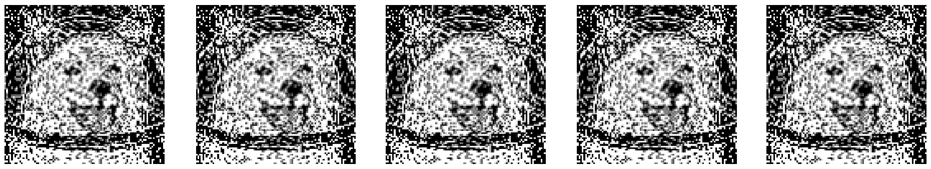


Image trajectory for `noise`$= 0.3$, $\beta = 5.0$:



In text box below, comment on what you observe and provide a brief explanation on why this might occur.

***Solution.***
On average, as the $\beta$ value increases, more of the noise will remain in the image after each iteration of the loopy-bp algorithm.

This occurs because in the `get_message` algorithm, beta is multiplied by the original pixel values, and so the higher the beta, the more likely a pixel value of 1 will remain in the image upon each iteration, and the less likely an update in the message will lower that belief to change the pixel to a white one.

## Problem 2: Markov chain Monte Carlo in the TrueSkill model

The goal of this question is to get you familiar with the basics of Bayesian inference in medium-sized models with continuous latent variables, and the basics of Hamiltonian Monte Carlo.

### Background

We'll implement a variant of the TrueSkill model, a player ranking system for competitive games originally developed for Halo 2. It is a generalization of the Elo rating system in Chess.

This assignment is based on this one developed by Carl Rasmussen at Cambridge for his course on probabilistic machine learning.

### Model definition

We'll consider a slightly simplified version of the original trueskill model. We assume that each player has a true, but unknown skill $z_i \in \mathbb{R}$. We use $N$ to denote the number of players.

**The prior:**
The prior over each player's skill is a standard normal distribution, and all player's skills are *a priori* independent.

**The likelihood:**
For each observed game, the probability that player $i$ beats player $j$, given the player's skills $z_A$ and $z_B$, is:

$$p(A \text{ beat } B | z_A, z_B) = \sigma(z_i - z_j)$$

where

$$\sigma(y) = \frac{1}{1 + \exp(-y)}$$

We chose this function simply because it's close to zero or one when the player's skills are very different, and equals one-half when the player skills are the same. This likelihood function is the only thing that gives meaning to the latent skill variables $z_1 \dots z_N$.

There can be more than one game played between a pair of players. The outcome of each game is independent given the players' skills.

We use $M$ to denote the number of games.

```
!pip install wget
import os
import os.path

import matplotlib.pyplot as plt
import wget

import pandas as pd



import numpy as np
from scipy.stats import norm
import scipy.io
import scipy.stats
import torch
import random
from torch.distributions.normal import Normal

from functools import partial

import matplotlib.pyplot as plt

# Helper function
def diag_gaussian_log_density(x, mu, std):
    # axis=-1 means sum over the last dimension.
    m = Normal(mu, std)
    return torch.sum(m.log_prob(x), axis=-1)
```

1. **Implementing the TrueSkill Model**

   1.a. **(5 pts) Implement a function** `log_joint_prior`

   1.b. **(5 pts) Implement functions** `logp_a_beats_b` **and** `logp_b_beats_a`

2. **Examining the posterior for only two players and toy data**

   2.a. **(3 pts) Plot the isocontours of the joint prior over the skills of two players A and B ($z_A$ and $z_B$)**

   2.b. **(3 pts) Plot isocountours of the joint posterior over $z_A$ and $z_B$ given that player A beat player B in one match**

   2.c. **(2 pts) Plot isocountours of the joint posterior over $z_A$ and $z_B$ given that 10 matches were played, and player A beat player B all 10 times.**

   2.d. **(2 pts) Plot isocontours of the joint posterior over $z_A$ and $z_B$ given that 20 matches were played, and each player beat the other 10 times.**

3. **Hamiltonian Monte Carlo on Two Players and Toy Data**

   3.a. **(Free) Approximate the joint posterior where we observe player A winning 1 game, sampling from HMC.**

   3.b. **(3 pts) Approximate the joint posterior where we observe player A winning 10 games against player B, sampling from HMC.**

   3.c. **(3 pts) Approximate the joint posterior where we observe player A winning 10 games and player B winning 10 games, sampling from HMC.**

4. **Approximate inference conditioned on real data**

   4.a.1) **(4 pts) Implement a function** `log_games_likelihood` **that returns the total log-likelihoods given a collection of observed games** `games`**.**

   4.a.2) **(3 pts) Implement a function** `joint_log_density` **which combines the log-prior and log-likelihood of the observations to give** $p(z_1, z_2, \ldots, z_N, \text{all game outcomes})$

**4.b. (3 pts) Run Hamiltonian Monte Carlo for 10000 samples.**

**4.c. (3pts) Plot the approximate mean and variance of the marginal skill of each player, sorted by average skill of the samples.**

**4.d. (3 pts) List the names of the 5 players with the lowest mean skill and 5 players with the highest mean skill according to the samples.**

**4.e. (3 pts) Use a scatterplot to show your samples from the joint posterior over the skills of lelik3310 and thebestofthebad.**

**4.f. (3 pts) Comparing probabilitiies regarding the skill of two players, as estimated from the samples.**

**4.g. (3 pts) Do the games between other players besides $i$ and $j$ provide information about the skill of players $i$ and $j$?**

## 1. Implementing the TrueSkill Model

**1.a.** Implement a function `log_joint_prior` that computes the log of the prior, jointly evaluated over all player's skills.

Specifically, given a $K \times N$ array where each row is a setting of the skills for all $N$ players, it returns a $K \times 1$ array, where each row contains a scalar giving the log-prior for that set of skills.

***Solution.***

```python
def log_joint_prior(zs_array):
    # Hint: Use diag_gaussian_log_density
    # TODO
    return diag_gaussian_log_density(zs_array, 0.0, 1.0)
```

**1.b.** Implement two functions `logp_a_beats_b` and `logp_b_beats_a`.

Given a pair of skills $z_a$ and $z_b$, `logp_a_beats_b` evaluates the log-likelihood that player with skill $z_a$ beat player with skill $z_b$ under the model detailed above, and `logp_b_beats_a` is vice versa.

To ensure numerical stability, use the function `torch.logaddexp`

***Solution.***

```python
def logp_a_beats_b(z_a, z_b):
    # Hint: Use torch.logaddexp
    # TODO
    return -torch.logaddexp(torch.tensor(0),
                            torch.tensor(z_b) - torch.tensor(z_a))

def logp_b_beats_a(z_a, z_b):
    # Hint: Use torch.logaddexp
    # TODO
    return -torch.logaddexp(torch.tensor(0),
                            torch.tensor(z_a) - torch.tensor(z_b))
```

## 2. Examining the posterior for only two players and toy data

To get a feel for this model, we'll first consider the case where we only have 2 players, $A$ and $B$. We'll examine how the prior and likelihood interact when conditioning on different sets of games.

Provided in the starter code is a function `skillcontour` which evaluates a provided function on a grid of $z_A$ and $z_B$'s and plots the isocontours of that function. As well there is a function `plot_line_equal_skill`. We have included an example for how you can use these functions.

We also provided a function `two_player_toy_games` which produces toy data for two players. I.e. `two_player_toy_games(5,3)` produces a dataset where player A wins 5 games and player B wins 3 games.

```python
# Plotting helper functions
def plot_isocontours(ax, func, steps=100):
    x = torch.linspace(-4, 4, steps=steps)
    y = torch.linspace(-4, 4, steps=steps)
    X, Y = torch.meshgrid(x, y)
    Z = func(X, Y)
    #Z = zs.reshape(X.shape)
    plt.contour(X, Y, Z )
    ax.set_yticks([])
    ax.set_xticks([])

def plot_2d_fun(f, x_axis_label="", y_axis_label="", scatter_pts=None):
    # This is the function your code should call.
    # f() should take two arguments.
    fig = plt.figure(figsize=(8,8), facecolor='white')
    ax = fig.add_subplot(111, frameon=False)
    ax.set_xlabel(x_axis_label)
    ax.set_ylabel(y_axis_label)
    plot_isocontours(ax, f)
    if scatter_pts is not None:
      plt.scatter(scatter_pts[:,0], scatter_pts[:, 1])
    plt.plot([4, -4], [4, -4], 'b--')   # Line of equal skill
    plt.show(block=True)
    plt.draw()
```

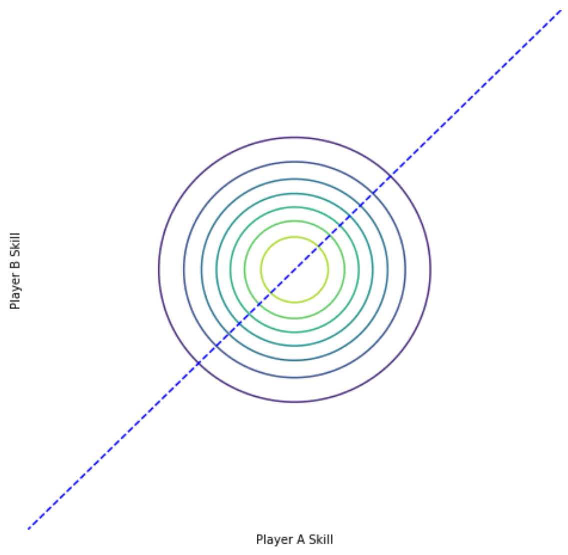
**2.a.** For two players $A$ and $B$, plot the isocontours of the joint prior over their skills. Also plot the line of equal skill, $z_A = z_B$. You've already implemented the log of this function, you just need to graph it using the helper function `plot_2d_fun` above.

For this and the following plots, label both axes.

***Solution.***

```python
def log_prior_over_2_players(z1, z2):
    #return z1+z2
    m = Normal(torch.tensor([0.0]), torch.tensor([[1.0]]))
    return m.log_prob(z1) + m.log_prob(z2)
```

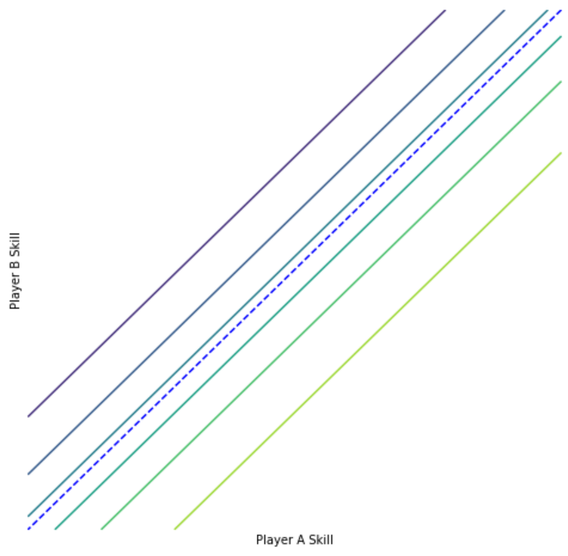
```
def prior_over_2_players(z1, z2):
    #TODO
    return torch.exp(log_prior_over_2_players(z1, z2))

plot_2d_fun(prior_over_2_players, "Player A Skill", "Player B Skill")
```



According to the prior, what's the chance that player A is better than player B?

***Solution.***

```
# Note:  This isn't part of the assignment
def likelihood_over_2_players(z1, z2):
    return torch.exp(logp_a_beats_b(z1, z2))

plot_2d_fun(likelihood_over_2_players, "Player A Skill", "Player B Skill")
```

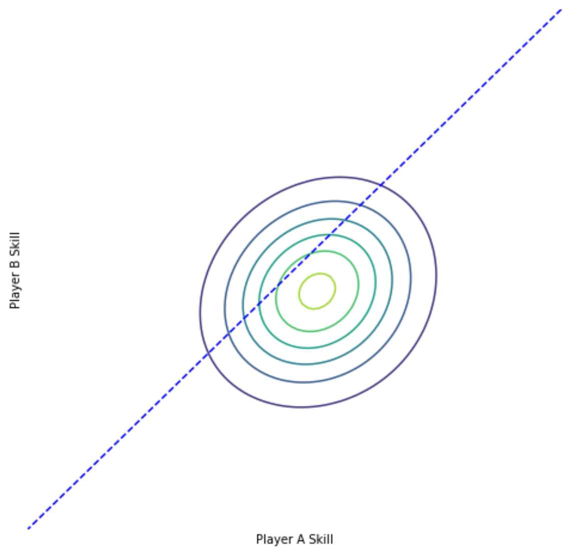


**2.b.** Plot isocountours of the joint posterior over $z_A$ and $z_B$ given that player A beat player B in one match. Since the contours don't depend on the normalization constant, you can simply plot the isocontours of the log of joint distribution of $p(z_A, z_B, \text{A beat B})$. Also plot the line of equal skill, $z_A = z_B$.

***Solution.***

```
def log_posterior_A_beat_B(z1, z2):
    # TODO: Combine the prior for two players with the likelihood for A beat B.
    # You might want to use the log_prior_over_2_players function from above.
    return log_prior_over_2_players(z1, z2) + logp_a_beats_b(z1, z2)

def posterior_A_beat_B(z1, z2):
    return torch.exp(log_posterior_A_beat_B(z1, z2))

plot_2d_fun(posterior_A_beat_B, "Player A Skill", "Player B Skill")
```

which outputs the following plot:



To think about: According to this posterior, which player is likely to have higher skill?

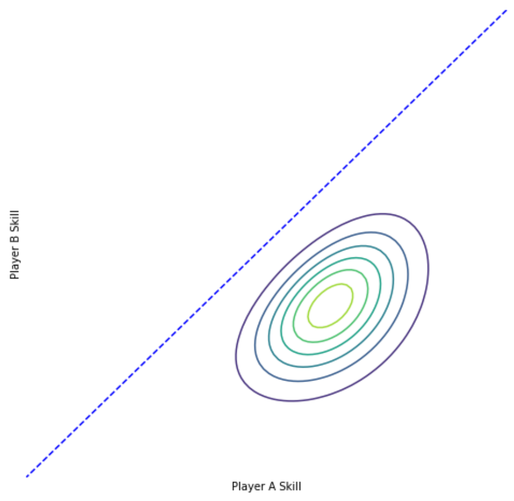
***Solution.***

**2.c.** Plot isocountours of the joint posterior over $z_A$ and $z_B$ given that 10 matches were played, and player A beat player B all 10 times. Also plot the line of equal skill, $z_A = z_B$.

***Solution.***

```
def log_posterior_A_beat_B_10_times(z1, z2):
    # TODO: Combine the prior for two players with the likelihood for A beat B.
    # You might want to use your log_prior_over_2_players function from above.
    return log_prior_over_2_players(z1, z2) + 10 * logp_a_beats_b(z1, z2)

def posterior_A_beat_B_10_times(z1, z2):
    return torch.exp(log_posterior_A_beat_B_10_times(z1, z2))
```

```
plot_2d_fun(posterior_A_beat_B_10_times, "Player A Skill", "Player B Skill")
```



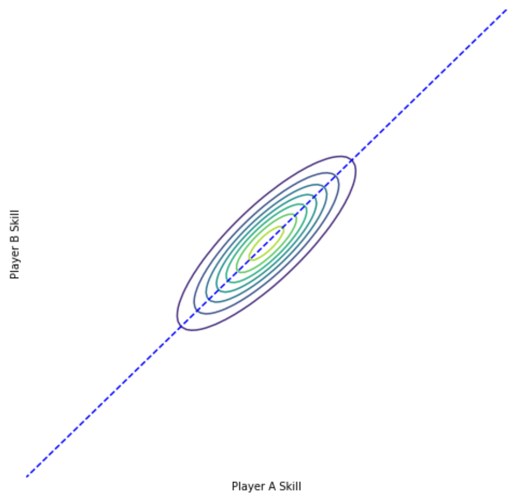
To think about: According to this posterior, is it plausible that player B is more skilled than player A?

***Solution.***

**2.d.** Plot isocontours of the joint posterior over $z_A$ and $z_B$ given that 20 matches were played, and each player beat the other 10 times. Also plot the line of equal skill, $z_A = z_B$.

***Solution.***

```
def log_posterior_beat_each_other_10_times(z1, z2):
    # TODO: Combine the prior for two players with the likelihood for A beat B.
    # You might want to use your log_prior_over_2_players function from above.
    return log_prior_over_2_players(z1, z2) + 10 * logp_a_beats_b(z1, z2) \
    + 10 * logp_b_beats_a(z1, z2)

def posterior_beat_each_other_10_times(z1, z2):
    return torch.exp(log_posterior_beat_each_other_10_times(z1, z2))
plot_2d_fun(posterior_beat_each_other_10_times, "Player A Skill", "Player B Skill")
```



To think about: According to this posterior, is it likely that one player is much better than another? Is it plausible that both players are better than average? Worse than average?

***Solution.***

## 3. Hamiltonian Monte Carlo on Two Players and Toy Data

One nice thing about a Bayesian approach is that it separates the model specification from the approximate inference strategy. The original Trueskill paper from 2007 used message passing. Carl Rasmussen's assignment uses Gibbs sampling.

In this question we will approximate posterior distributions with gradient-based Hamiltonian Monte Carlo.

In the next assignment, we'll use gradient-based stochastic variational inference, which wasn't invented until around 2014.

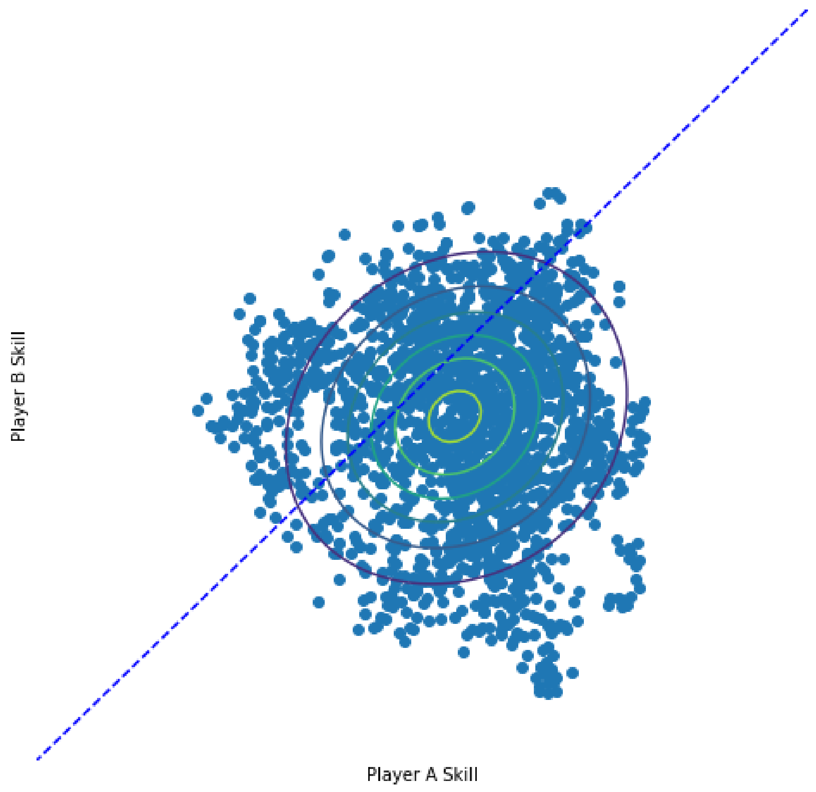
```
random.seed(0)
```

```
# Hamiltonian Monte Carlo
from tqdm import trange, tqdm_notebook  # Progress meters

def leapfrog(params_t0, momentum_t0, stepsize, logprob_grad_fun):
    # Performs a reversible update of parameters and momentum
    # See https://en.wikipedia.org/wiki/Leapfrog_integration
    momentum_thalf = momentum_t0    + 0.5 * stepsize * logprob_grad_fun(params_t0)
    params_t1 =      params_t0      +       stepsize * momentum_thalf
    momentum_t1 =    momentum_thalf + 0.5 * stepsize * logprob_grad_fun(params_t1)
    return params_t1, momentum_t1


def iterate_leapfrogs(theta, v, stepsize, num_leapfrog_steps, grad_fun):
    for i in range(0, num_leapfrog_steps):
        theta, v = leapfrog(theta, v, stepsize, grad_fun)
    return theta, v

def metropolis_hastings(state1, state2, log_posterior):
    # Compares the log_posterior at two values of parameters,
    # and accepts the new values proportional to the ratio of the posterior
    # probabilities.
    accept_prob = torch.exp(log_posterior(state2) - log_posterior(state1))
    if random.random() < accept_prob:
        return state2  # Accept
    else:
        return state1  # Reject

def draw_samples(num_params, stepsize, num_leapfrog_steps, n_samples, log_posterior):
    theta = torch.zeros(num_params)

    def log_joint_density_over_params_and_momentum(state):
        params, momentum = state
        return diag_gaussian_log_density(momentum, torch.zeros_like(momentum), torch.
    ones_like(momentum)) \
        + log_posterior(params)

    def grad_fun(zs):
        zs = zs.detach().clone()
        zs.requires_grad_(True)
        y = log_posterior(zs)
        y.backward()
        return zs.grad

    sampleslist = []
    for i in trange(0, n_samples):
        sampleslist.append(theta)

        momentum = torch.normal(0, 1, size = np.shape(theta))

        theta_new, momentum_new = iterate_leapfrogs(theta, momentum, stepsize,
    num_leapfrog_steps, grad_fun)

        theta, momentum = metropolis_hastings((theta, momentum), (theta_new,
    momentum_new), log_joint_density_over_params_and_momentum)
    return torch.stack((sampleslist))
```

**3.a.** Using samples generated by HMC, approximate the joint posterior where we observe player A winning 1 game.

***Solution.***

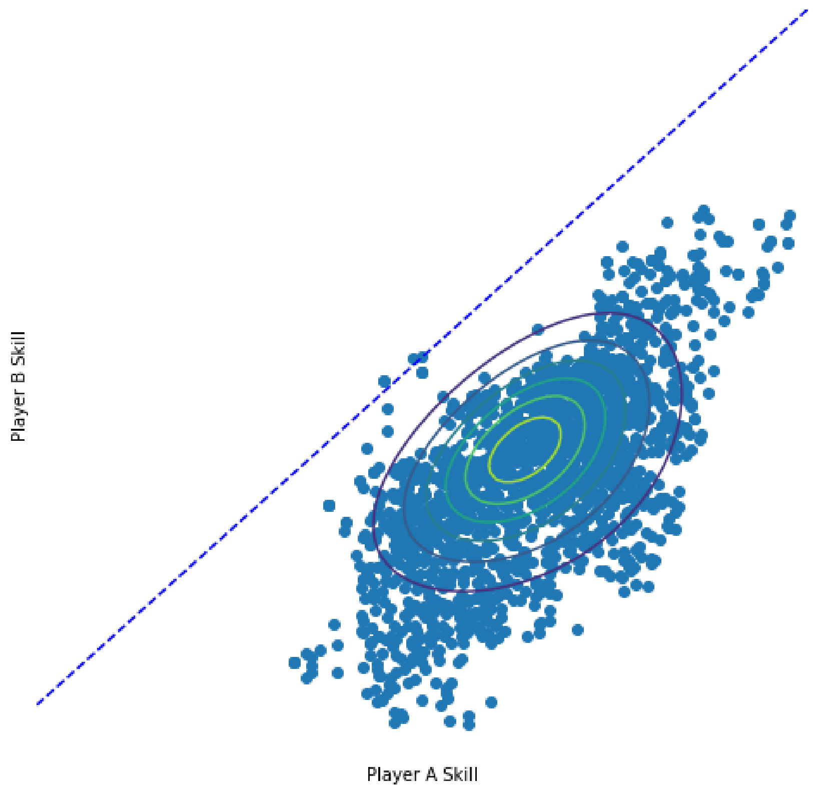
```
# Hyperparameters
num_players = 2
num_leapfrog_steps = 20
n_samples = 2000
stepsize = 0.01

def log_posterior_a(zs):
    return log_posterior_A_beat_B(zs[0], zs[1])

samples_a = draw_samples(num_players, stepsize, num_leapfrog_steps, n_samples,
    log_posterior_a)
plot_2d_fun(posterior_A_beat_B, "Player A Skill", "Player B Skill", samples_a)
```



**3.b.** Using samples generated by HMC, approximate the joint posterior where we observe player A winning 10 games against player B. Hint: You can re-use the code from when you plotted the isocontours.

***Solution.***

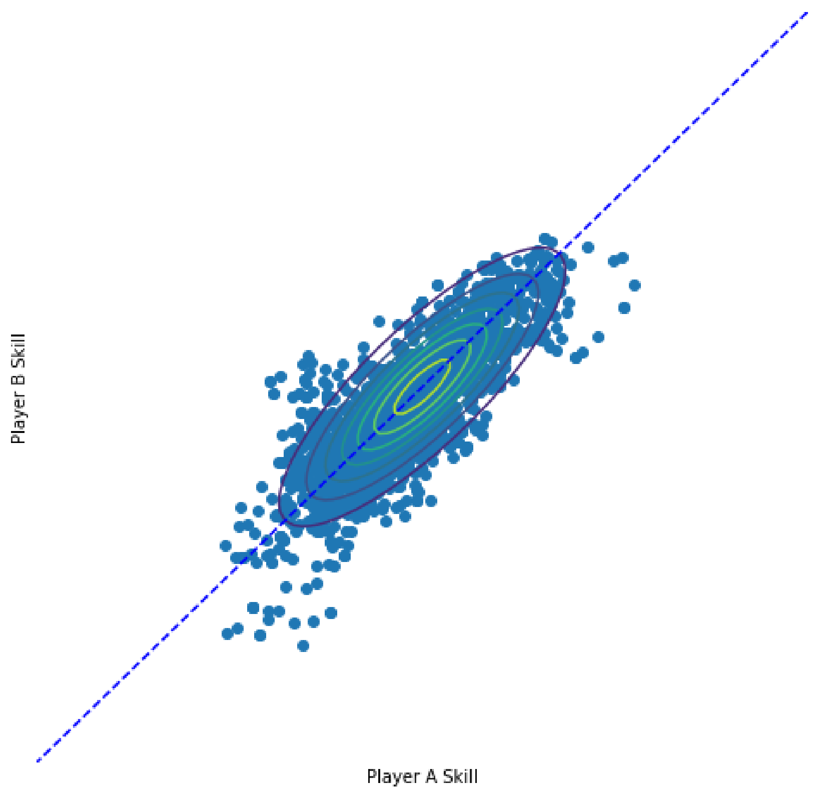
```
# Hyperparameters
num_players = 2
num_leapfrog_steps = 20
n_samples = 2000
stepsize = 0.01
key = 42

def log_posterior_b(zs):
    return log_posterior_A_beat_B_10_times(zs[0], zs[1])

samples_b = draw_samples(num_players, stepsize, num_leapfrog_steps, n_samples,
    log_posterior_b)
ax = plot_2d_fun(posterior_A_beat_B_10_times, "Player A Skill", "Player B Skill",
    samples_b)
```



**3.c.** Using samples generated by HMC, approximate the joint posterior where we observe player A winning 10 games and player B winning 10 games.

***Solution.***

```
# Hyperparameters
num_players = 2
num_leapfrog_steps = 20
n_samples = 2000
stepsize = 0.01

def log_posterior(zs):
    return log_posterior_beat_each_other_10_times(zs[0], zs[1])

samples_c = draw_samples(num_players, stepsize, num_leapfrog_steps, n_samples,
    log_posterior)
ax = plot_2d_fun(posterior_beat_each_other_10_times, "Player A Skill", "Player B Skill"
    , samples_c)
```



## 4. Approximate inference conditioned on real data

The dataset contains data on 2546 chess games amongst 1434 players:

- names is a 1434 by 1 matrix, whose $i$'th entry is the name of player $i$.
- games is a 2546 by 2 matrix of game outcomes (actually chess matches), one row per game.

The first column contains the indices of the players who won. The second column contains the indices of the player who lost.

It is based on the kaggle chess dataset: `https://www.kaggle.com/datasets/datasnaek/chess`

```
wget.download("https://erdogdu.github.io/sta414/hw/hw2/chess_games.csv")
games = pd.read_csv("chess_games.csv")[["winner_index", "loser_index"]].to_numpy()
wget.download("https://erdogdu.github.io/sta414/hw/hw2/chess_players.csv")
names = pd.read_csv("chess_players.csv")[["index", "player_name"]].to_numpy()
```

```
games = torch.IntTensor(games)
```

**4.a.1)** Assuming all game outcomes are i.i.d. conditioned on all players' skills, implement a function `log_games_likelihood` that takes a batch of player skills `zs` and a collection of observed games `games` and gives the total log-likelihoods for all those observations given all the skills.
Hint: You should be able to write this function without using `for` loops, although you might want to start that way to make sure what you've written is correct. If $A$ is an array of integers, you can index the corresponding entries of another matrix $B$ for every entry in $A$ by writing `B[A]`.

***Solution.***

```
def log_games_likelihood(zs, games):
    # games is an array of size (num_games x 2)
    # zs is an array of size num_players
    #
    # Hint: With broadcasting, this function can be written
    # with no for loops.
    #
    winning_player_ixs = games[:,0].type(torch.LongTensor)
    losing_player_ixs = games[:,1].type(torch.LongTensor)
    #TODO: Look up the skill of the winning player in each game.
    winning_player_skills = zs[winning_player_ixs]
    #TODO: Look up the skills of the losing player in each game.
    losing_player_skills = zs[losing_player_ixs]
    #TODO: Compute the log_likelihood of each game outcome.
    log_likelihoods = logp_a_beats_b(winning_player_skills, losing_player_skills)
    #TODO: Combine the log_likelihood of independent events.
    return torch.sum(log_likelihoods)
```

**4.a.2)** Implement a function `joint_log_density` which combines the log-prior and log-likelihood of the observations to give $p(z_1, z_2, \ldots, z_N, \text{all game outcomes})$

***Solution.***

```python
def log_joint_probability(zs, games):
    # TODO: Combine log_prior and log_likelihood.
    return log_joint_prior(zs) + log_games_likelihood(zs, games)
```

**4.b.** Run Hamiltonian Monte Carlo on the posterior over all skills conditioned on all the chess games from the dataset. Run for 10000 samples.

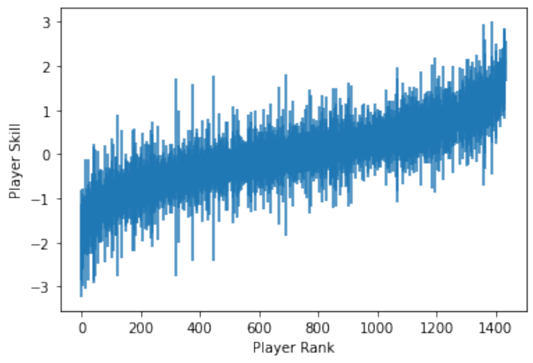
***Solution.***

```python
# Hyperparameters
num_players = 1434
num_leapfrog_steps = 20
n_samples = 10000
stepsize = 0.01

#Hint: you will need to use games
def log_posterior(zs):
    return log_joint_probability(zs, games)

all_games_samples = draw_samples(num_players, stepsize, num_leapfrog_steps, n_samples,
    log_posterior)
```

**4.c.** Based on your samples from the previous question, plot the approximate mean and variance of the marginal skill of each player, sorted by average skill. There's no need to include the names of the players. Label the axes "Player Rank", and "Player Skill".

***Solution.***

```python
mean_skills = torch.mean(all_games_samples, axis = 0)
var_skills = torch.var(all_games_samples, axis = 0)

order = np.argsort(mean_skills)

plt.xlabel("Player Rank")
plt.ylabel("Player Skill")
plt.errorbar(range(num_players), mean_skills[order], var_skills[order])
```



**4.d.** List the names of the 5 players with the lowest mean skill and 5 players with the highest mean skill according to your samples. Hint: you can re-use 'order' from the previous question.

***Solution.***

```
num_order = 5

# Top 5 players with the lowest mean skill
for j in range(num_order):
    print(names[order[j]])

# Top 5 players with the highest mean skill
for i in range(num_order):
    print(names[order[-(i + 1)]])
```

The following are the 5 players with the lowest mean skill (output from the first loop):

```
[512 'thebestofthebad']
[840 'shijima220']
[414 'schoonied']
[0 'lukarpov']
[1245 'javi_r']
```

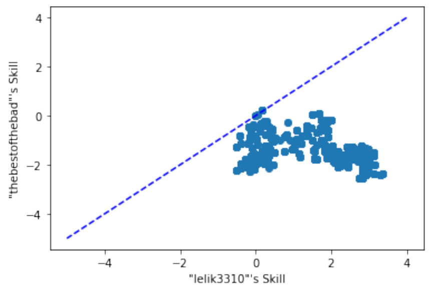
and the 5 players with the highest mean skill (output from the second loop):

```
[841 'cdvh']
[1180 'trouty1997']
[1337 'lcf64']
[303 'mrzoom47']
[618 'meme_master']
```

**4.e.** Use a scatterplot to show your samples from the joint posterior over the skills of lelik3310 and thebestofthebad. Include the line of equal skill. Hint: you can use `plt.scatter`.

***Solution.***

```
lelik3310_ix = 496
thebestofthebad_ix = 512

plt.xlabel("\"lelik3310\"'s Skill")
plt.ylabel("\"thebestofthebad\"'s Skill")
plt.plot([4, -5], [4, -5], 'b--')   # Line of equal skill
plt.scatter(all_games_samples[:, lelik3310_ix], all_games_samples[:, thebestofthebad_ix
    ])
```



**4.f.** Using your samples, find the players that have the eleventh highest mean skill. Print an unbiased estimate of the probability that the player with the eleventh highest highest mean skill is not worse than lelik3310, again as estimated from your samples.

Hint: Probabilities of Bernoulli random variables can be written as the expectation that the Bernoulli takes value 1, so you can use simple Monte Carlo. The final formula will be very simple.

***Solution.*** Since the MLE for $\theta$ of a Bernoulli random variable is simply

$$\hat{\theta} = \frac{\sum_i X_i}{N} = \mathbb{E}(X)$$

it follows that the mean number of times that the eleventh highest mean skill player is *not worse** than 'lelik3310' (i.e. $z_{\text{11th highest skill}} \geq z_{\text{lelik3310}}$) is given by

```
print(np.round(torch.mean((all_games_samples[:, order[-11]]
                          >= all_games_samples[:, lelik3310_ix]).float()).item(), 4))
```

which outputs

```
0.2974
```

i.e. the probability that the player with the eleventh highest highest mean skill is not worse than lelik3310 is aproximately 29.74%.

** here, "not worse" is taken to mean "better than or as good as", i.e.* $\geq$

**4.g.** For any two players $i$ and $j$, $p(z_i, z_j|\text{all games})$ is always proportional to $p(z_i, z_j, \text{all games})$, as a function of $z_i$ and $z_j$.

In general, are the isocontours of $p(z_i, z_j|\text{all games})$ the same as those of $p(z_i, z_j|\text{games between } i \text{ and } j)$? That is, do the games between other players besides $i$ and $j$ provide information about the skill of players $i$ and $j$? A simple yes or no suffices.

Hint: One way to answer this is to draw the graphical model for three players, $i$, $j$, and $k$, and the results of games between all three pairs, and then examine conditional independencies. If you do this, include the graphical models in your assignment.

***Solution.*** No, in general, the isocontours of $p(z_i, z_j|\text{all games})$ are not the same as those of $p(z_i, z_j|\text{games between } i \text{ and } j)$. This is because the games that are played between player $i$ and a third player $k$ (a different from $i$ and $j$) and the games played between player $j$ and $k$ both provide information about the skill of players $i$ and $j$. This can be generalized for all other $N-2$ players, i.e. for all possible games.