Probabilistic ML: Assignment 2

- **Deadline**: Feb 16, 23:59 ET
- **Submission**: You need to submit your solutions through Crowdmark, including all your derivations, plots, and your code. You can produce the files however you like (e.g. $\LaTeX$, Microsoft Word, etc), as long as it is readable. Points will be deducted if we have a hard time reading your solutions or understanding the structure of your code.
- **Collaboration policy**: After attempting the problems on an individual basis, you may discuss and work together on the assignment with up to two classmates. However, **you must write your own code and write up your own solutions individually and explicitly name any collaborators** at the top of the homework.

Q1 - Image Denoising

In this problem, we will implement the max-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.

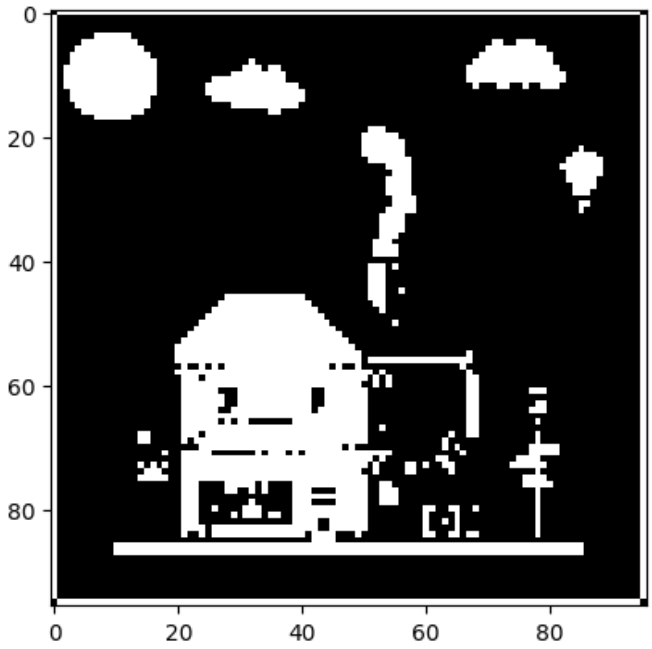
Data preparation

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

First, we load a black and white image 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 = "trc1l3gqu9651.png", "https://i.redd.it/trc1l3gqu9651.png"

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

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

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

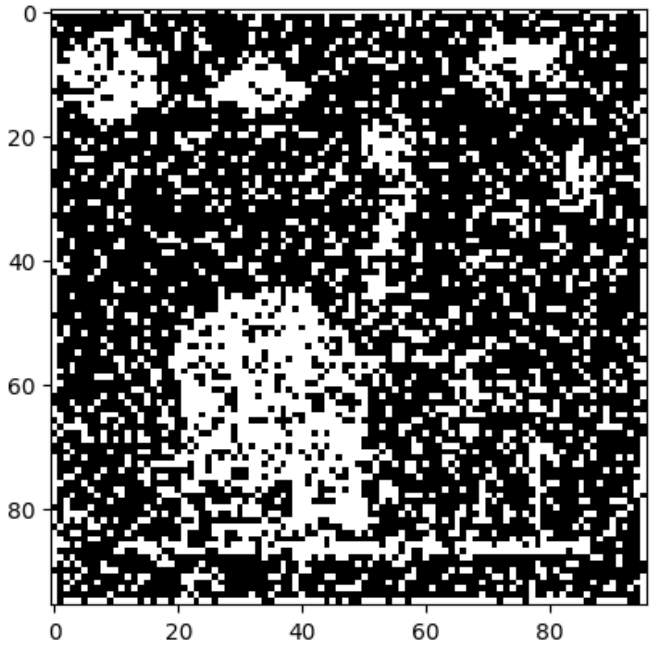
```
Collecting wget
  Downloading wget-3.2.zip (10 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9656 sha256=0ca3f70ffa7fc36b774370b0
  Stored in directory: /root/.cache/pip/wheels/40/b3/0f/a40dbd1c6861731779f62cc4babcb234387e11d697df70
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
<matplotlib.image.AxesImage at 0x7caa84d08c50>
```



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 * (1 - 2 * swap)

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

<matplotlib.image.AxesImage at 0x7caa84078b50>



The Loopy BP algorithm

Recall from lecture and tutorial, the Loopy-BP algorithm iteratively updates the messages of each node through a sum-product or max-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 (MAP) value for each variable through taking the maximum over variables. In this question, we will implement the **max-product** BP to obtain the MAP estimate

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. Maximize over $x_j = \{-1, +1\}$ to get $m_{j\to i}(x_i)$:
$$m_{j\to i}(x_i) = \max_{x_j} \psi_j(x_j)\psi_{ij}(x_i, x_j) \prod_{k\in N(j)\neq i} m_{k\to j}(x_j)$$
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.

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

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 $m_{j\to i}(x_i) = 1/k$. 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 and β.

```
# we observe the noisy pixels, the y_i's
# √n x √n image
y = img_noisy.reshape([img_true.size, ])
# vectorize the image R^n
num_nodes = len(y)
init_message = np.zeros([2, num_nodes, num_nodes]) + .5 # n x 1 vector
# initialize the J and β
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)
```

Q1.1 Implement message passing in BP [20 points]

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

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

`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`.

```
def get_message(node_from, node_to, messages):
  """
  Computes the message passed from node_from (x_j) to node_to (x_i)
  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] (m_ij) 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
  """
  # node_from is x_j, node_to is x_i
  states = np.array([1.0, -1.0])

  # 1. Compute the product of inbound messages from neighbours of  xj :
  neighbors = get_neighbors_of(node_from) - set([node_to]) # remove i from N(j)
  neighbor_messages = np.array([messages[:, n, node_from] for n in neighbors])
  prod_mess = np.prod(neighbor_messages)

  # 2. Compute potentials  ψj(xj)=exp(βxjyj)  and  ψij(xi,xj)=exp(Jxixj).
  # This expression specifically holds when  x∈{-1,+1}.
  psi_j = np.exp(beta * states * y[node_from])
  psi_ij = np.exp(np.outer(states, states) * J)

  # 3. Maximize over  xj={-1,+1}  to get  mj→i(xi):
  m_jis = psi_j * psi_ij * prod_mess
  return np.max(m_jis, axis=1)

# 4. Normalize messages for stability  mj→i(xi)=mj→i(xi)/∑ximj→i(xi)
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:

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

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

Q1.2 Computing belief from messages [10 points]

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

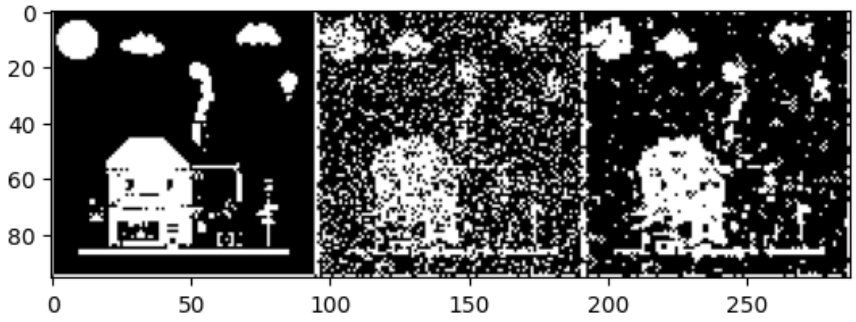
$$b(x_i) = \frac{\tilde{b(x_i)}}{\sum_{x_j} \tilde{b(x_j)}}.$$

```
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
    """
    # Computing beliefs after message passing is done
    states = np.array([1, -1])
    beliefs = np.zeros([2, num_nodes])

    # calculate the unnormalized belief for each pixel
    for node in range(num_nodes):
      # compute Ψ_i
      psi_i = np.exp(beta * states * y[node])

      # product mji(xi) for all neighbors of node i
      neighbors = get_neighbors_of(node)
      neighbor_messages = np.array([messages[:, n, node] for n in neighbors])
      prod_j_Ni = np.prod(neighbor_messages, axis=0)

      # compute the belief: b(x_i) = Ψ_i(x_i) • (π mji(xi) for all neighbors of node i)
      b_xi = psi_i * prod_j_Ni

      # update the belief
      beliefs[:, node] = b_xi

    # normalize the belief across all pixels
    beliefs = beliefs / np.sum(beliefs, axis=0)
    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, which is the same as choosing the pixel with maximum probability

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

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

```
<matplotlib.image.AxesImage at 0x7caa84e38c50>
```



Q1.3 Momentum in belief propagation [10 points]

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.

Q1.3.a [5 points]

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

```
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
    predicted_images = []

    # solution:
    for it in range(max_step):
        messages = step_bp(step_size, messages)
        beliefs = update_beliefs(messages)
        # binary predicted image
        predicted_image = 2. * ((beliefs[0, :] > .5) + .0) - 1.
        predicted_image = predicted_image.reshape(img_true.shape)
        predicted_images.append(predicted_image)
        # images.append(beliefs[0, :].reshape(img_true.shape)) # this was wrong bc it did grayscale

    return predicted_images
```

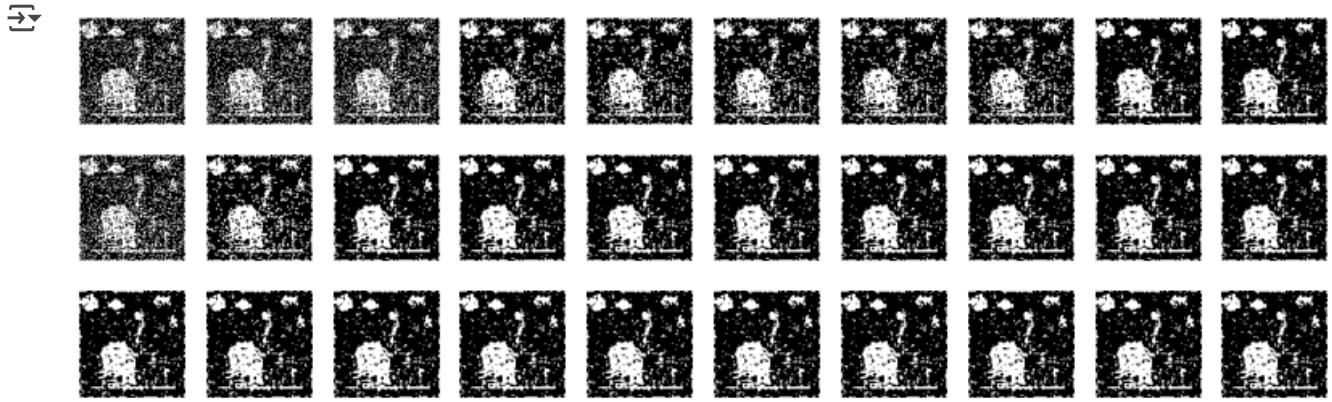
Q1.3.b [5 points]

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.

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?

```
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_sizes = [0.1, 0.3, 1.0]
steps = 10
for step_size in step_sizes:
  image_series = test_trajectory(step_size, steps)
  plot_series(image_series)
```



Response to 1.3.b (Enter your response below):

The step size of 0.1 appears to improve the image gradually, whereas the step size of 1.0 doesn't really improve the image much. This could be because when a large step size is used for too many iterations, loopy BP will have problems with convergence due to instability & overshooting. A large step size for too many iterations may cause divergence or cause the updates to overshoot.

How I would adjust hyperparameters to counteract this effect:

- reducing the step size for smaller updates, to avoid overshooting, and to try to get smoother convergence.
- increasing the number of iterations because smaller step size & more iterations would allow for exploration with less risk of divergence.

Q1.4 Noise level and the hyperparameter J [10 points]

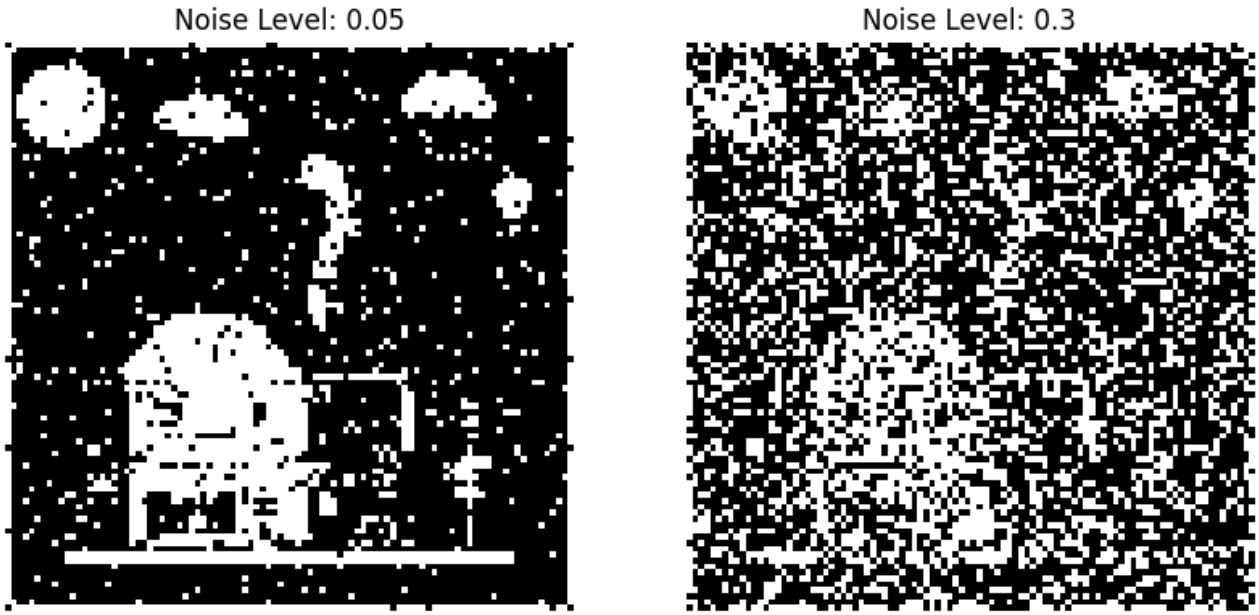
In this question, we will study how the level of noise in the image influences our choice in the hyperparameter J.

Q1.4.a [5 points]

First, generate and display images with noise of $0.05, 0.3$. In the text box below, comment on what would happen if noise was set to 0.5 and 1.0

```
# Solution

noisy_images = []
noise_levels = [0.05, 0.3]
for noise_level in noise_levels:
  noisy_image = gen_noisyimg(img_true, noise_level)
  noisy_images.append((noisy_image, noise_level))
```

```
# code to make the plots side by side. i used chat gpt to make the plots show up side by side
fig, axes = plt.subplots(1, len(noise_levels), figsize=(10, 5))  # only 3 noise levels here right now

for ax, (noisy_image, noise_level) in zip(axes, noisy_images):
    ax.imshow(noisy_image, cmap='gray')
    ax.set_title(f'Noise Level: {noise_level}')
    ax.axis('off')

plt.show()

# ------------------------------------------------------------------------------
# my original code before asking chat gpt to make them side by side
# for noisy_image, noise_level in noisy_images:
#   plt.imshow(noisy_image, cmap='gray')
#   plt.title(f'Noisy Image (Noise = {noise_level})')
#   plt.axis('off')
#   plt.show()
```



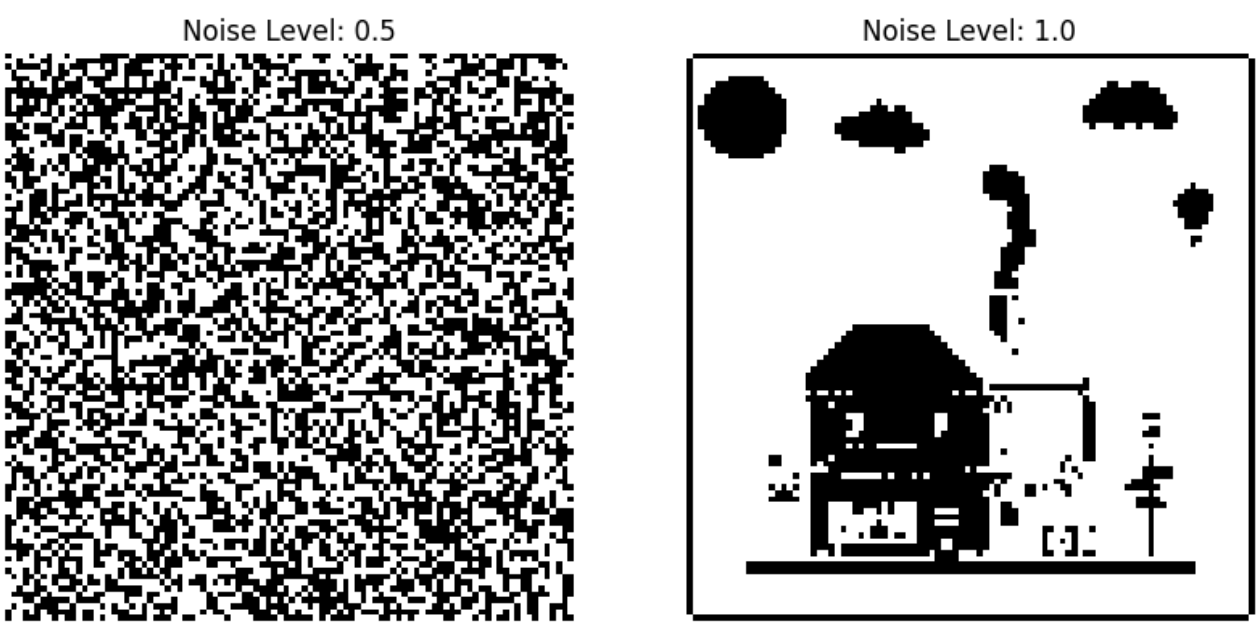
Response to 1.4.a (enter your response below):

If noise, ϵ, was set to 0.5, the image would just be a noisy mess of seemingly random black and white pixels because there would be a 50% probability of the pixel showing up as the opposite color. If noise ϵ were set to 1.0, the image's pixels would be inverted (pixels that should be black would be white and vice versa). This makes sense because a noise level, ϵ, of 0.5 would cause the most information loss and therefore the most distortion.

Images with noise of 0.5 and 1.0 are shown below:

```
# Noise levels 0.5 and 1
noisy_images = []
noise_levels = [0.5, 1.0]
for noise_level in noise_levels:
  noisy_image = gen_noisyimg(img_true, noise_level)
  noisy_images.append((noisy_image, noise_level))

fig, axes = plt.subplots(1, len(noise_levels), figsize=(10, 5))  # Adjust figure size as needed

for ax, (noisy_image, noise_level) in zip(axes, noisy_images):
    ax.imshow(noisy_image, cmap='gray')
    ax.set_title(f'Noise Level: {noise_level}')
```

```
    ax.axis('off')

plt.show()
```



Q1.4.b [5 points]

Now, perform image denoising on images with noise levels 0.05 and 0.3 using $J = 0.1, J = 0.5, J = 1.0$, and $J = 5.0$. 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). In text box below, comment on what you observe and provide a brief explanation on why this might occur.

```
# Solution
J_values = [0.1, 0.5, 1.0, 5.0]
noise_levels = [0.05, 0.3]
step_size = 0.8
max_step = 5

# epsilon = noise_level
for noise_level in noise_levels:
  for J_value in J_values:
    noisy_image = gen_noisyimg(img_true, noise_level)
    y = noisy_image.reshape([img_true.size,])
    beta = 0.5 * np.log((1 - noise_level) / noise_level) # beta as a function of epsilon
    J = J_value
    image_series = test_trajectory(step_size, max_step)
    # title = f"Noise Level: {noise_level}, J: {J_value}"
    plot_series(image_series)
    # plt.suptitle(title)
    # plt.show()
```

Response to 1.4.b (enter your response below):

The noise level 0.3 looks a lot more random than 0.05. this makes sense, because as the noise level gets closer to 0.5 loss and random scatter increase.

Across the different noise levels and J values we see little to no improvement after each step of message passing. This could be because the step size of 0.8 is too large and/or max_steps=5 is too small.

At both noise level 0.05 and noise level 0.3, we see that as J increases, so do clusters of white dots. At noise level 0.05, as J increases white specks of noise in the black background almost completely disappear -- this could be because with a larger J, pixels should be more similar to their neighbors. We also see this at noise level 0.3, where at the smaller J's (0.1, 0.5) the backgrounds of the image just look like noise, whereas at J=5 there's a white blob in the shape of the white house and less noise in the background because, once again, with the larger J, neighboring pixels x_i and x_j are more similar.

Ignore this I just want my whole colab visible when i save as pdf

Random chat gpt generated code to plot a circle so that this gets cut off and not my question 1.4 solutions.

```
# Code to make my response not cut off
import matplotlib.pyplot as plt
import numpy as np

# Define the circle's parameters
radius = 1
center = (0, 0)

# Create an array of angles from 0 to 2pi
theta = np.linspace(0, 2 * np.pi, 100)

# Parametric equations for the circle
x = center[0] + radius * np.cos(theta)
y = center[1] + radius * np.sin(theta)

# Plot the circle
plt.figure(figsize=(6, 6))
plt.plot(x, y)
plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.gca().set_aspect('equal', adjustable='box')  # Make sure the aspect ratio is equal
plt.title("Circle Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()
```