```
In [1]:
%matplotlib inline
%load_ext autoreload
%autoreload 2

import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd

from scipy.special import expit

from aml_utils import test_case_checker, perform_computation, show_test_cases
```

0. Data

Since the MNIST data (http://yann.lecun.com/exdb/mnist/) is stored in a binary format, we would rather have an API handle the loading for us.

Pytorch (https://pytorch.org/) is an Automatic Differentiation library that we may see and use later in the course.

Torchvision (https://pytorch.org/docs/stable/torchvision/index.html?highlight=torchvision#module-torchvision) is an extension library for pytorch that can load many of the famous data sets painlessly.

We already used Torchvision for downloading the MNIST data. It is stored in a numpy array file that we will load easily.

0.1 Loading the Data

```
In [2]:
if os.path.exists('../MeanField-lib/mnist.npz'):
    npzfile = np.load('../MeanField-lib/mnist.npz')
    train_images_raw = npzfile['train_images_raw']
    train_labels = npzfile['train_labels']
    eval_images_raw = npzfile['eval_images_raw']
    eval_labels = npzfile['eval_labels']
else:
    import torchvision
    download_ = not os.path.exists('../MeanField-lib/mnist')
    data_train = torchvision.datasets.MNIST('../MeanField-lib/mnist', train=True, tra
    data_eval = torchvision.datasets.MNIST('../MeanField-lib/mnist', train=False, tra

    train_images_raw = data_train.data.numpy()
    train_labels = data_train.targets.numpy()
    eval_images_raw = data_eval.data.numpy()
    eval_labels = data_eval.targets.numpy()

    np.savez('../MeanField-lib/mnist.npz', train_images_raw=train_images_raw, train_la
             eval_images_raw=eval_images_raw, eval_labels=eval_labels)
```

```
In [3]:
noise_flip_prob = 0.04
```

Task 1

Write the function `get_thresholded_and_noised` that does image thresholding and flipping pixels. More specifically, this functions should exactly apply the following two steps in order:

1. **Thresholding**: First, given the input threshold argument, you must compute a thresholded image array. This array should indicate whether each element of `images_raw` is **greater than or equal to** the `threshold` argument. We will call the result of this step the thresholded image.
2. **Noise Application (i.e., Flipping Pixels)**: After the image was thresholded, you should use the `flip_flags` input argument and flip the pixels with a corresponding `True` entry in `flip_flags`.
   - `flip_flags` mostly consists of `False` entries, which means you should not change their corresponding pixels. Instead, whenever a pixel had a `True` entry in `flip_flags`, that pixel in the thresholded image must get flipped. This way you will obtain the noised image.
3. **Mapping Pixels to -1/+1**: You need to make sure the output image pixels are mapped to -1 and 1 values (as opposed to 0/1 or True/False).

`get_thresholded_and_noised` should take the following arguments:

1. `images_raw`: A numpy array. Do not assume anything about its shape, dtype or range of values. Your function should be careless about these attributes.
2. `threshold`: A scalar value.
3. `flip_flags`: A numpy array with the same shape as `images_raw` and `np.bool` dtype. This array indicates whether each pixel should be flipped or not.

and return the following:

- `mapped_noised_image`: A numpy array with the same shape as `images_raw`. This array's entries should either be -1 or 1.

Notes

numpy.where(condition, x, y): Return elements chosen from x or y depending on condition.
Parameters:

- condition: array_like, or bool: Where True, yield x, otherwise yield y.
- x, y: array_like: Values from which to choose.
  `x, y and condition need to be broadcastable to some shape.`

Returns:

- out: ndarray: An array with elements from x where condition is True, and elements from y elsewhere.

1. indicate whether each element of images_raw is greater than or equal to 'threshold' => thresholded image
   A. Calculate theshed_image array using boolean array check on images_raw.
2. if flip_flags pixel == True that pixel in the thresholded image must get flipped => noised
   A. Using logical_not function you create a not_threshed_image array which is logical not of threshed_image.
   B. Use numpy where function to check condition flip_flags == True and return either value from not_threshed_image or threshed_image.
3. Output image pixels in (-1, 1)

Use numpy.where and numpy.logical_not How to apply numpy.logical_not using the index produced in the numpy.where: -return a truth array. -1 if your image is less than threshhold, +1 if your image is larger than threshhold

1. get a true/false mask based off the threshhold value.
2. use flip_flags to flip this true/false mask. Flip flags is just a matrix of true/false. just use it to index into the matrix you wish to flip values for.
3. get mapped_noised_image by the logic: if the mask entry is True --> then use values of 1, else use values of -1.

4. comparison between images_raw and threshold and save results.

5. use np.logical_not and np where to find where flip flags is true and return result of my result of np.logical_not. result = np.logical_not(step1) result[np.where(flip_flags == True)]. noised = thresholded^flip_flags where, thresholded is the array from step 1
6. convert boolean results to int values.

```
In [4]:
def get_thresholded_and_noised(images_raw, threshold, flip_flags):

    # Beginning of Mo's code

    #1. indicate whether each element of images_raw is greater than or equal to 'thre
        #1. Calculate theshed_image array using boolean array check on images_raw.
    thresholded = images_raw >= threshold
    #2 np.logical_not(x, out, where)
    noised = thresholded
    np.logical_not(thresholded, out=noised, where=flip_flags.astype(bool))
    #3
    mapped_noised_image = np.where(noised == True, 1, -1)

    # End of Mo's code

    assert (np.abs(mapped_noised_image)==1).all()
    return mapped_noised_image.astype(np.int32)
```

```
In [5]:
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

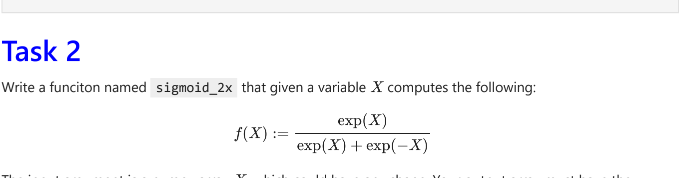
```
In [6]:
# Performing sanity checks on your implementation
def test_thresh_noise(x, seed = 12345, p = noise_flip_prob, threshold = 128):
    np_random = np.random.RandomState(seed=seed)
    flip_flags = (np_random.uniform(0., 1., size=x.shape) < p)
    return get_thresholded_and_noised(x, threshold, flip_flags)

(orig_image, ref_image, test_im, success_thr) = show_test_cases(test_thresh_noise, ta

assert success_thr

# Checking against the pre-computed test database
test_results = test_case_checker(get_thresholded_and_noised, task_id=1)
assert test_results['passed'], test_results['message']
```

```
The reference and solution images are the same to a T! Well done on this test case.
```




```
    Enter nothing to go to the next image
or
    Enter "s" when you are done to recieve the three images.
        **Don't forget to do this before continuing to the next step.**
s
```

```
In [7]:
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

0.2 Applying Thresholding and Noise to Data

```
In [8]:
if perform_computation:
    X_true_grayscale = train_images_raw[:10, :, :]

    np_random = np.random.RandomState(seed=12345)
    flip_flags = flip_flags = (np_random.uniform(0., 1., size=X_true_grayscale.shape)
    initial_pi = np_random.uniform(0, 1, size=X_true_grayscale.shape) # Initial Rando

    X_true = get_thresholded_and_noised(X_true_grayscale, threshold=128, flip_flags=f
    X_noised = get_thresholded_and_noised(X_true_grayscale, threshold=128, flip_flags=
```

Task 2

Write a funciton named `sigmoid_2x` that given a variable X computes the following:

$$f(X) := \frac{\exp(X)}{\exp(X) + \exp(-X)}$$

The input argument is a numpy array X, which could have any shape. Your output array must have the same shape as X.

Important Note: Theoretically, f satisfies the following equations:

$$\lim_{X \to +\infty} f(X) = 1$$

$$\lim_{X \to -\infty} f(X) = 0$$

Your implementation must also work correctly even on these extreme edge cases. In other words, you must satisfy the following tests.

- `sigmoid_2x(np.inf)==1`
- `sigmoid_2x(-np.inf)==0`.

Hint: You may find `scipy.special.expit` useful.

divide numerator and denominator of expit $f(X) := \frac{\exp(X)}{\exp(X)+\exp(-X)}$ by $\exp(X)$ to get $\frac{1}{1+\exp(-2X)}$

```
In [9]:
def sigmoid_2x(X):

    # Beginning of Mo's code
    output = expit(2*X)
    # End of Mo's code

    return output
```

```
In [10]:
# Performing sanity checks on your implementation
assert sigmoid_2x(+np.inf) == 1.
assert sigmoid_2x(-np.inf) == 0.
assert np.array_equal(sigmoid_2x(np.array([0, 1])).round(3), np.array([0.5, 0.881]))

# Checking against the pre-computed test database
test_results = test_case_checker(sigmoid_2x, task_id=2)
assert test_results['passed'], test_results['message']
```

```
In [11]:
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

1. Applying Mean-field Approximation to Boltzman Machine's Variational Inference Problem

Task 3

Write a `boltzman_meanfield` function that applies the mean-field approximation to the Boltzman machine.

Recalling the textbook notation, X_i is the observed value of pixel i, and H_i is the true value of pixel i (before applying noise). For instance, if we have a 3×3 image, the corresponding Boltzman machine looks like this:

```
       X_1        X_2        X_3
      /          /          /
     H_1 ------ H_2 ------ H_3
      |          |          |
      |          |          |
      |          |          |
      | X_4      | X_5      | X_6
      |/         |/         |/
     H_4 ------ H_5 ------ H_6
      |          |          |
      |          |          |
      |          |          |
      | X_7      | X_8      | X_9
      |/         |/         |/
     H_7 ------ H_8 ------ H_9
```

Here, we a adopt a slightly simplified notation from the textbook and define $\mathcal{N}(i)$ to be the neighbors of pixel i (the pixels adjacent to pixel i). For instance, in the above figure, we have $\mathcal{N}(1) = \{2, 4\}$, $\mathcal{N}(2) = \{1, 3, 5\}$, and $\mathcal{N}(5) = \{2, 4, 6, 8\}$.

With this, the process in the textbook can be summarized as follows:

```
1. for iteration = 1, 2, 3, ....,
  2. Pick a random pixel i.
  3. Find pixel i's new parameter as
```

$$\pi_i^{\text{new}} = \frac{\exp(\theta_{ii}^{(2)} X_i + \sum_{j \in \mathcal{N}(i)} \theta_{ij}^{(1)} (2\pi_j - 1))}{\exp(\theta_{ii}^{(2)} X_i + \sum_{j \in \mathcal{N}(i)} \theta_{ij}^{(1)} (2\pi_j - 1)) + \exp(-\theta_{ii}^{(2)} X_i - \sum_{j \in \mathcal{N}(i)} \theta_{ij}^{(1)} (2\pi_j - 1))}.$$

```
  4. Replace the existing parameter for pixel i with the new one.
```

$$\pi_i \leftarrow \pi_i^{\text{new}}$$

Since our computational resources are extremely vectorized, we will make the following minor algorithmic modification and ask you to implement the following instead:

```
1. for iteration = 1, 2, 3, ....,
  2. for each pixels i:
  3. Find pixel i's new parameter, but do not update the original parameter
yet.
```

$$\pi_i^{\text{new}} = \frac{\exp(\theta_{ii}^{(2)} X_i + \sum_{j \in \mathcal{N}(i)} \theta_{ij}^{(1)} (2\pi_j - 1))}{\exp(\theta_{ii}^{(2)} X_i + \sum_{j \in \mathcal{N}(i)} \theta_{ij}^{(1)} (2\pi_j - 1)) + \exp(-\theta_{ii}^{(2)} X_i - \sum_{j \in \mathcal{N}(i)} \theta_{ij}^{(1)} (2\pi_j - 1))}.$$

```
  4. Once you have computed all the new parameters, update all of them at the
same time:
```

$$\pi \leftarrow \pi^{\text{new}}$$

We assume that the parameters $\theta_{ii}^{(2)}$ have the same value for all i and denote their common value by scalar `theta_X`. Moreover, we assume that the parameters $\theta_{ij}^{(1)}$ have the same value for all i, j and denote their common value by scalar `theta_pi`.

The `boltzman_meanfield` function must take the following input arguments:

1. `images`: A numpy array with the shape `(N,height,width)`, where
   - `N` is the number of samples and could be anything,
   - `height` is each individual image's height in pixels (i.e., number of rows in each image),
   - and `width` is each individual image's width in pixels (i.e., number of columns in each image).
     - Do not assume anything about `images`'s dtype or the number of samples or the `height` or the `width`.
     - The entries of `images` are either -1 or 1.
2. `initial_pi`: A numpy array with the same shape as `images` (i.e. `(N,height,width)`). This variable is corresponding to the initial value of π in the textbook analysis and above equations. Note that for each of the N images, we have a different π variable.

3. `theta_X`: A scalar with a default value of `0.5*np.log(1/noise_flip_prob-1)`. This variable represents $\theta_{ii}^{(2)}$ in the above update equation.

4. `theta_pi`: A scalar with a default value of 2. This variable represents $\theta_{ij}^{(1)}$ in the above update equation.

5. `iterations`: A scalar with a default value of 100. This variable denotes the number of update iterations to perform.

The `boltzman_meanfield` function must return the final π variable as a numpy array called `pi`, and should contain values that are between 0 and 1.

Hint: You may find the `sigmoid_2x` function, that you implemented earlier, useful.

Hint: If you want to find the summation of neighboring elements for all of a 2-dimensional matrix, there is an easy and efficient way using matrix operations. You can initialize a zero matrix, and then add four shifted versions (i.e., left-, right-, up-, and down-shifted versions) of the original matrix to it. You will have to be careful in the assignment and selection indices, since you will have to drop one row/column for each shifted version of the matrix.

- **Important Note**: Do **not** use `np.roll` if you're taking this approach.

Important Note: When evaluating the neighborhood sum experssions (i.e., terms with $\sum_{j \in \mathcal{N}(i)}$), make sure that you do not inadvertently include a "ghost" pixel in $\mathcal{N}(i)$. For instance, make sure you're only using `H_5`, `H_7`, `H_9`, and `X_8` when computing an update for `H_8`. That is, only left-, right-, and down-shifted pixels should be contributing to `H_8`'s neighourhood sums. If you mistakenly add an up-shifted pixel to `H_8`'s neighourhood sums (whether it's a copy of `H_8` or a no-ink/zero pixel), you are effectively imposing an extra neighborhood edge between `H_8` and a "ghost" pixel below it; notice that our boltzman machine doesn't have a neighborhood edge between `H_8` and anything below it, therefore, neither `H_8` nor an extra non-inky pixel should be participating in `H_8`'s neighborhood sums and update.

- Missing this point can cause an initial mismatche in the edge pixel updates, which will be disseminated through iterative updates to other pixels.

Notes

1. Create a separate function which uses matrix operations to calculate sum of neighboring elements of pi
2. For each iteration:

- For each pi
  - get neighboring value sum array using step 1 function
  - Calculate the term for updating parameters: $\theta_{ii}^{(2)} X_i + \sum_{j \in \mathcal{N}(i)} \theta_{ij}^{(1)} (2\pi_j - 1)$

1. Pass above term through sigmoid_2x function to get updated value for π

```
In [12]:
def boltzman_meanfield(images, initial_pi, theta_X=0.5*np.log(1/noise_flip_prob-1), th
    if len(images.shape)==2:
        # In case a 2d image was given as input, we'll add a dummy dimension to be co
        X = images.reshape(1,*images.shape)
    else:
        # Otherwise, we'll just work with what's given
        X = images

    pi = initial_pi

    # Beginning of Mo's code

    #1 Create a separate function which uses matrix operations to calculate sum of ne
    def shift(pi, right, up):
        e = np.empty_like(pi)
        if right >= 0:
            e[:, :right, :] = 0
            e[:, right:, :] = pi[:, :-right, :]
        else:
            e[:, right:, :] = 0
            e[:, :right, :] = pi[:, -right:, :]
        if up >= 0:
            e[:, :, :up] = 0
            e[:, :, up:] = pi[:, :, :-up]
        else:
            e[:, :, up:] = 0
            e[:, :, :up] = pi[:, :, -up:]
        return e

    def SumNi(pi):
        #Ni = shift(pi, 1, 0) + shift(pi, -1, 0) + shift(pi, 0, 1) + shift(pi, 0, -1)
        #left[left!=0]=2*left[left!=0]-1 ##from campuswire???
        #do I multiply all pi here by theta_pi?
        r = shift(pi, 1, 0)[shift(pi, 1, 0) !=0]   = 2 * shift(pi, 1, 0)[shift(pi, 1,
        l = shift(pi, -1, 0)[shift(pi, -1, 0) !=0] = 2 * shift(pi, -1, 0)[shift(pi, -
        d = shift(pi, 0, -1)[shift(pi, 0, -1) !=0] = 2 * shift(pi, 0, -1)[shift(pi, 0,
        u = shift(pi, 0, 1)[shift(pi, 0, 1) !=0]   = 2 * shift(pi, 0, 1)[shift(pi, 0,
        Ni = r + l + d + u
        return Ni

    #do I replace pi with theta_pi * (2 * pi - 1)
    for i in range(iterations):
        #2   - get neighboring value sum array using step 1 function
        #    - Calculate the term for updating parameters
        #cmn_term = theta_X * X + SumNi(pi) * theta_pi * (2 * pi - 1)

        cmn_term = theta_X * X + SumNi(theta_pi * (2 * pi - 1))

        #3 Pass above term through sigmoid_2x function to get updated value for pi
        pi = sigmoid_2x(cmn_term)

    # End of Mo's code

    return pi.reshape(*images.shape)
```

```
In [13]:
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```
In [14]:
# Performing sanity checks on your implementation
def test_boltzman(x, seed = 12345, theta_X=0.5*np.log(1/noise_flip_prob-1), theta_pi=2
    np_random = np.random.RandomState(seed=seed)
    initial_pi = np_random.uniform(0,1, size=x.shape)
    return boltzman_meanfield(x, initial_pi, theta_X=theta_X,
                              theta_pi=theta_pi, iterations=iterations)

(orig_image, ref_image, test_im, success_is_row_inky) = show_test_cases(test_boltzman,

assert success_is_row_inky

# Checking against the pre-computed test database
test_results = test_case_checker(boltzman_meanfield, task_id=3)
assert test_results['passed'], test_results['message']
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [14], in <module>
      4     initial_pi = np_random.uniform(0,1, size=x.shape)
      5     return boltzman_meanfield(x, initial_pi, theta_X=theta_X,
      6                               theta_pi=theta_pi, iterations=iterations)
----> 8 (orig_image, ref_image, test_im, success_is_row_inky) = show_test_cases(test_b
oltzman, task_id='3_V')
     10 assert success_is_row_inky
     12 # Checking against the pre-computed test database

File ~/work/release/MeanField-lib/aml_utils.py:451, in show_test_cases(test_func, task
_id)
    449 orig_images = npz_file['raw_images']
    450 ref_images = npz_file['ref_images']
--> 451 test_images = test_func(orig_images)
    453 rtol = 1e-05
    454 if ('meanfield' in this_dir.lower()) and ('3' in str(task_id)):

Input In [14], in test_boltzman(x, seed, theta_X, theta_pi, iterations)
      3 np_random = np.random.RandomState(seed=seed)
      4 initial_pi = np_random.uniform(0,1, size=x.shape)
----> 5 return boltzman_meanfield(x, initial_pi, theta_X=theta_X,
      6                           theta_pi=theta_pi, iterations=iterations)

Input In [12], in boltzman_meanfield(images, initial_pi, theta_X, theta_pi, iteration
s)
     43 #do I replace pi with theta_pi * (2 * pi - 1)
     44 for i in range(iterations):
     45     #2   - get neighboring value sum array using step 1 function
     46     #    - Calculate the term for updating parameters
     47     #cmn_term = theta_X * X + SumNi(pi) * theta_pi * (2 * pi - 1)
---> 49     cmn_term = theta_X * X + SumNi(theta_pi * (2 * pi - 1))
     51     #3 Pass above term through sigmoid_2x function to get updated value for pi
     52     pi = sigmoid_2x(cmn_term)

Input In [12], in boltzman_meanfield.<locals>.SumNi(pi)
     32 def SumNi(pi):
     33     #Ni = shift(pi, 1, 0) + shift(pi, -1, 0) + shift(pi, 0, 1) + shift(pi, 0,
 -1) #original
     34     #left[left!=0]=2*left[left!=0]-1 ##from campuswire???
     35     #do I multiply all pi here by theta_pi?
---> 36     r = shift(pi, 1, 0)[shift(pi, 1, 0) !=0]   = 2 * shift(pi, 1, 0)[shift(pi,
 1, 0) != 0] - 1
     37     l = shift(pi, -1, 0)[shift(pi, -1, 0) !=0] = 2 * shift(pi, -1, 0)[shift(p
i, -1, 0) != 0] - 1
     38     d = shift(pi, 0, -1)[shift(pi, 0, -1) !=0] = 2 * shift(pi, 0, -1)[shift(p
i, 0, -1) != 0] - 1

Input In [12], in boltzman_meanfield.<locals>.shift(pi, right, up)
     22 if up >= 0:
     23     e[:, :, :up] = 0
---> 24     e[:, :, up:] = pi[:, :, :-up]
     25 else:
     26     e[:, :, up:] = 0

ValueError: could not broadcast input array from shape (100,28,0) into shape (100,28,2
8)
```

```
In [ ]:
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

2. Tuning the Boltzman Machine's Hyper-Parameters

Now, with the `boltzman_meanfield` function that you implemented above, here see the effect of changing hyper parameters `theta_X` and `theta_pi` which were defined in Task 3.

- We set `theta_X` to be `0.5*np.log(1/noise_flip_prob-1)` where `noise_flip_prob` was the probability of flipping each pixel. Try to think why this is a reasonable choice. (This is also related to one of the questions in the follow-up quiz).
- We try different values for `theta_pi`.

For each value of `theta_pi`, we the apply the denoising and compare the denoised images to the original ones. We adopt several statistical measures to compare original and denoised images and to finally decide which value of `theta_pi` is better. Remember that during the noising process, we chose some pixels and decide to flip them, and during the denoising process we essentially try to detect such pixels. Let `P` be the total number of pixels that we flip during the noise adding process, and `N` be the total number of pixels that we do not flip during the noise adding process. We can define:

- True Positive (`TP`). Defined to be the total number of pixels that are flipped during the noise adding process, and we successfully detect them during the denoising process.
- True Positive Rate (`TPR`). Other names: sensitivity, recall. Defined to be the ratio `TP / P`.
- False Positive (`FP`). Defined to be the number of pixels that were detected as being noisy during the denosing process, but were not really noisy.
- False Positive Rate (`FPR`). Other name: fall-out. Defined to be the ratio `FP/N`.
- Positive Predictive Value (`PPV`). Other name: precision. Defined to be the ratio `TP / (TP + FP)`.
- `F1` score. Defined to be the harmonic mean of precision (`PPV`) and recall (`TPR`), or equivalently `2 TP / (2 TP + FP + FN)`.

Since we fix `theta_X` in this section and evaluate different values of `theta_pi`, in the plots, `theta` refers to `theta_pi`.

```
In [ ]:
def get_tpr(preds, true_labels):
    TP = (preds * (preds == true_labels)).sum()
    P = true_labels.sum()
    if P==0:
        TPR = 1.
    else:
        TPR = TP / P

    return TPR

def get_fpr(preds, true_labels):
    FP = (preds * (preds != true_labels)).sum()
    N = (1-true_labels).sum()
    if N==0:
        FPR=1
    else:
        FPR = FP / N
    return FPR

def get_ppv(preds, true_labels):
    TP = (preds * (preds == true_labels)).sum()
    FP = (preds * (preds != true_labels)).sum()
    if (TP + FP) == 0:
        PPV = 1
    else:
        PPV = TP / (TP + FP)
    return PPV

def get_f1(preds, true_labels):
    TP = (preds * (preds == true_labels)).sum()
    FP = (preds * (preds != true_labels)).sum()
    FN = ((1-preds) * (preds != true_labels)).sum()
    if (2 * TP + FP + FN) == 0:
        F1 = 1
    else:
        F1 = (2 * TP) / (2 * TP + FP + FN)
    return F1
```

```
In [ ]:
if perform_computation:
    all_theta = np.arange(0, 10, 0.2).tolist() + np.arange(10, 100, 5).tolist()

    tpr_list, fpr_list, ppv_list, f1_list = [], [], [], []

    for theta in all_theta:
        meanfield_pi = boltzman_meanfield(X_noised, initial_pi, theta_X=0.5*np.log(1/n
        X_denoised = 2 * (meanfield_pi > 0.5) - 1

        predicted_noise_pixels = (X_denoised != X_noised)
        tpr = get_tpr(predicted_noise_pixels, flip_flags)
        fpr = get_fpr(predicted_noise_pixels, flip_flags)
        ppv = get_ppv(predicted_noise_pixels, flip_flags)
        f1 = get_f1(predicted_noise_pixels, flip_flags)

        tpr_list.append(tpr)
        fpr_list.append(fpr)
        ppv_list.append(ppv)
        f1_list.append(f1)
```

```
In [ ]:
if perform_computation:
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12,4), dpi=90)

    ax=axes[0]
    ax.plot(all_theta, tpr_list)
    ax.set_xlabel('Theta')
    ax.set_ylabel('True Positive Rate')
    ax.set_title('True Positive Rate Vs. Theta')
    ax.set_xscale('log')

    ax=axes[1]
    ax.plot(all_theta, fpr_list)
    ax.set_xlabel('Theta')
    ax.set_ylabel('False Positive Rate')
    ax.set_title('False Positive Rate Vs. Theta')
    ax.set_xscale('log')
```

```
In [ ]:
if perform_computation:
    fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(12,3), dpi=90)

    ax=axes[0]
    ax.plot(fpr_list, tpr_list)
    ax.set_xlabel('False Positive Rate')
    ax.set_ylabel('True Positive Rate')
    ax.set_title('ROC Curve')
    ax.set_xlim(-0.05, 1.05)
    ax.set_ylim(-0.05, 1.05)
    ax.plot(np.arange(-0.05, 1.05, 0.01), np.arange(-0.05, 1.05, 0.01), ls='--', c='bl

    ax=axes[1]
    ax.plot(all_theta, f1_list)
    ax.set_xlabel('Theta')
    ax.set_ylabel('F1-statistic')
    ax.set_title('F1-score Vs. Theta')
    ax.set_xscale('log')

    ax=axes[2]
    ax.plot(tpr_list, ppv_list)
    ax.set_xlabel('Recall')
    ax.set_ylabel('Precision')
    ax.set_title('Precision Vs. Recall')
    ax.set_xlim(-0.05, 1.05)
    ax.set_ylim(-0.05, 1.05)
    ax.plot(np.arange(-0.05, 1.05, 0.01), 1-np.arange(-0.05, 1.05, 0.01), ls='--', c=
    None
```

```
In [ ]:
if perform_computation:
    best_theta = all_theta[np.argmax(f1_list)]
    print(f'Best theta w.r.t. the F-score is {best_theta}')
```

Now let's try the tuned hyper-parameters, and verify whether it visually improved the Boltzman machine.

```
In [ ]:
# This cell is left empty as a seperator. You can leave this cell as it is, and you si
```

```
In [ ]:
if perform_computation:
    def test_boltzman(x, seed = 12345, theta_X=0.5*np.log(1/noise_flip_prob-1), theta_
        np_random = np.random.RandomState(seed=seed)
        initial_pi = np_random.uniform(0,1, size=x.shape)
        return boltzman_meanfield(x, initial_pi, theta_X=theta_X,
                                  theta_pi=theta_pi, iterations=iterations) >  0.5

    (orig_image, ref_image, test_im, success_is_row_inky) = show_test_cases(test_boltz
```

```
In [ ]:
```