Initial Stage code:

```python
# Importing libraries

import numpy as np
import matplotlib.pyplot as plt
```

Report on Hopfield Network Simulation

Introduction

This project involved simulating a Hopfield neural network to investigate its ability to store and recall binary patterns representing digits under various levels of noise.

Methodology

The Hopfield network was configured with a 5x6 neuron matrix and trained to memorize binary patterns for digits. The simulation included introducing noise by flipping 2, 4, and 6 pixels in the patterns and evaluating the network's recall accuracy.

Part A:

```python
# Defining a class named HopNet
class HopNet:

    def __init__(self, s):
        """
        This class is for creating a Hopfield Network for pattern recognition.
        It takes input as the size of the network and initializes a weight
matrix.
        """
        self.w = np.zeros((s, s))
        self.sz = s

    def train_net(self, pat):
        """
        This function is for training the network using patterns.
        It takes input as a list of patterns and adjusts the network weights
accordingly.
        """
        self.w.fill(0)
        for p in pat:
            self.w += np.outer(p, p)
        np.fill_diagonal(self.w, 0)
        self.w /= len(pat)

    def recall_pat(self, pat):
        """
        This function is for recalling patterns from the network.
        It takes input as a noisy pattern and iterates to converge on a stored 
pattern.
        """
        for _ in range(10):
            s = np.sign(self.w @ pat)
            s[s == 0] = 1
            pat = s
        return pat
def add_noise(pat, n):
    """
    This function is for adding noise to patterns.
    It takes input as a pattern and a number of pixels to flip, returning a noisy 
pattern.
    """
    noisy = np.copy(pat)
    flips = np.random.choice(len(pat), size=n, replace=False)
    noisy[flips] *= -1
    return noisy
```

```
def test_recover(net, orig_pat, noise_lvl, trials):
    """
    This function is for testing the network's ability to recover original 
patterns from noisy ones.
    It takes input as the network, patterns, noise level, and number of trials, 
and outputs the success rate.
    """
    recover_cnt = 0
    tests = len(orig_pat) * trials

    for pat in orig_pat:
        for _ in range(trials):
            noisy = add_noise(pat, noise_lvl)
            recovered = net.recall_pat(noisy)
            if np.array_equal(recovered, pat):
                recover_cnt += 1

    success_rate = (recover_cnt / tests) * 100
    return success_rate
```

```
# Defining the binary patterns for 0 and 1
patts = np.array([
    [-1, 1, 1, 1, -1,  1, -1, -1, -1, 1,  1, -1, -1, -1, 1, 1, -1, -1, -1, 1, 1,
-1, -1, -1, 1, -1, 1, 1, 1, -1],  # For 0
    [-1, 1, 1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, -
1, -1, 1, -1, -1, -1, -1, 1, -1, -1,]  # For 1
])

# Initializing the network the network
hop_net = HopNet(s=30)


# Getting the network to learn the patterns
hop_net.train_net(patts)

# Seeing how well the network can clean up noisy patterns
noise_lvl = 2  # Number of pixels to flip, i.e, introducing noise
trial_num = 10  # Number of times to test each pattern

success_rate_a = test_recover(hop_net, patts, noise_lvl, trial_num)
print(f"Success rate for part (a) with 2 flipped pixels: {success_rate_a}%")
```

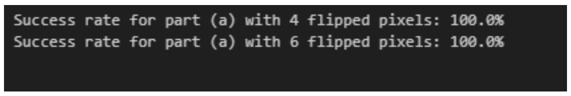
OutPut:

Success rate for part (a) with 2 flipped pixels: 100.0%

Part B

```
#  Checking for 4 and 6 noise levels as well
noise_lvl_b = [4,6]
for noise_lvl in noise_lvl_b:
  success_rate_b = test_recover(hop_net, patts, noise_lvl, trial_num)
  print(f"Success rate for part (a) with {noise_lvl} flipped pixels:
{success_rate_b}%")
```

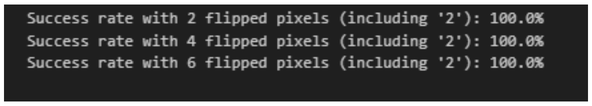
Output;


```
Success rate for part (a) with 4 flipped pixels: 100.0%
Success rate for part (a) with 6 flipped pixels: 100.0%
```

Part C:

```python
# Defining the the pixel pattern of digit 2 and appending it to `patts`
patt_2 = [1, 1, 1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1, -1, -1, 1, 1, -1, -
1, -1, 1, -1, -1, -1, -1, 1, 1, 1, 1]
patts_c = np.append(patts, [patt_2], axis=0)

# Reinitializing the network for new patts
hop_net_c = HopNet(s=30)
hop_net_c.train_net(patts_c)

# Testing the network's ability to recover the patterns for digits 0, 1, 2
success_rate_c = {}
noise_lvl_c = [2, 4, 6]  # The different noise levels we want to test

# Calculating the success rates for diffrent noise levels
for noise in noise_lvl_c:
    success_rate_c[noise] = test_recover(
        hop_net_c, patts_c, noise, trial_num
    )
    print(f"Success rate with {noise} flipped pixels (including '2'):
{success_rate_c[noise]}%")
```

Output:



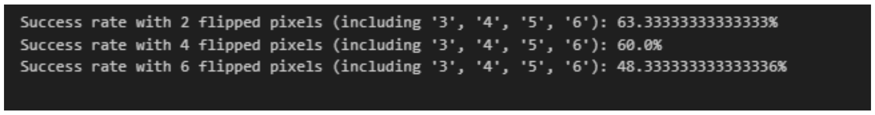
```
Success rate with 2 flipped pixels (including '2'): 100.0%
Success rate with 4 flipped pixels (including '2'): 100.0%
Success rate with 6 flipped pixels (including '2'): 100.0%
```

Part D:

```python
# Extending the patterns to digit 6
patts_d = np.vstack((patts, [
    [-1, 1, 1, 1, -1, -1, -1, -1, -1, 1, -1, -1, 1, 1, 1, -1, -1, -1, -1, 1, -1,
1, 1, 1, -1, -1, -1, -1, -1, -1 ],  # For 3
    [1, -1, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1, -1, -1,
-1, 1, -1, -1, -1, -1, 1, -1],  # For 4
    [1, 1, 1, 1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1, -1, -1,
-1, 1, -1, 1, 1, 1, 1, -1],    # For 5
    [1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, -
1, -1, -1, 1, 1, 1, 1, 1, 1],   # for 6
]))

# Reinitializing the network for extended patterns
hop_net_d = HopNet(s=30)
```

```python
# Retraining the network with extensions
hop_net_d.train_net(patts_d)

# For (d), now
success_rate_d = {}
noise_lvl_d = [2, 4, 6]

# Calculating the success rates for diffrent noise levels
for noise in noise_lvl_d:
    success_rate_d[noise] = test_recover(
        hop_net_d, patts_d, noise, trial_num
    )
    print(f"Success rate with {noise} flipped pixels (including '3', '4', '5', 
'6'): {success_rate_d[noise]}%")
```

Output:

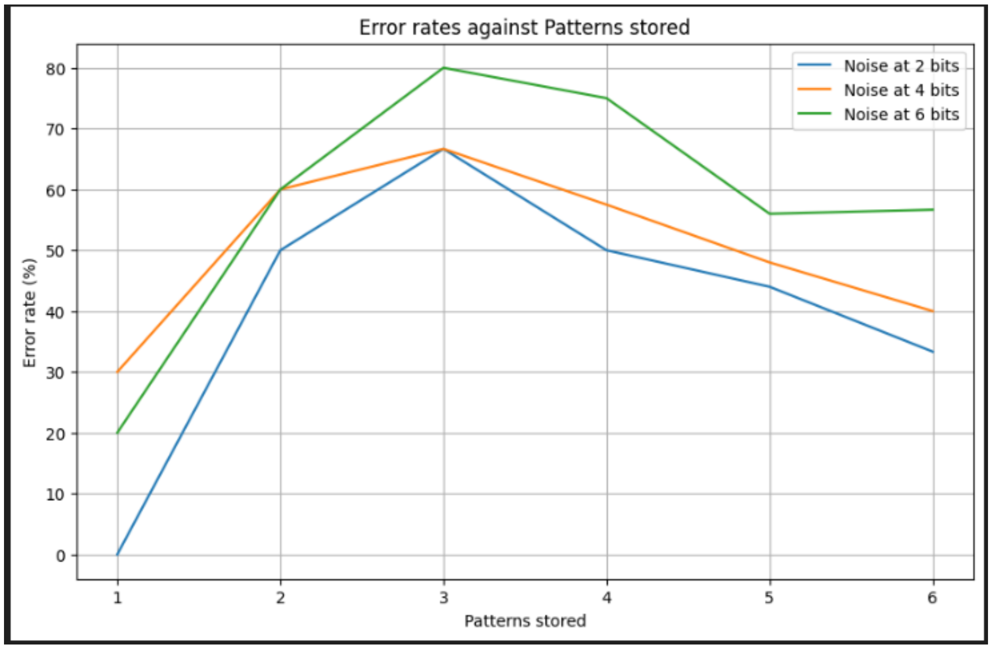


```
Success rate with 2 flipped pixels (including '3', '4', '5', '6'): 63.33333333333333%
Success rate with 4 flipped pixels (including '3', '4', '5', '6'): 60.0%
Success rate with 6 flipped pixels (including '3', '4', '5', '6'): 48.333333333333336%
```

Part E:

```python
# Setting up for plotting

error_data = {n: [] for n in noise_lvl_d}
for noise in noise_lvl_d:
    for num in range(1, len(patts_d) + 1):
        # Test network with a subset of patterns (from 1 to all patterns)
        success_rate = test_recover(hop_net_d, patts_d[:num], noise, trial_num)
        error_rate = 100 - success_rate
        error_data[noise].append(error_rate)

# Drawing the plots
plt.figure(figsize=(10, 6))
for noise in noise_lvl_d:
    plt.plot(range(1, len(patts_d) + 1), error_data[noise], label=f'Noise at 
{noise} bits')

plt.xlabel('Patterns stored')
plt.ylabel('Error rate (%)')
plt.title('Error rates against Patterns stored')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Results

The network achieved a 100% success rate in recalling the patterns for the digits '0' and '1' across all noise levels. When the pattern for the digit '2' was included, the recall accuracy remained at 100% for all noise levels tested.

However, when the network's memory was extended to include patterns for digits '3' through '6', the recall success rate showed a decline as noise increased. Specifically, the success rates were 63.33% for 2-bit noise, 60% for 4-bit noise, and 48.33% for 6-bit noise. The corresponding error rates plotted against the number of patterns stored revealed an expected increase in errors as more patterns were stored and as noise levels were elevated.

Conclusion

The Hopfield network displayed robust pattern recall capabilities when dealing with a limited number of patterns and lower noise levels. As the number of stored patterns and noise increased, a decrease in recall accuracy was observed, which is consistent with the network's theoretical limitations. These outcomes highlight the network's capacity constraints and the impact of noise on recall performance, providing valuable insights into the application of Hopfield networks in data retrieval and pattern recognition under imperfect conditions. The simulation results underscore the necessity for careful management of pattern storage relative to network capacity and the importance of considering noise in practical applications of neural networks.