Assignment #3

November 19, 2017

COMP 4107 Fall 2017
Basim Ramadhan 100 901 646
Christian Abbott 100 863 049

1 Question 1

1.1 Running Our Code

We have provided an easy-to-use Makefile to help you run our program:

```
make prepare-venv
./env/bin/python hopfieldnet.py 2
```

Otherwise, if you have matplotlib, numpy, and scikit-learn installed already:

```
# Syntax
python3 hopfieldnet.py num_training_patterns
# Example
python3 hopfieldnet.py 2
```

Running the program will do the following:

1. Load the MNIST dataset using scikit-learn then subsample to only include 1's and 5's.
2. Pick some random MNIST images to train with; quantity of images is user-defined using the **num_training_patterns** command-line parameter.
3. Initialize a Hopfield network.
4. Train the network using Storkey's learning rule.
5. Degrade each training image with 20% noise *(flip 20% of the image's bits)*.
6. Test whether the network can restore the degraded images satisfactorily.
7. Print out the network's recovery accuracy.

After the program performs the above, it will display the following visualizations:

1. The network's weights.
2. The network's state *(the sum of each neuron's own weights)*.
3. A comparison between each original image, its degraded version, and its recovered version.

1.2 Accuracy of Pattern Recovery / Classification

We experimented with the Hebb and Storkey learning rules for Hopfield networks storing between 1 and 20 images. For each number of images stored in the network, we repeated the experiment 20 times. In other words, our experimentation was the following:

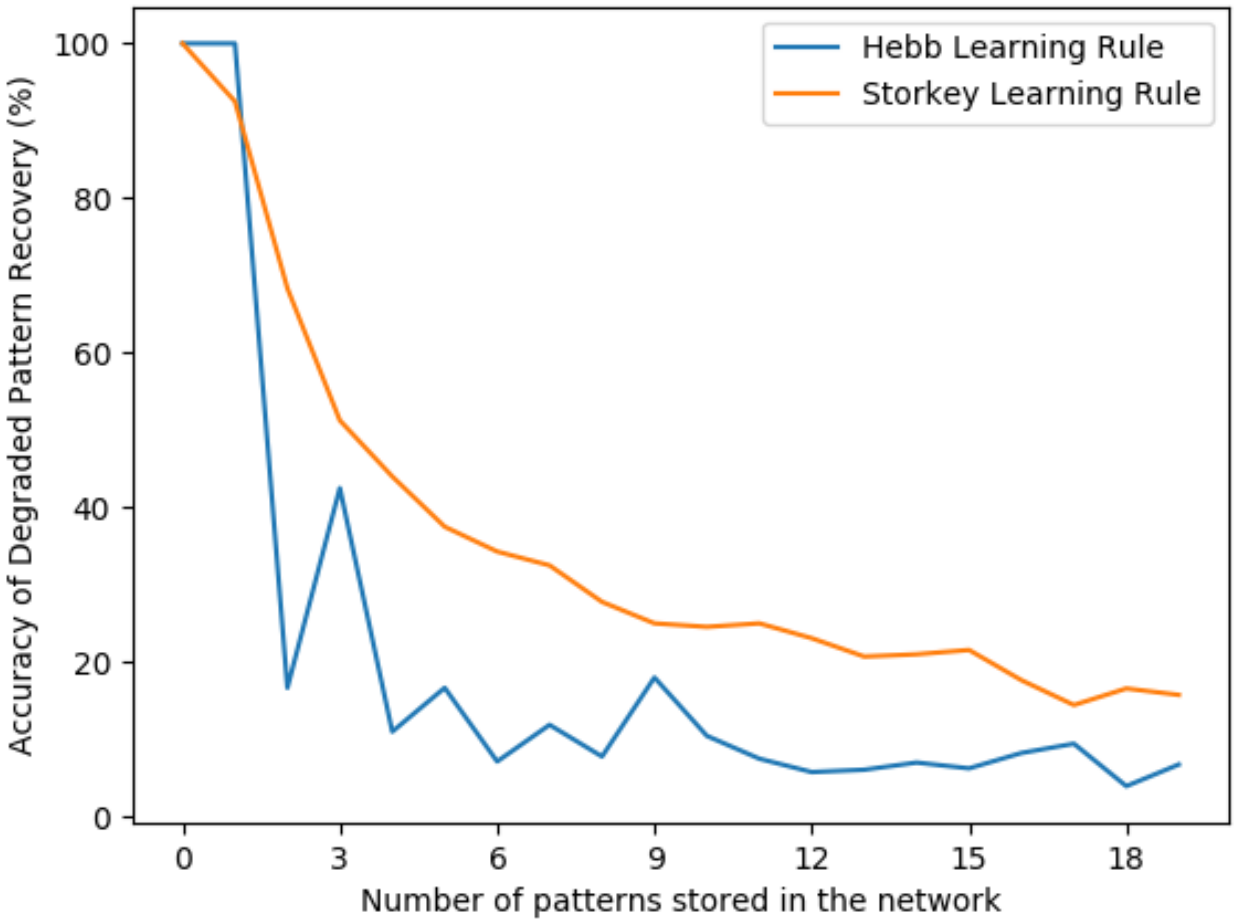
- Train on 1 image with Hebb's rule and test the network's recovery accuracy, 20 times.
- Train on 2 images with Hebb's rule and test the network's recovery accuracy, 20 times.
- . . .
- Train on 20 images with Hebb's rule and test the network's recovery accuracy, 20 times.
- Train on 1 image with Storkey's rule and test the network's recovery accuracy, 20 times.
- Train on 2 images with Storkey's rule and test the network's recovery accuracy, 20 times.
- . . .
- Train on 20 images with Storkey's rule and test the network's recovery accuracy, 20 times.

A few notes regarding our experiments:

- to degrade our images, we applied 20% noise, which means that a random 20% of the pattern's bits were flipped
- the threshold for a recovery being considered a success was the following: the L2-norm between the original MNIST image and the recovered image must be less than 10. We found this threshold only lets very good recoveries pass

These experiments yields the accuracy values in the code and chart below.

1.2.1 Accuracy Chart



Pattern Recovery Accuracy

1.2.2 Accuracy Chart Code

Check out the accuracy values inside the code used to produce the previous chart:

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

hebb_accuracy = [
    100.00, 100.00, 16.67, 42.50, 11.00,
    16.67, 7.14, 11.88, 7.78, 18.00,
    10.45, 7.50, 5.77, 6.07, 7.00,
    6.25, 8.24, 9.44, 3.95, 6.75
]
storkey_accuracy = [
    100.00, 92.50, 68.33, 51.25, 44.00,
    37.50, 34.29, 32.50, 27.78, 25.00,
    24.55, 25.00, 23.08, 20.71, 21.00,
    21.56, 17.65, 14.44, 16.58, 15.75
]

fig = plt.figure()
ax = fig.gca()

plt.plot(hebb_accuracy)
plt.plot(storkey_accuracy)

ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.xlabel("Number of patterns stored in the network")
plt.ylabel("Accuracy of Degraded Pattern Recovery (%)")
plt.legend(['Hebb Learning Rule', 'Storkey Learning Rule'], loc='upper right')

plt.show()
```

1.2.3 Analysis

Based on the data above—in addition to our experiences—we have the following conclusions:

- In general, training with Storkey's rule produces more accurate networks.
- Hebb's rule surprisingly works better for 2-image networks than Storkey's rule.
- Training with Hebb's rule (<1 second) is significantly faster than with Storkey's rule (~5 seconds).

1.3 Implementation

1.3.1 Learning Rule Implementations

For both learning rules, our initial implementations were simple and followed their respective definitions closely. These implementations were slow because they used nested for-loops with multiplications in each iteration. We then implemented optimized versions of the learning rules which used matrix operations instead. In the following code blocks we show both unoptimized and optimized implementations.

1.3.2 Hebb's Learning Rule

```
def train_hebbian(self, patterns):
    """Train the Hopfield network using the Hebbian learning rule (1949).
    https://en.wikipedia.org/wiki/Hopfield_network
    """
    for p in patterns:
        a = p.reshape((self.shape, 1))
        b = a.T
        self.weights += np.dot(a, b)
    self.weights -= (np.identity(patterns[0].size) * patterns.shape[0])
    return self.weights

def train_hebbian_unoptimized(self, patterns):
    """Inefficient version of the train_hebbian function.
    Performs individual multiplications instead of efficient matrix operations."""
    n = self.shape
    for i, j in itertools.product(range(n), range(n)):
        self.weights[i][j] = sum([p[i] * p[j] for p in patterns]) / n
    return self.weights
```

1.3.3 Storkey's Learning Rule

```
def train_storkey(self, patterns):
    """Train the Hopfield network using the Storkey learning rule (1997).
    https://en.wikipedia.org/wiki/Hopfield_network#The_Storkey_learning_rule
    """
    n = self.shape
    for p in patterns:
        for i, j in itertools.product(range(n), range(n)):
            wt = self.weights
            w = wt[i][j]
            x = p[i] * p[j]
            y = p[i] * (np.dot(wt[j], p) - wt[j][i] * p[i] - wt[j][j] * p[j])
            z = p[j] * (np.dot(wt[i], p) - wt[i][i] * p[i] - wt[i][j] * p[j])
            wt[i][j] = w + ((x - y - z) / n)


def train_storkey_unoptimized(self, patterns):
    """Inefficient version of the train_storkey function.
    Performs individual multiplications instead of efficient matrix operations."""
    n = self.shape
    for p in patterns:
        for i, j in itertools.product(range(n), range(n)):
```

```
            w = self.weights[i][j]
            x = p[i] * p[j] / n
            y = p[i] * sum([w[j][k] * p[k] for k in range(n) if k not in [i, j]]) / n
            z = p[j] * sum([w[i][k] * p[k] for k in range(n) if k not in [i, j]]) / n
            w[i][j] = w + x - y - z
```

1.3.4 Activation

Our activation function follows the definition on the Wikipedia page.

```
def activate(self, i):
    """Determine whether the given neuron should be active or inactive.
    https://en.wikipedia.org/wiki/Hopfield_network#Updating"""
    weight_sum = np.dot(self.weights[i], self.state)
    self.state[i] = 1 if weight_sum > self.thresholds[i] else -1

def activate_unoptimized(self, i):
    """Inefficient version of activate."""
    num_neurons = self.shape
    weight_sum = 0.0
    for j in range(num_neurons):
        weight_sum += self.weights[i][j] * self.state[j]
    self.state[i] = 1 if weight_sum > self.thresholds[i] else -1
```

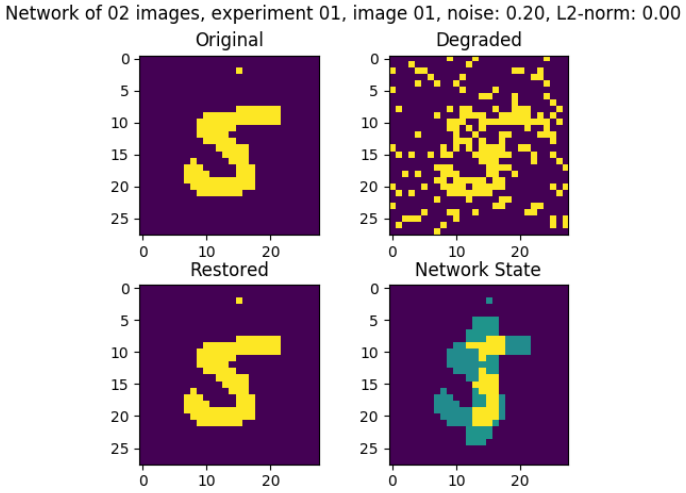
1.3.5 Recovery

We took our own approach to recovering degraded patterns. At each iteration, we call activate on each neuron in random order. If no state changes take place during a single iteration, then the network state is stable the image has (hopefully) been recovered. If the network is not stable yet, we repeat for another iteration.

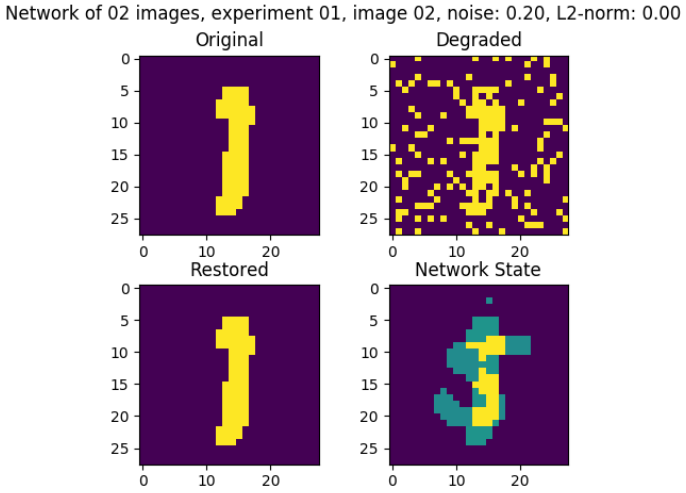
```
def restore(self, degraded_pattern):
    """Recover the original pattern of the degraded input pattern."""
    self.state = np.copy(degraded_pattern)
    num_neurons = self.shape

    # During each iteration: ensure each neuron is activated at least once
    iterations = 0
    while iterations < 10:
        changed = False
        neurons = list(range(num_neurons))
        random.shuffle(neurons)
        while neurons:
            neuron = neurons.pop()
            old_state = self.state[neuron]
            self.activate(neuron)
            new_state = self.state[neuron]
            changed = True if old_state != new_state else changed
        iterations += 1
        if not changed:
            break

    recovered_pattern = np.copy(self.state)
    return recovered_pattern
```

1.4 Pattern Recovery Visualizations

1.4.1 Successful Recovery from a 2-image Hebb-trained Network



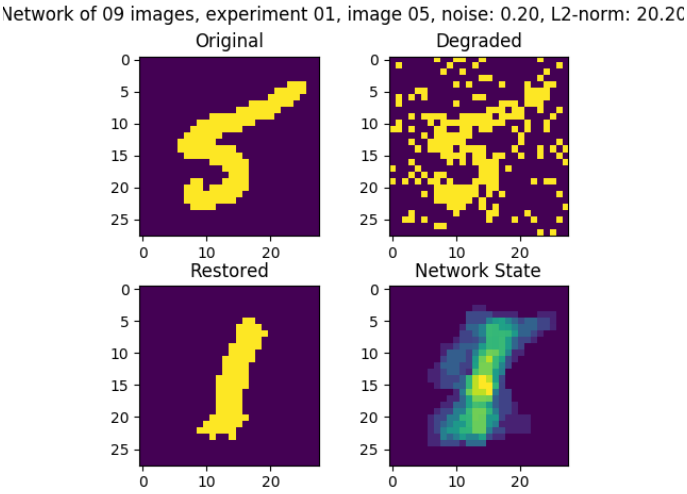
(a) Recovering a 5



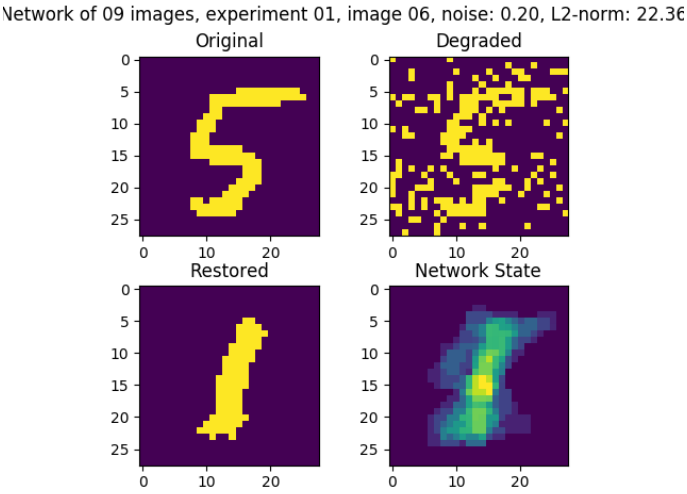
(b) Recovering a 1

Recovering from a 2-image Hebb-trained network

1.4.2 Failed Recovery from a 9-image Hebb-trained Network



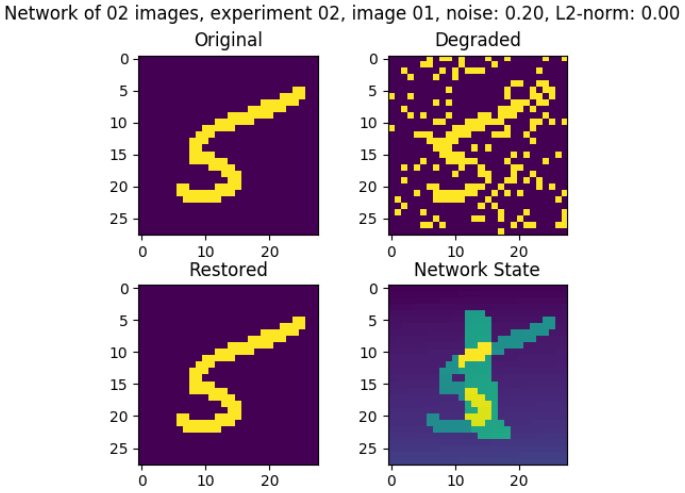
(a) Failing to recover a 5



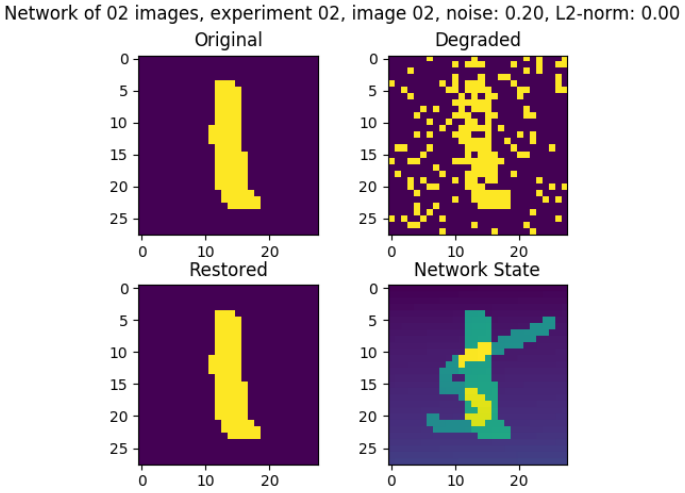
(b) Failing to recover a 5, again

Failing to recover from a 9-image Hebb-trained network

1.4.3 Successful Recovery from Storkey-trained Networks

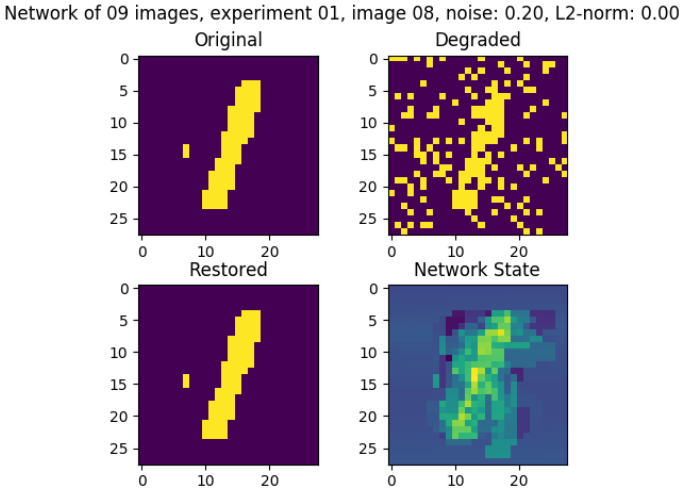


(a) Recovering a 5

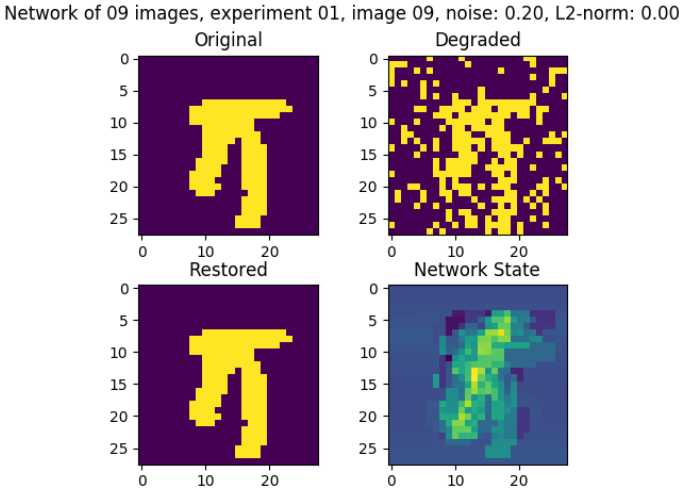


(b) Recovering a 1

Recovering from a 2-image Storkey-trained network



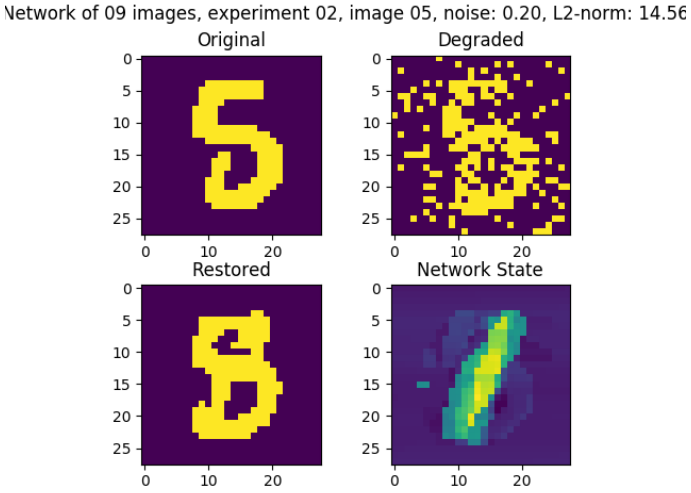
(a) Recovering a 1



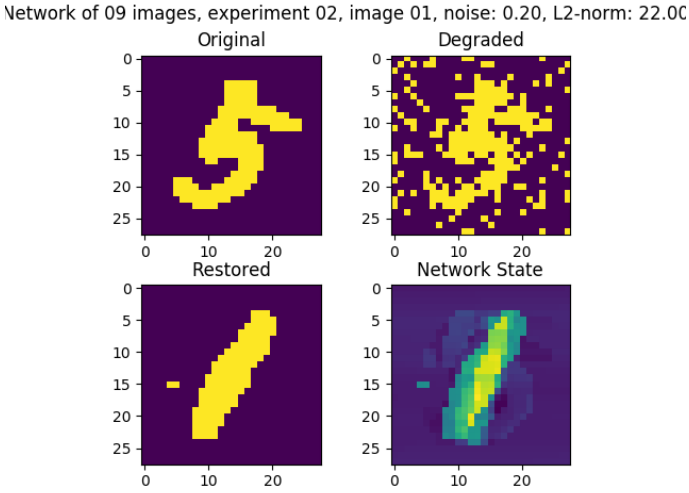
(b) Recovering a 1

Recovering from a 9-image Storkey-trained network

1.4.4 Failed Recovery from Storkey-trained Networks



(a) Failing to recover a 5



(b) Failing to recover a 5, again

Failing to recover from a 9-image Storkey-trained network

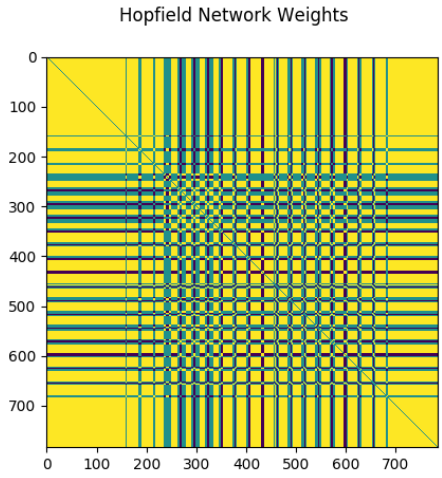
1.5 Network Visualizations

We visualized our Hopfield networks after training them. We did this to get a better understanding of how the and Hebb and Storkey learning rules affect the network. We found that Hebb's rule produced a simpler, less-detailed network; conversely, Storkey's rules produced a more nuanced network.

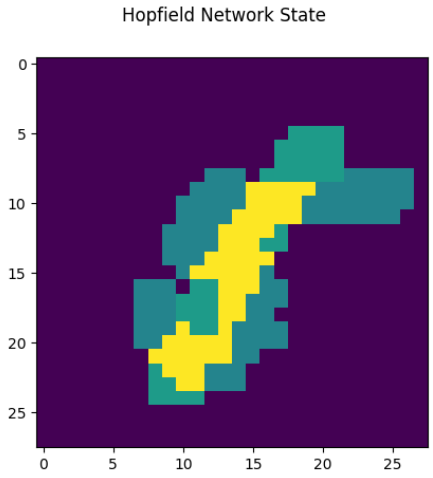
You can see this in the diagrams that follow: in Hebb-trained networks, the visualizations use fewer colors; this means that there's less information in the network. Meanwhile in Storkey-trained networks, the presence of more shades of color indicate a nuanced, more detailed network.

Furthermore, it's also interesting to visually see patterns stored in the network in the "Network State" visualizations. We did this visualization by summing the weights of each neuron to decide on the color; then plotting a 28x28 grid of neurons with their respective colors.

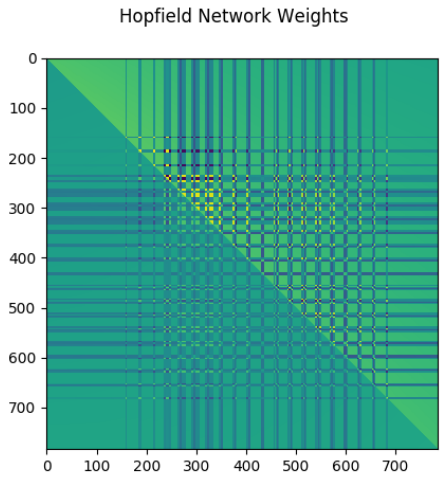
In both examples, the same two patterns were used to train the networks.



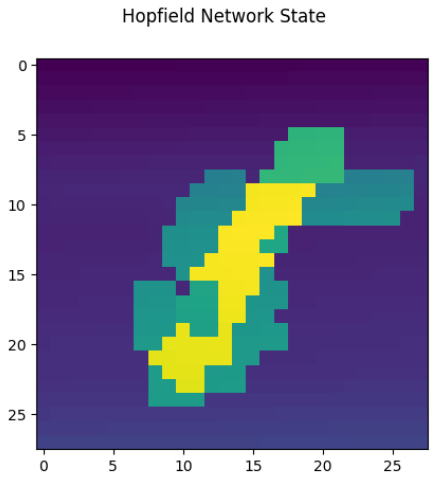
(a) Hebb Network Weights



(b) Hebb Network State



(c) Storkey Network Weight



(d) Storkey Network State

2 Question 2

2.1 Running Our Code

We have provided an easy-to-use Makefile to help you run our program:

```
make prepare-venv
./env/bin/python som.py
```

Otherwise, if you have tensorflow, matplotlib, numpy, and scikit-learn installed already:

```
python3 som.py
```

Running the program will do the following:

1. Fetch MNIST data and retrieve the 1's and 5's; this will be our dataset.
2. Initialize the SOM and train it on a subset of the dataset from step 1, in random order without replacement.
3. Test the clustering accuracy of the trained SOM and print out the value.
4. Perform K-means clustering on the dataset reduced to 2 dimensions.
5. Test the clustering accuracy of the K-means clustering and print out the value.

After the programs performs the above, it will display three visualizations:

- The state of the SOM upon initialization.
- The state of the SOM after training.
- The K-means clustering of the dataset.

2.2 Accuracy of Clustering

When you run our program it will print out the clustering accuracies of our trained SOM as well as our K-means output. We got the following accuracy values:

- **SOM clustering accuracy: 94.80%**
- **K-means clustering accuracy: 90.94%**

2.3 SOM Dimensions

In our implementation of an SOM, we used the following architecture:

- **Input layer:** 784 neurons / features
- **Output layer:** 900 neurons

In our implementation, we found it useful to consider the output layer as a 30x30 grid of neurons. Note that this is identical to saying we used 900 output neurons.

2.4 SOM Parameters

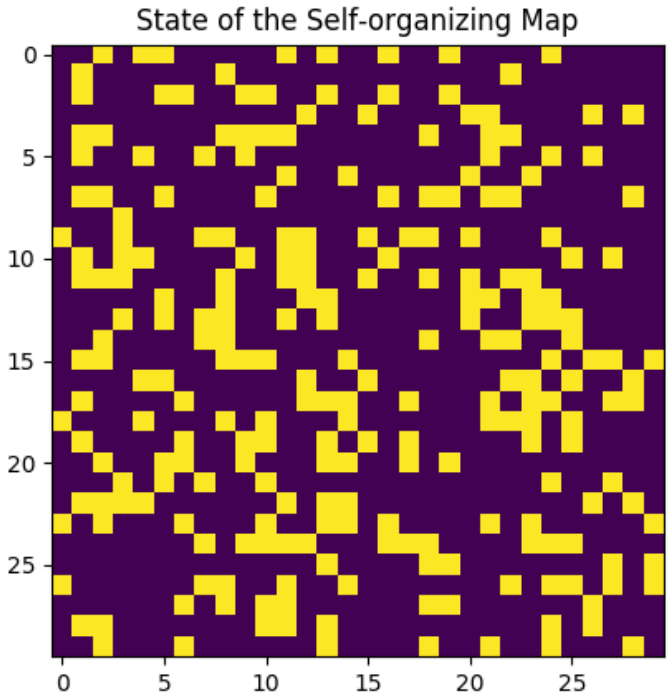
We used the following parameters for our SOM implementation:

- Learning rate: 0.5
- sigma (σ): 5.0 *(for the Gaussian that updates the winner neuron's neighbours)*
- Number of input neurons: 784
- Number of output neurons: 900

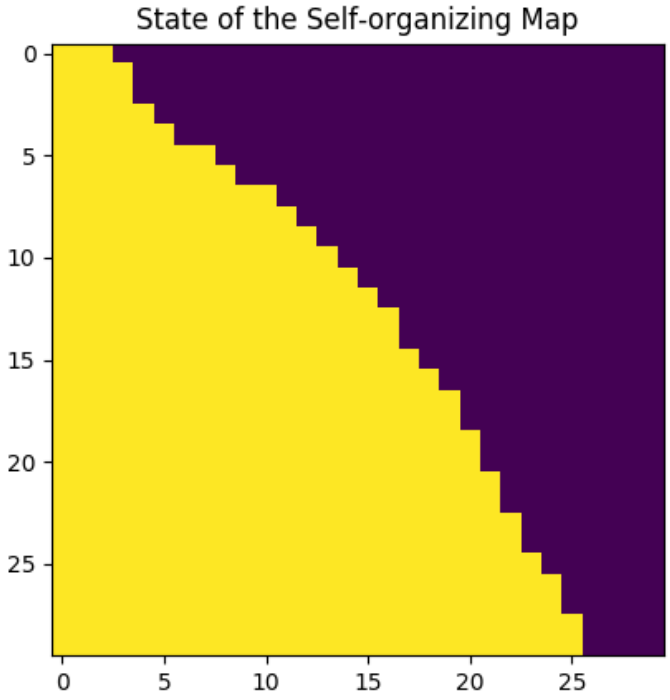
For our weights, we initialized them according to a normal distribution with $\mu = 0.5$ and $\sigma = 1.0$.

2.5 Visualizations

2.5.1 SOM Network



(a) Before Training



(b) After Training

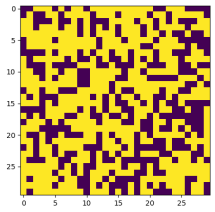
The SOM Network Before & After Training

2.5.2 SOM Training Process

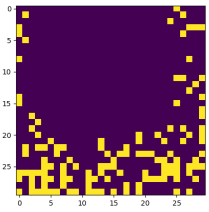
We were interested in visualizing the SOM's training process. We created visualizations after each training iteration to see how the SOM changes over time. We start off in the top left with random weights.

We progressively train with MNIST images until, at iteration 15 (bottom right) we have a clear partitioning of the map into two partitions. Training any more just makes the map dance around more, not improving the map very much.

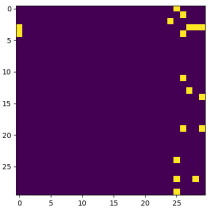
The purple partition represents MNIST images of the number 1. The yellow partition represent MNIST images of the number 5.



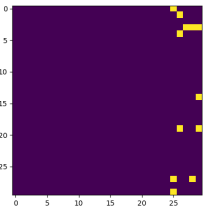
(a) Iteration 0



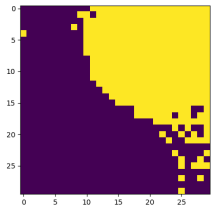
(b) Iteration 1



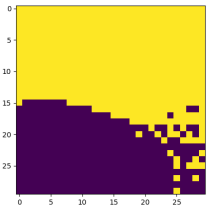
(c) Iteration 2



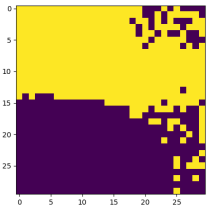
(d) Iteration 3



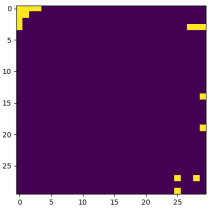
(e) Iteration 4



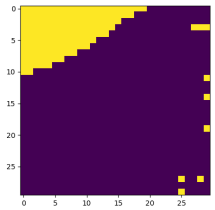
(f) Iteration 5



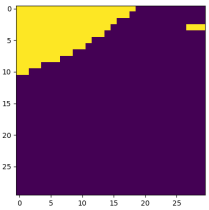
(g) Iteration 6



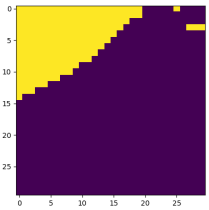
(h) Iteration 7



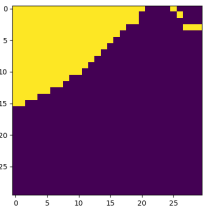
(i) Iteration 8



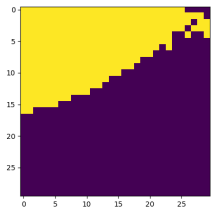
(j) Iteration 9



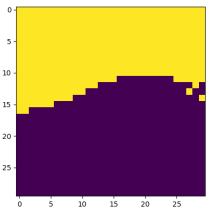
(k) Iteration 10



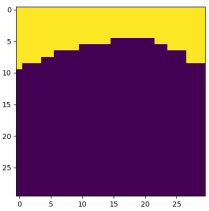
(l) Iteration 11



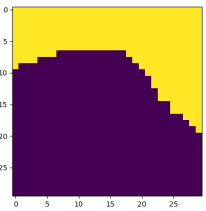
(m) Iteration 12



(n) Iteration 13



(o) Iteration 14

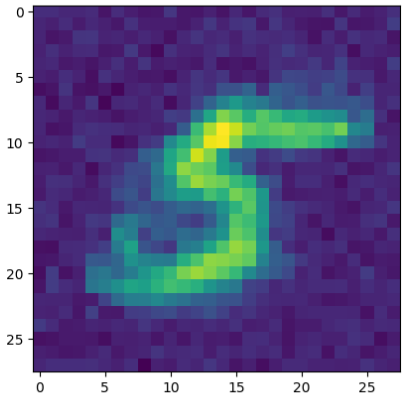


(p) Iteration 15

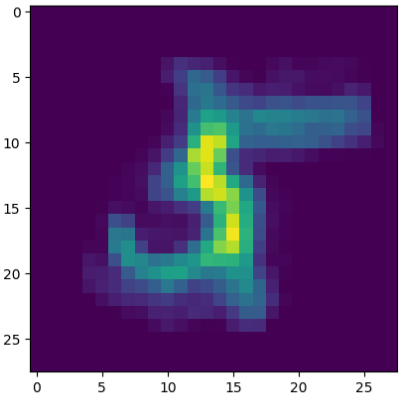
2.5.3 SOM Output Neuron Prototypes

We were interested in visualizing a few neurons' prototypes. A single neuron's weights to the input later (784 weights in our case) represents its prototype.

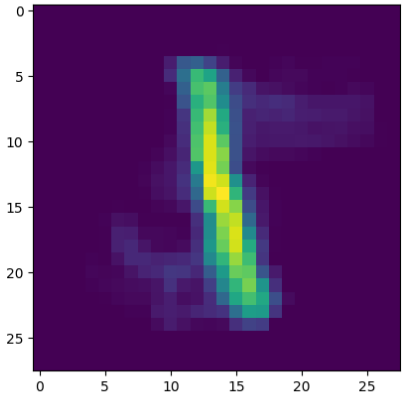
The following charts visualize the prototypes of a few randomly selected prototypes. Seeing these neat visualizations help us be confident that our SOM is behaving correctly.



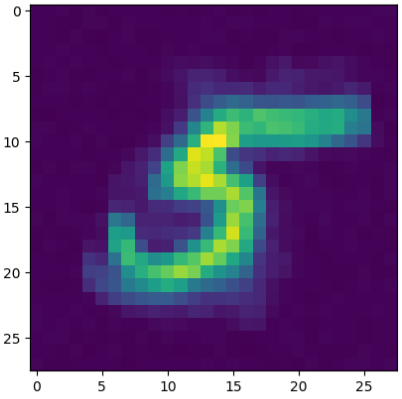
(a) Random Prototype #1



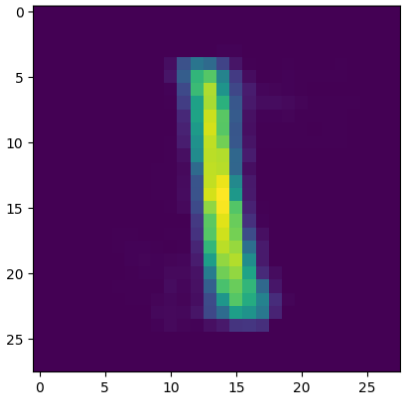
(b) Random Prototype #2



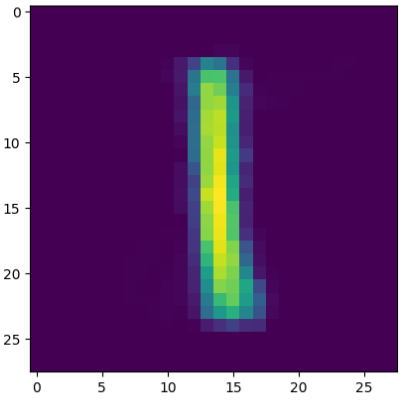
(c) Random Prototype #3



(d) Random Prototype #4



(e) Random Prototype #5

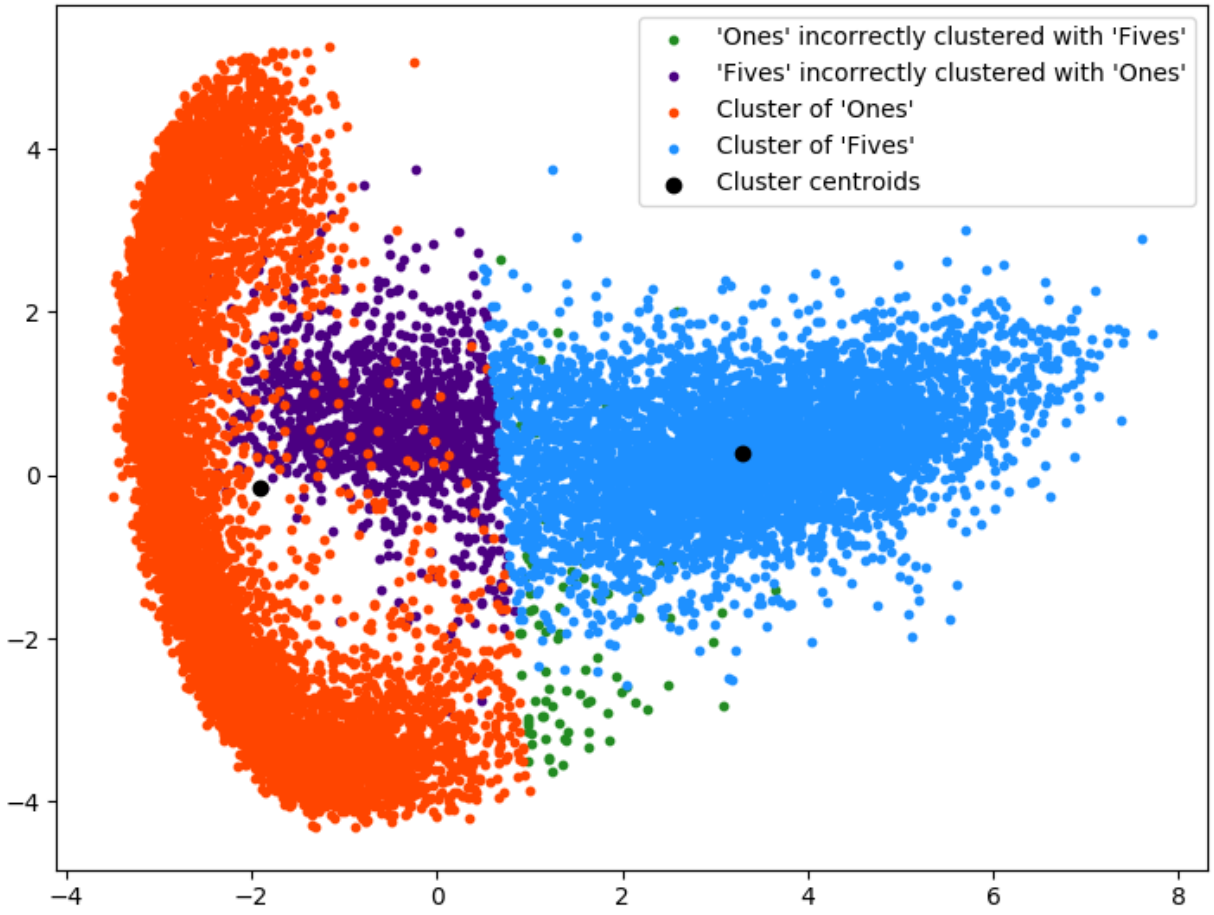


(f) Random Prototype #6

2.5.4 K-means Clustering

We wanted to keep the comparison between SOM "clustering" and K-means clustering close, so we let $k = 2$. After performing the K-means clustering, we were able to tell if each element was correctly clustered because we know the label for each MNIST image. We performed PCA on the MNIST data to reduce it to 2 dimensions for visualization, as the assignment specification suggests. Note that our use of PCA involves performing SVD, which the assignment also suggested for this part of the assignment.

This allowed us to create the chart below, which indicated the elements that were clustered correctly and those that were not. We also show the cluster centroids computed by the K-means algorithm.



K-means Clustering of the MNIST Data Reduced to 2D

3 Question 3

3.1 Running Our Code

Important: our submission includes the LFW face data that our program needs. We had to create a separate Python 2.7 utility obtain this data because the scikit-learn tutorial requires PIL (an outdated, little-supported) to read the data. We had to compile PIL from source, so it's easier for us to include the data instead of making you manually compile PIL and run *yet* another script.

We have provided an easy-to-use Makefile to help you run our program:

```
make prepare-venv
./env/bin/python eigenfaces.py      # don't perform PCA on face data
./env/bin/python eigenfaces.py 100  # perform PCA to obtain 100 components / eigenfaces
```

Otherwise, if you have tensorflow, matplotlib, numpy, and scikit-learn installed already:

```
# Syntax
python3 eigenfaces.py [num_PCA_components]
# Example
python3 eigenfaces.py      # don't perform PCA on face data
python3 eigenfaces.py 100  # perform PCA to obtain 100 eigenfaces
```

Running the program will do the following:

1. Import the LFW face data from local files *(provided with our submission)*.
2. Perform 10-fold cross-validation; for each fold do:
   - Initialize a feed-forward network from scratch.
   - Train the classifier on all 9 training folds for 100 epochs.
   - After the final epoch, test for accuracy using the testing fold.
   - Save the accuracy for this fold.
3. Compute the average accuracy over the 10 folds and print it out.

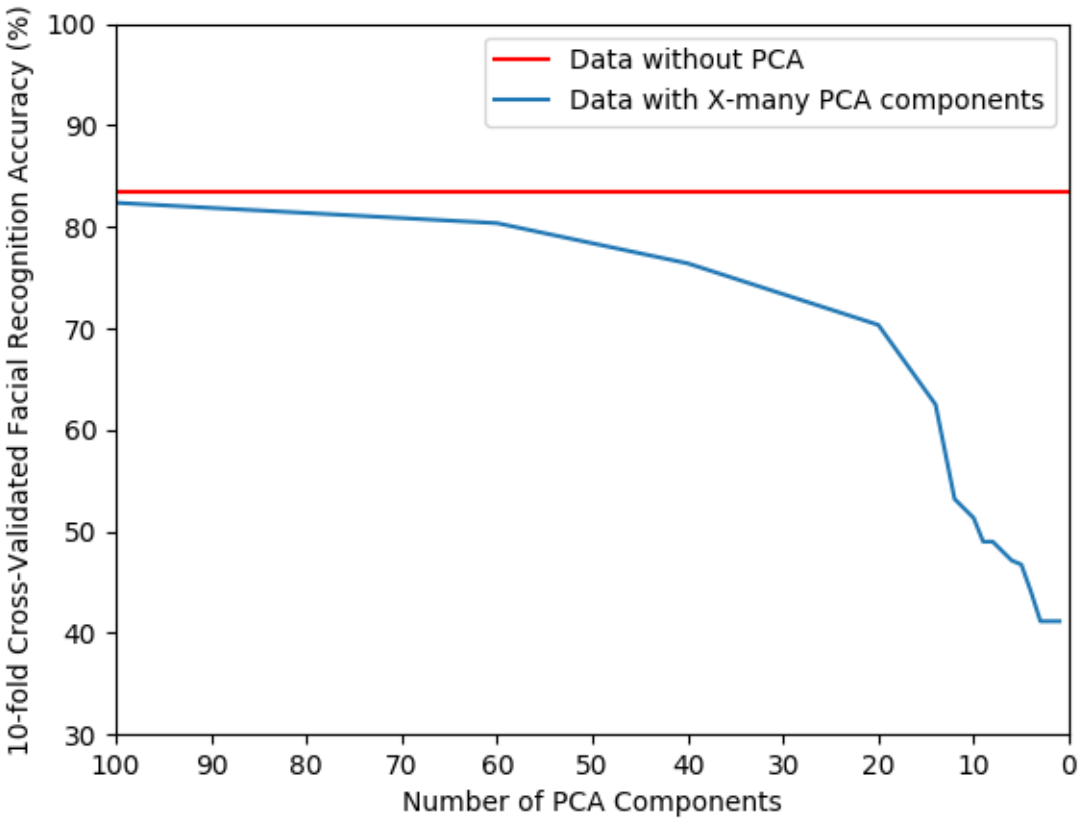
3.2 Accuracy of Classification / Facial Recognition

First, we tested our feed-forward network with the face data to see how well we can classify faces without the use of PCA. We performed 10-fold cross validation, along with the following hyper-parameters:

```
# Hyper-parameters
num_folds = 10
batch_size = 64
epochs = 100
learning_rate = 0.020

# Architecture (USED FOR ALL EXPERIMENTS)
input_neurons = 1850     # face images were 50x37 in size
hidden_neurons_1 = 160   # hidden layer 1
hidden_neurons_2 = 60    # hidden layer 2
output_neurons = 7       # dataset had faces of 7 unique people
```

This yielded a classifier with **83.39%** averaged cross-validated accuracy.
We then applied PCA to the data before training on it, again using 10-fold cross-validation. We tried several values for the number of PCA components (eigenfaces) to reduce the data to. **We used the exact same architecture and hyper-parameters as the experiment on the raw face data.** These experiments on the dimensionality-reduced datasets yielded the cross-validated accuracy values in the chart below:

3.2.1 Accuracy Chart



Classifier Accuracy: with and without PCA

3.2.2 Accuracy Chart Code

```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

non_pca_accuracy = 83.39

num_eigenfaces = [
    100, 60, 40, 20,
     14, 12, 10,  9,
      8,  7,  6,  5,
      4,  3,  2,  1,
]
pca_accuracy = [
    82.37, 80.36, 76.40, 70.34,
    62.50, 53.19, 51.32, 48.99,
    48.99, 48.06, 47.13, 46.74,
    44.10, 41.15, 41.15, 41.15,
]

fig = plt.figure()
ax = fig.gca()

plt.axhline(y=non_pca_accuracy, color='red')
plt.plot(num_eigenfaces, pca_accuracy)

ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.axis([100, 0, 30, 100])
plt.xlabel("Number of PCA Components")
plt.ylabel("10-fold Cross-Validated Facial Recognition Accuracy (%)")
plt.legend(['Data without PCA', 'Data with X-many PCA components'], loc='upper right')

plt.show()
```

3.3 Implementation

3.3.1 Performing PCA

We used scikit-learn to perform PCA on the input face data:

```
num_components = int(sys.argv[1])
pca = PCA(n_components=num_components, svd_solver='randomized', whiten=True).fit(data)
data = pca.transform(data)
num_features = data.shape[1]
```

3.3.2 10-fold Cross-validation

In our training process, our program performs the following: * for each of the 10 folds: * initialize a feed-forward network from scratch * train the classifier on all training folds for 100 epochs * after the final epoch, save the accuracy using the testing fold * compute the average accuracy over the 10 folds

```
for fold in range(num_folds):
    print('Using fold {:02d} / {:02d} as the training fold:'.format(fold + 1, num_folds))

    train_indices, test_indices = folds[fold]
    trX, teX = data[train_indices], data[test_indices]
    trY, teY = labels[train_indices], labels[test_indices]

    with tf.Session() as sess:
        tf.global_variables_initializer().run()
        for epoch in range(1, epochs + 1):
            batch_starts = range(0, len(trX), batch_size)
            batch_ends = range(batch_size, len(trX) + 1, batch_size)

            for start, end in zip(batch_starts, batch_ends):
                sess.run(train_op, feed_dict={X: trX[start:end], Y: trY[start:end]})

            if epoch % 20 == 0:
                epoch_accuracy = np.mean(np.argmax(teY, axis=1) == sess.run(predict_op, feed_dict={X:
                print('\tEpoch {:3} ---> {:.2f}%'.format(epoch, epoch_accuracy * 100))

    fold_accuracy.append(epoch_accuracy)
    print('Accuracy with fold #{} as training: {:.2f}%\n'.format(fold + 1, fold_accuracy[-1] * 100))

accuracy = sum(fold_accuracy) / len(fold_accuracy)
print('Average {}-fold cross validation accuracy: {:.2f}%'.format(num_folds, accuracy * 100))
```

---

4 References

4.1 Question 1

- https://en.wikipedia.org/wiki/Hopfield_network
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.103&rep=rep1&type=pdf *(Storkey's paper)*

4.2 Question 2

- https://github.com/JustGlowing/minisom
- https://en.wikipedia.org/wiki/Self-organizing_map

4.3 Question 3

- We used the TensorFlow feed-forward code from our in-class tutorials.
- http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
- http://scikit-learn.org/0.18/auto_examples/applications/face_recognition.html

---