1 One-dimensional Kohonen network

1.1 a)

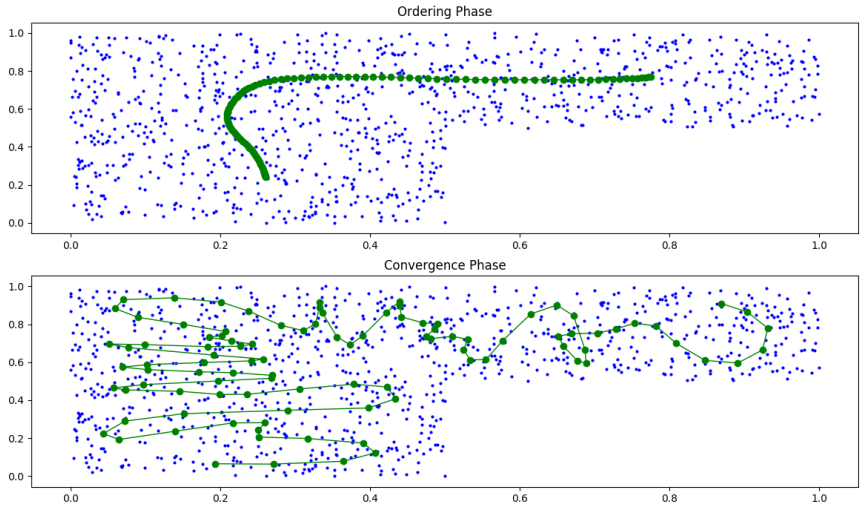

Figure 1: Plot of the weights (green) and the generated patterns (blue) after the different phases with $\sigma_0 = 100$

The network does learn the properties of the distribution and adjusts the output weights accordingly as seen in the picture above. In the ordering phase the output weights updates in a "smooth" way even though it's learning from the patterns. The convergence phase is more of a process of fine-tuning the output weights to the input data as seen in the last picture, even though this process is not as smooth as the ordering phase.

1.2 b)

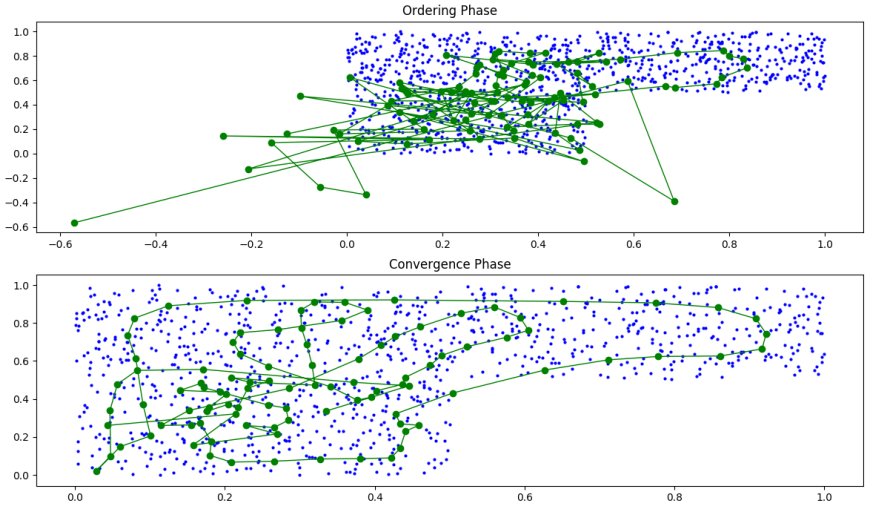

Figure 2: Plot of the weights (green) and the generated patterns (blue) after the different phases with $\sigma_0 = 5$

There is no clear order obtained in this case as seen in the picture above. Despite this, the network manages to learn and adapt to the input pattern and gets quite close to the result seen in the previous case, especially after the convergence phase. The way we choose our learning parameter and sigma is especially important and makes a big difference in our two output cases. In our case we are given the parameters but in real-life scenarios there is much trial and error to find values that result in a well-performing network.

2 Unsupervised Oja's learning with one linear unit

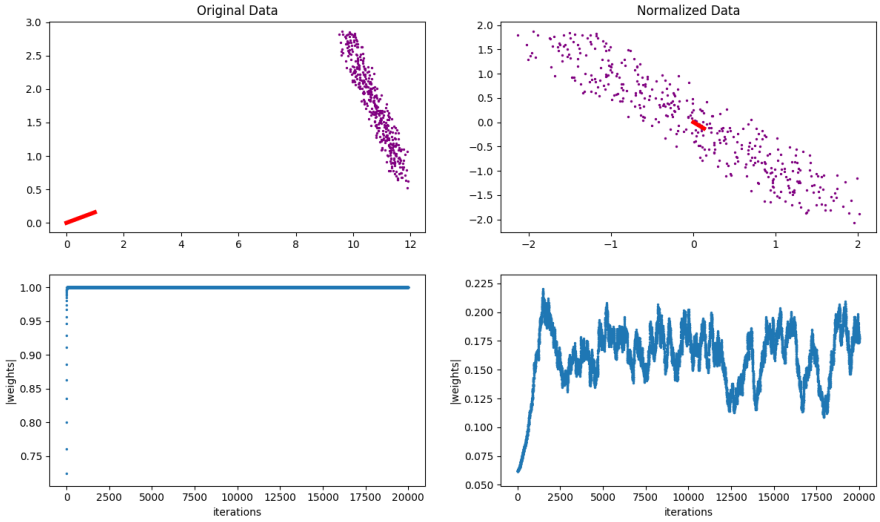

Figure 3: Plot of the weights (red), the input data (purple) and the modulus of the weight vector (blue)

We can see by comparing our results that the modulus of the weight vector converges to 1 when running on the original data, which doesn't happen when running on normalized data. Since this is the first of three properties of a weight vector to which the network is expected to converge, we do know that we don't converge in the case with normalized data.

3 Unsupervised simple competitive learning combined with supervised simple perceptron

After some discussion, we decided to not split our data set into training- and testing data. We took this decision after we plotted the given data for this task and realized that it was quite obvious that the data were generated from mathematical distributions and not real data. Training with all our data will therefore only make our model more accurate, and we will not experience the disadvantages with overfitting.

3.1 a

Our best experiment with 4 neurons gave us an average classification error of 0.032375. After the unsupervised simple competitive learning, the weights of our Gaussian neurons were following:

$$\begin{pmatrix} -6.41740939 & 5.36536793 \\ 16.73233576 & -0.23593904 \\ 4.61508451 & 7.27280455 \\ 6.70788162 & -2.49057948 \end{pmatrix}$$

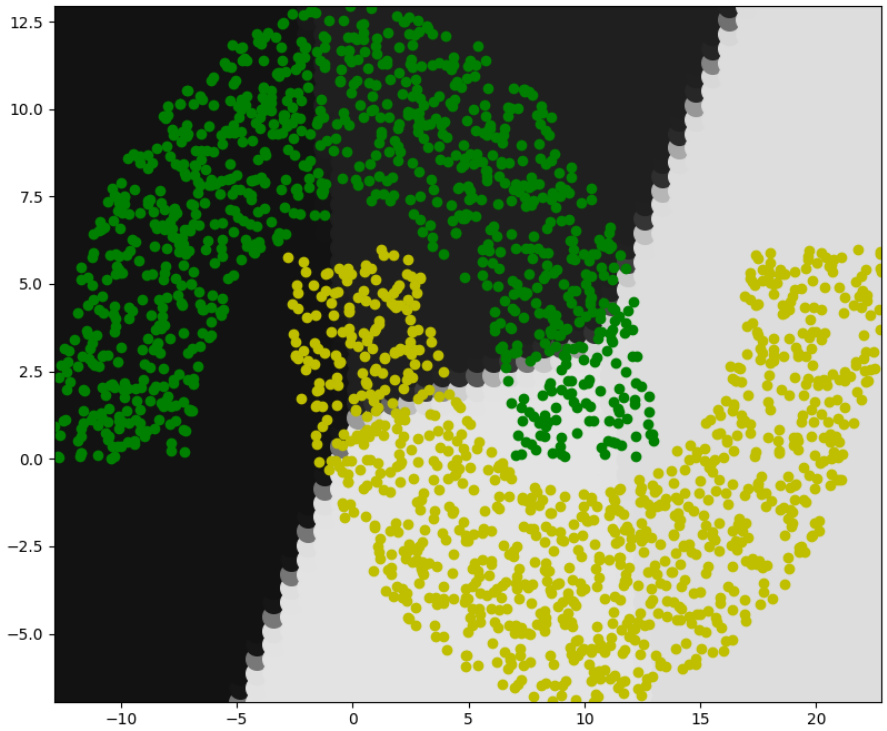

Figure 4: Plot of the input data and decision boundary with 4 Gaussian neurons

With 4 neurons our model are not able to classify the data especially good. Even though we get an average error of around 3.2%, which usually is not that bad, we clearly see on

our decision-boundary that our model doesn't perform good enough. Our input data is too complex to represent with this amount of Gaussian neurons.
The unsupervised network in our model clusters the data and sends the output to the supervised network, which then acts perceptron and does the classification of the clustered data. Since we don't get a especially detailed cluster of our input with only 4 Gaussian neurons, our classification in the supervised network isn't able to perform as good as we expect.

3.2 b

Our best experiment with 10 neurons gave us an average classification error of 0.0003. After the unsupervised simple competitive learning, the weights of our Gaussian neurons were following:

$$\begin{pmatrix} 9.80250726 & -3.68018599 \\ -7.31883889 & 7.16076731 \\ 19.75246521 & 2.96505811 \\ 9.57427408 & 3.44711739 \\ 4.39754371 & -1.93107159 \\ 15.02716904 & -2.37800023 \\ 4.75191332 & 8.86723937 \\ -9.86685303 & 2.41246487 \\ -2.03306545 & 10.10086259 \\ 0.18621978 & 2.91234529 \end{pmatrix}$$

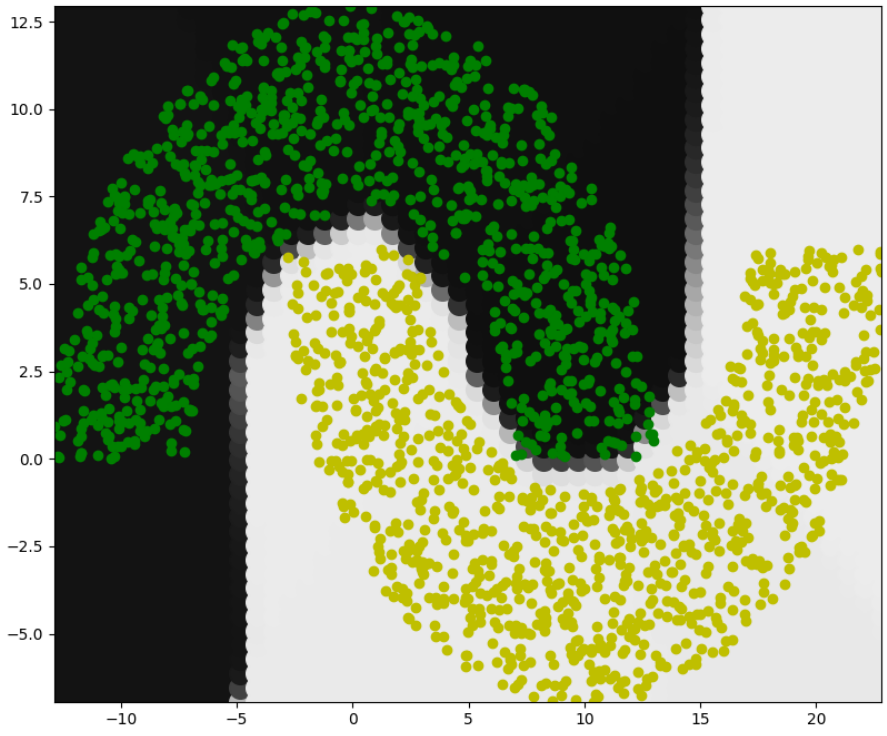

Figure 5: Plot of the input data and decision boundary with 10 Gaussian neurons

Training our model with 10 Gaussian neurons instead makes it possible for the network to classify more complex data, as the data in this task. This is possible because our unsupervised model can generate a more detailed cluster which makes it easier for the supervised classification to perform well.

3.3 c

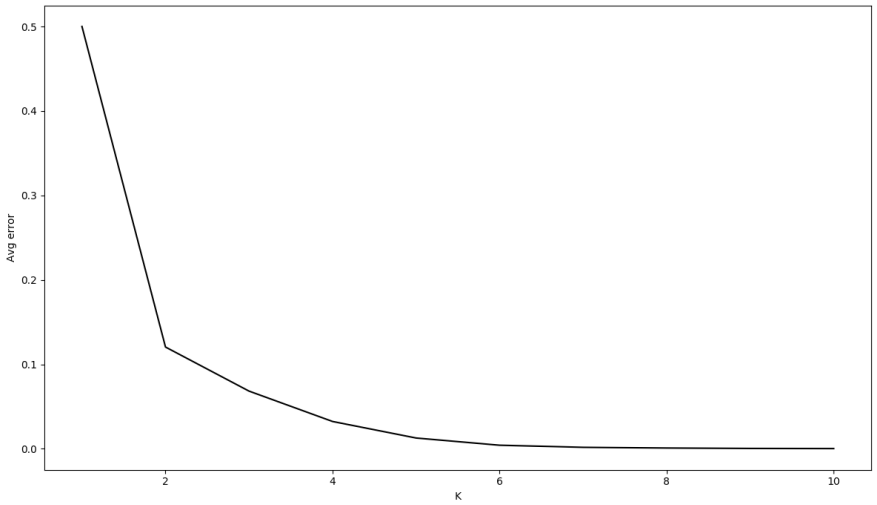

Figure 6: Plot of best avg error for different amount of neurons

The more Gaussian neurons our weights contain, the better the model performs. Since each neuron uses a nonlinear activation function in the simple perceptron network, it's possible for a model with more neurons to adapt to complex data. Since the detailness of the clustered data we send from our unsupervised network to our supervised network depends on the size of our Gaussian neurons, our model performs better for each neuron we add.

4 Program code

4.1 One-dimensional Kohonen network

4.1.1 som.py

```
import numpy as np
import matplotlib.pyplot as plt
import sys
from random import randint

INPUT_SIZE = 1000
NEURON_SIZE = 100

TAO_SIGMA = 300

T_ORDER = 1000
SIGMA_ORDER_ZERO = 5
ETA_ORDER_ZERO = .1

T_CONV = 20000
SIGMA_CONV = .9
ETA_CONV = .01

PLOT_POINTS = False
PLOT_ION = False


def gaussian_width(sigma_zero, t, tao_sigma):
    return sigma_zero * np.exp(-float(t) / tao_sigma)


def _generate_points(x1_low, x1_high, x2_low, x2_high, size):
    x1 = np.random.uniform(x1_low, x1_high, size)
    x2 = np.random.uniform(x2_low, x2_high, size)
    points = []
    for i in range(size):
        points.append([x1[i], x2[i]])
    return np.array(points)


def generate_neurons(size):
    return _generate_points(-1, 1, -1, 1, size)


def generate_points(size):
    iter_size = size // 3
    a = _generate_points(0, 0.5, 0, 0.5, iter_size)
    a = np.concatenate((a, _generate_points(0.5, 1, 0.5, 1, iter_size)))
```

```
    a = np.concatenate((a, _generate_points(0, 0.5, 0.5, 1,
        (size - (iter_size * 2)))))
    return a


def generate_weights(n_points, n_neurons):
    return np.random.rand(n_points, n_neurons) * 2 - 1


def calc_distance(point, neuron):
    return np.linalg.norm(point - neuron)


def neighbor_func(winning_index, other_index, sigma):
    return np.exp((-((other_index - winning_index) ** 2) /
        (2 * (sigma ** 2))))


def calc_delta_weight(learning_rate, neighbor, distance):
    return learning_rate * neighbor * distance


def find_nearest_neuron(point_index, points, neurons_weights):
    min_distance = 1000000.0
    nearest_neuron_index = 0
    for i in range(0, len(neurons_weights)):
        dist = calc_distance(points[point_index], neurons_weights[i])
        if dist < min_distance:
            nearest_neuron_index = i
            min_distance = dist
    return nearest_neuron_index


def get_random_point_index(high):
    return randint(0, high - 1)


points = generate_points(INPUT_SIZE)
np.random.shuffle(points)
neurons_W = generate_neurons(NEURON_SIZE)

plt.figure(1)
plt.subplot(211)
plt.title('Ordering Phase')

if PLOT_ION:
    plt.ion()
    plt.scatter(points[:, 0], points[:, 1], c='b')
```

```
    plt.plot(neurons_W[:, 0], neurons_W[:, 1], 'go-', linewidth=1)
    plt.show()

for i in range(T_ORDER):
    sys.stdout.write(
        "ORDER: Running epoch %d of %d...\r" % (i + 1, T_ORDER))
    sys.stdout.flush()

    learning_rate = gaussian_width(ETA_ORDER_ZERO, i, TAO_SIGMA)
    sigma_rate = gaussian_width(SIGMA_ORDER_ZERO, i, TAO_SIGMA)

    random_point_index = get_random_point_index(INPUT_SIZE)
    winning_neuron_index = find_nearest_neuron(random_point_index,
        points, neurons_W)
    for j in range(NEURON_SIZE):
        neurons_W[j] += calc_delta_weight(learning_rate,
            neighbor_func(winning_neuron_index, j, sigma_rate),
                points[random_point_index] - neurons_W[j])

    if PLOT_ION:
        if i % 50 == 0:
            plt.clf()  # Clears plot before rendering
            if PLOT_POINTS:
                plt.scatter(points[:, 0], points[:, 1], c='b', s=8)
            plt.plot(neurons_W[:, 0], neurons_W[:, 1], 'go-', linewidth=1)
            plt.pause(0.1)

if PLOT_POINTS:
    plt.scatter(points[:, 0], points[:, 1], c='b', s=4)
plt.plot(neurons_W[:, 0], neurons_W[:, 1], 'go-', linewidth=1)

print ''

plt.subplot(212)
plt.title('Convergence Phase')

for i in range(T_CONV):
    sys.stdout.write("CONV: Running epoch %d of %d...\r" % (i + 1, T_CONV))
    sys.stdout.flush()
    random_point_index = get_random_point_index(INPUT_SIZE)
    winning_neuron_index = find_nearest_neuron(random_point_index,
        points, neurons_W)
    for j in range(NEURON_SIZE):
        neurons_W[j] += calc_delta_weight(ETA_CONV,
            neighbor_func(winning_neuron_index, j, SIGMA_CONV),
                points[random_point_index] - neurons_W[j])

    if PLOT_ION:
```

```
            if i % 1000 == 0:
                plt.clf() # Clears plot before rendering
                if PLOT_POINTS:
                    plt.scatter(points[:, 0], points[:, 1], c='b')
                plt.plot(neurons_W[:, 0], neurons_W[:, 1], 'go-', linewidth=1)
                plt.pause(0.1)

if PLOT_ION:
    plt.ioff()
    plt.clf()
print ''

if PLOT_POINTS:
    plt.scatter(points[:, 0], points[:, 1], c='b', s=4)
plt.plot(neurons_W[:, 0], neurons_W[:, 1], 'go-', linewidth=1)
plt.show()
```

4.2 Unsupervised Oja's learning with one linear unit

4.2.1 oja.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

LEARNING_RATE = 0.001
updates = np.arange(0, 20000)


def generate_weights(n_points, n_neurons):
    return np.random.rand(n_points, n_neurons) * 2 - 1


def get_input_patterns(lines):
    a = []
    for l in lines:
        first = float(l.split("\t")[0])
        second = float(l.split("\t")[1])
        a.append(np.array([first, second]).T)
    return np.asarray(a)


def random_pattern_index(size):
    return np.random.randint(0, size)


def network_output(input_pattern, weights):
    return np.sum(np.multiply(weights, patterns[input_pattern]))


def update_weight(input_patterns, index, weights):
    output = network_output(index, weights)
    return LEARNING_RATE * output *
        (input_patterns[index] - output * weights)


def oja(input_data, weights):
    weight_modulus = []
    for i in range(len(updates)):
        index = random_pattern_index(len(input_data))
        weights += update_weight(input_data, index, weights)
        weight_modulus.append(np.linalg.norm(weights))
    return weights, weight_modulus


f = open('data_ex2_task2_2017.txt', 'r')
```

```
data_lines = f.readlines()
f.close()

patterns = get_input_patterns(data_lines)
norm_patterns = stats.zscore(patterns)

W = generate_weights(1, 2)
norm_W = generate_weights(1, 2)

W, W_mod = oja(patterns, W)
norm_W, norm_W_mod = oja(norm_patterns, norm_W)

f, axarr = plt.subplots(2, 2)
axarr[0, 0].set_title('Original Data')
axarr[0, 0].scatter(patterns[:, 0], patterns[:, 1], color='purple', s=2)
axarr[0, 0].plot([0, W[0, 0]], [0, W[0, 1]], color='red', linewidth=4)

axarr[1, 0].set_ylabel('|weights|')
axarr[1, 0].set_xlabel('iterations')
axarr[1, 0].scatter(updates, np.asarray(W_mod), s=2)

axarr[0, 1].set_title('Normalized Data')
axarr[0, 1].scatter(norm_patterns[:, 0],
    norm_patterns[:, 1], color='purple', s=2)
axarr[0, 1].plot([0, norm_W[0, 0]], [0, norm_W[0, 1]],
    color='red', linewidth=4)

axarr[1, 1].set_ylabel('|weights|')
axarr[1, 1].set_xlabel('iterations')
axarr[1, 1].scatter(updates, np.asarray(norm_W_mod), s=2)

plt.show()
```

4.3 Unsupervised simple competitive learning combined with supervised simple perceptron

4.3.1 main.py

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

N_NEURONS = 1
LEARNING_RATE = 0.02
N_UPDATES = 100000
EXPERIMENTS = 20
PIXEL_PER_RANGE = 50


def get_data(lines):
    temp = []
    Y = []
    for l in lines:
        temp_b = l.split("\t")
        first = float(temp_b[0])
        second = float(temp_b[1])
        third = float(temp_b[2])
        Y.append(first)
        temp.append(np.array([second, third]).T)
    return np.asarray(Y), np.asarray(temp)


def generate_weights(n_neurons, n_cols):
    return np.random.rand(n_neurons, n_cols) * 2 - 1


def random_pattern_index(size):
    return np.random.randint(0, size)


def activation(x, w):
    denominator = 0
    for i in range(0, N_NEURONS):
        denominator += np.exp((-np.linalg.norm(x - w[i]) ** 2) / 2)
    g_ = []
    for i in range(0, N_NEURONS):
        g = np.exp((-np.linalg.norm(x - w[i]) ** 2) / 2) / denominator
        g_.append(g)
    return np.array(g_, dtype=np.float_)
```

```
def shuffle_data(X, Y):
    if len(X) != len(Y):
        raise IndexError('X and Y does need to have same length')
    a = np.arange(0, len(X))
    np.random.shuffle(a)

    Y_rand, X_rand = [], []
    for i in a:
        X_rand.append(X[i])
        Y_rand.append(Y[i])
    return np.array(X_rand), np.array(Y_rand)


f = open('data_ex2_task3_2017.txt', 'r')
data_lines = f.readlines()
f.close()

best_error = 1
best_weight_supervised = None
best_weight_unsupervised = None

for experiment in range(0, EXPERIMENTS):
    print 'Running experiment %d by %d\n' % (experiment + 1, EXPERIMENTS)
    Y, patterns = get_data(data_lines)

    # Unsupervised simple competitive learning
    weights = generate_weights(N_NEURONS, 2)

    for index in range(0, N_UPDATES):
        rpi = random_pattern_index(len(patterns))
        winning_index = np.argmax(activation(patterns[rpi], weights))
        weights[winning_index] += LEARNING_RATE *
            (patterns[rpi] - weights[winning_index])
        sys.stdout.write("Running unsupervised training epoch %d of %d...\r"
            % (index + 1, N_UPDATES))
        sys.stdout.flush()
    print ''

    g_s = []
    for i in range(0, len(patterns)):
        g_s.append(activation(patterns[i], weights))

    X_new, Y_new = shuffle_data(g_s, Y)

    # Supervised simple perceptron network
    sm = supervised_model.SupervisedModel(N_NEURONS, 1)
    sm.train(X_new, Y_new)
    experiment_error = sm.valid(X_new, Y_new)
```

```
    print '\n'

    if experiment_error < best_error:
        # Store best experiment and it's result
        best_error = experiment_error
        best_weight_supervised = sm.w
        best_weight_unsupervised = weights

print 'Best avg classification error:', best_error
print 'Best supervised weights:', best_weight_supervised
print 'Best unsupervised weights:', best_weight_unsupervised

# Plotting
Y, patterns = get_data(data_lines)
input_pos = []
input_neg = []
for i in range(0, len(patterns)):
    if Y[i] == 1:
        input_pos.append(patterns[i])
    else:
        input_neg.append(patterns[i])
input_pos = np.array(input_pos)
input_neg = np.array(input_neg)

x_max, x_min = np.amax(patterns[:, 0]), np.amin(patterns[:, 0])
y_max, y_min = np.amax(patterns[:, 1]), np.amin(patterns[:, 1])

test_X = np.linspace(x_min, x_max, PIXEL_PER_RANGE)
test_Y = np.linspace(y_min, y_max, PIXEL_PER_RANGE)
plot_matrix = np.empty([PIXEL_PER_RANGE, PIXEL_PER_RANGE])

best_sm = supervised_model.SupervisedModel(N_NEURONS,
    1, best_weight_supervised)
color_map = plt.get_cmap('binary')

for i in range(0, PIXEL_PER_RANGE):
    for j in range(0, PIXEL_PER_RANGE):
        uns = activation(np.array([test_X[i], test_Y[j]]),
            best_weight_unsupervised)
        guess = best_sm.get_guess(uns)
        plt.scatter(test_X[i], test_Y[j], color=color_map((guess+1) / 2),
            s=200)

plt.scatter(input_pos[:, 0], input_pos[:, 1], c='g')
plt.scatter(input_neg[:, 0], input_neg[:, 1], c='y')

plt.axis([x_min, x_max, y_min, y_max])
plt.show()
```

4.3.2 supervised-model.py

```
import numpy as np
import sys

# Hyper parameters
learning_rate = .1
beta = .5
iterations = 3000


def activation(x, deriv=False):
    if not deriv:
        return np.tanh(beta * x)
    else:
        return beta * (1 - np.tanh(beta * x) ** 2)


def sgn(z):
    sign = lambda x: -1 if x < 0 else 1
    shp = z.shape
    z = np.fromiter((sign(xi) for xi in z), z.dtype)
    z = np.reshape(z, shp)
    return z


class SupervisedModel:
    def __init__(self, n, m, w=None):
        weight_interval = [-1, 1]
        self.n = n
        self.m = m
        if w is None:
            self.w = (weight_interval[1] - weight_interval[0]) *
                np.random.random((n, m)) + weight_interval[0]
        else:
            self.w = w

    def classification_error(self, x, y):
        p = x.shape[0]
        y_val = activation(np.dot(x, self.w))
        err = np.absolute(y - sgn(y_val))
        t = float(1) / (2 * p)
        return t * np.sum(err)

    def train(self, X, Y):
        # Initialize bias with mean 0 in interval [-1, 1]
        bias_interval = [-1, 1]
        b = (bias_interval[1] - bias_interval[0]) *
```

```
            np.random.random((1, self.m)) + bias_interval[0]
        Y = Y.reshape(len(Y), 1)

        for iteration in xrange(iterations):

            # Get a random pattern and its associated target
            length = X.shape[0]
            rand_index = int(np.floor(np.random.random_sample() * length))
            x = X[rand_index, :].reshape(1, self.n)
            y_actual = Y[rand_index, :].reshape(self.m, 1)

            # feed forward
            ys = np.dot(x, self.w) + b
            y = activation(ys)

            delta_i = (y_actual - y) * activation(ys, deriv=True)
            delta_w = learning_rate * np.dot(x.T, delta_i)
            delta_b = learning_rate * delta_i

            b += delta_b
            self.w += delta_w

            sys.stdout.write("Running supervised training epoch %d of
                %d...\r" % (iteration + 1, iterations))
            sys.stdout.flush()
        print ''

    def valid(self, X, Y):
        error = 0
        for i in range(0, len(X)):
            error += self.classification_error(X[i], Y[i])
        return error / len(X)

    def get_guess(self, x):
        return activation(np.dot(x, self.w))[0]
```