2 Kohonen Self-Organized-Map

2.1 What does unsupervised learning learn?

Unsupervised learning is a type of machine learning that looks for previously undetected patterns in a data set with no pre-existing labels and with a minimum of human supervision.In contrast, in unsupervised learning The objective is to find the natural structure inherent in the input data.

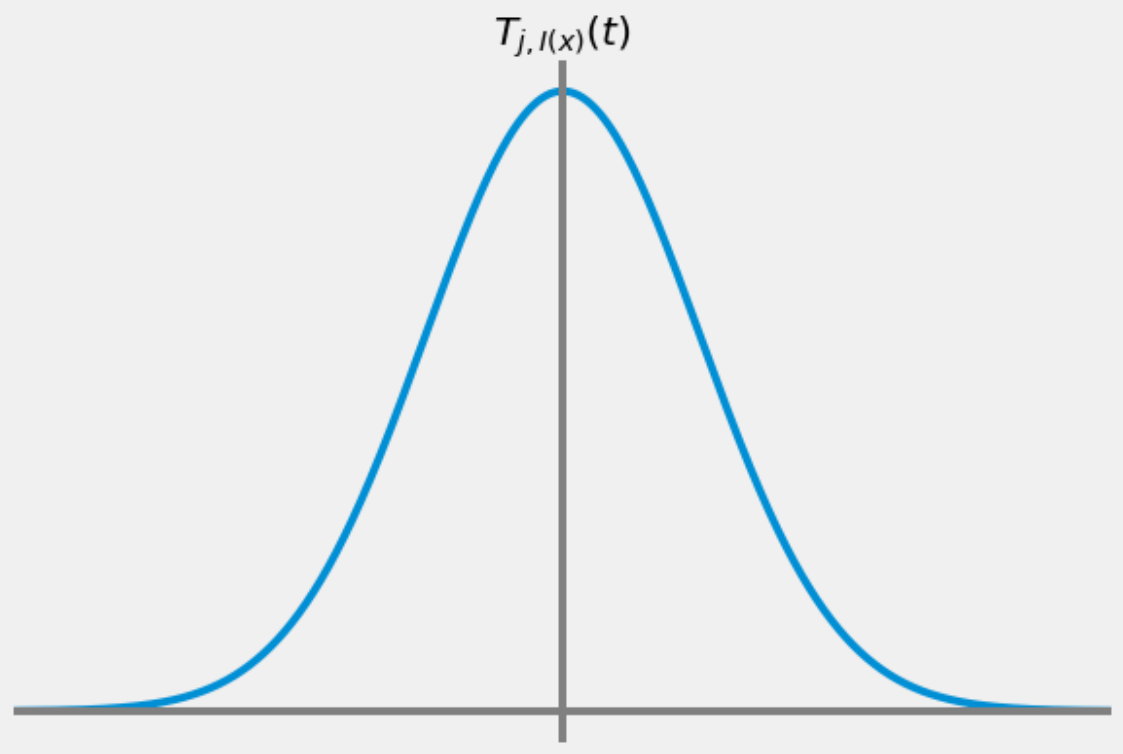
- **Parametric unsupervised** learning assumes a parametric distribution of data. What this means, is that this type of unsupervised learning assumes that the data comes from a population that follows a particular probability distribution based on some parameters.
- **Non-parametric unsupervised learning** refers to the clustering of the input data set. Each cluster, in essence, says something about the categories and classes of the data items present in the set. This is the most commonly used method for data modelling and analyzing data with small sample sizes. These methods are also referred to as distribution-free methods

2.2 Kohonen networks equations

1. $\Delta w_{ji} = \eta(t).T_{j,I(x)}(t).(x_i - w_{ji})$
2. $T_{j,I(x)}(t) = e^{\frac{-S^2_{j,I(x)}}{2\sigma^2(t)}}$

The first equation is a fraction of the difference between the old weight and the input vector x. In other words, the weight vector is 'moved' closer towards the input vector. Another important element to note is that the updated weight will be proportional to the 2D distance between the nodes in the neighbourhood radius and the BMU. Furthermore the updated weight should take into factor that the effect of the learning is close to none at the extremities of the neighbourhood, as the amount of learning should decrease with distance. Therefore, the equation has the extra neighbourhood function factor of $T_{j,I(x)}(t)$.

- S is the distance between BMU and current neuron.
- η is the learning rate which decreases over time.
- T is the influence the BMU has on the selected neuron.(below plot)
- σ is the time constant calculated as $\sigma = \frac{total\ epochs}{log^{(radius)}}$
- x is the input RGB vector.
- w is selected neuron's weight vector.



2.3 Train a Kohonen's Self-Organizing Feature Map

In this question we had to train a *SOFM* to map 3D data to 2D space.

2.3.1 Packages

Below are the packages used to achieve this.

[48]:
```
import numpy as np
import matplotlib.pyplot as plt
```

2.3.2 *SOM*

this class is where everything happens.

- ***map*** is the kohonen layer which is $40 \times 40 \times 3$.
- ***initial_lr*** is initial learning rate.
- ***initial radius*** is the initial radius for weight updates.
- ***epochs*** is the no. of iterations.
- ***landa*** is the time constant.
- ***radius*** is the radius for weight updates which decreases over time.

- ***lr*** is the learning rate which decreases over time.
- ***influence*** is a gaussian formula which detemines the influence the BMU has on a selected neuron.

Functions used in this class are:

- ***compute_landa*** computes time constant as $\sigma = \frac{total\ epochs}{log^{(initial\ radius)}}$
- ***compute_radius*** decays neighbourhood radius each iteration as $radius = initial\ radius * e^{(\frac{iteration}{\sigma})}$
- ***compute_lr*** decays learning rate each iteration as $lr = initial\ lr * e^{(\frac{iteration}{\sigma})}$
- ***compute_influence*** computes the neighbourhood influence using the euclidean distance between BMU and selected neuron as $influence = e^{-\frac{distance}{2*radius^2}}$
- ***compute_distance*** computes euclidean distance between two neurons.
- ***find_BMU*** finds the the neuron that best matches the input based on the euclidean distance between input and the selected neurons weights.(the lower the better)
- ***update_weights*** updates the weights of the neuron within the current radius of the BMU as $w(t+1) = w(t) + influence * learning\ rate * (x_i - w(t))$
- ***train*** is the where the network is trained. At each iteration the new values of learning rate and radius are computed and a new input in the input vector is chosen. Then the BMU is founded and weights are updated according to the above formulas. In the last part kohonen layer's weights are ploted.

[49]:
```
class SOM:
    def __init__(self, lr, epochs):
        self.map = np.random.uniform(0, 1, size=(40, 40, 3))
        self.initial_lr = lr
        self.initial_radius = 20
        self.epochs = epochs
        self.landa = 0
        self.compute_landa()
        self.radius = 0
        self.compute_radius()
        self.lr = 0
        self.compute_lr()
        self.influence = 0

    def compute_landa(self):
        self.landa = self.epochs / np.log(self.initial_radius)

    def compute_radius(self, epoch=0):
        self.radius = self.initial_radius * np.exp(-epoch / self.landa)

    def compute_lr(self, epoch=0):
        self.lr = self.initial_lr * np.exp(-epoch / self.landa)
```

```
    def compute_influence(self, distance):
        return np.exp(-distance / (2 * (self.radius ** 2)))

    def compute_distance(self, x1, y1, x2, y2):
        return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

    def find_BMU(self, x):
        arg_min = (-1, -1)
        dist = float("inf")
        for i in range(len(self.map)):
            for j in range(len(self.map[i])):
                if np.sqrt(np.sum((self.map[i][j] - x) ** 2)) < dist:
                    dist = np.sqrt(np.sum((self.map[i][j] - x) ** 2))
                    arg_min = (i, j)
        return arg_min

    def update_weights(self, minX, minY, x):
        for i in range(len(self.map)):
            for j in range(len(self.map[i])):
                dist = self.compute_distance(i, j, minX, minY)
                if dist < self.radius:
                    influence = self.compute_influence(dist)
                    self.map[i][j] += influence * self.lr * (x - self.map[i][j])

    def train(self, input):
        for epoch in range(self.epochs + 1):
            self.compute_radius(epoch)
            self.compute_lr(epoch)
            x = input[epoch % 1600]
            minX, minY = self.find_BMU(x)
            self.update_weights(minX, minY, x)
            if epoch % 400 == 0:
                plt.subplot(2, 2, (int((epoch + 1) / 400) % 4) + 1)
                plt.axis('off')
                plt.title("Epoch " + str(epoch))
                plt.imshow(self.map)
                if epoch in [1200, 2800]:
                    plt.show()
        plt.subplot(1, 1, 1)
        plt.axis('off')
        plt.title("Final Result")
        plt.imshow(self.map)
        plt.show()
```

2.3.3 *generate_data*

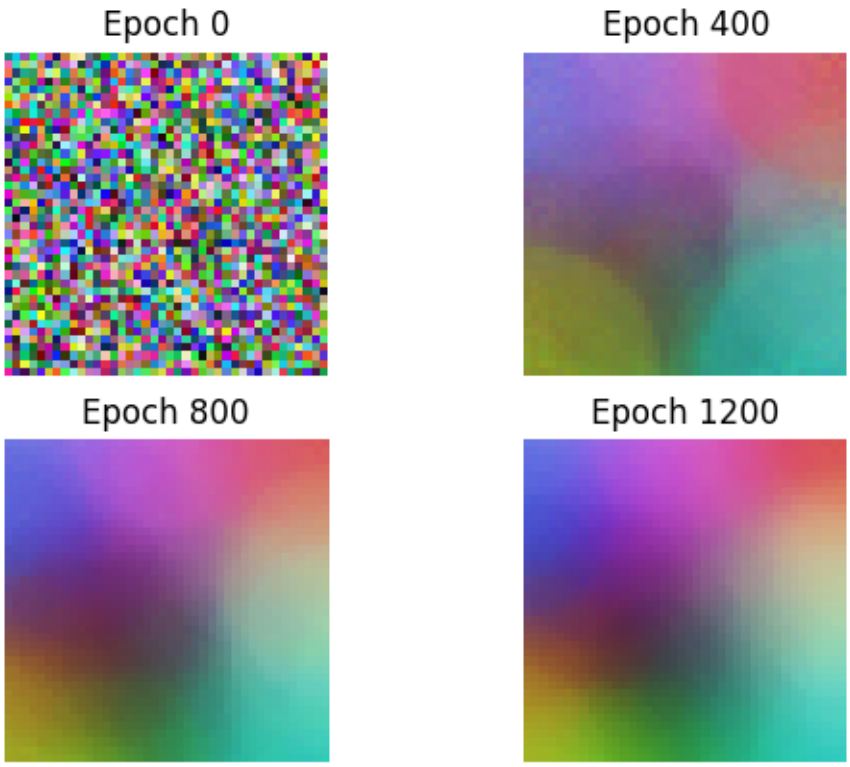
generates a random 1600×3 input.

[50]:
```
def generate_data():
    data = np.ndarray((1600, 3), dtype=float)
    for i in range(len(data)):
        r = np.random.randint(0, 255)
        g = np.random.randint(0, 255)
        b = np.random.randint(0, 255)
        data[i] = [r, g, b]
    return data
```

2.3.4 *main*

Here the network is trained with initial learning rate of 0.04 and 3200 iterations.

[51]:
```
if __name__ == '__main__':
    input = generate_data()
    input = input / input.max()
    som = SOM(0.04, 3200)
    som.train(input)
```



Epoch 1600

Epoch 2000

Epoch 2400

Epoch 2800

Final Result