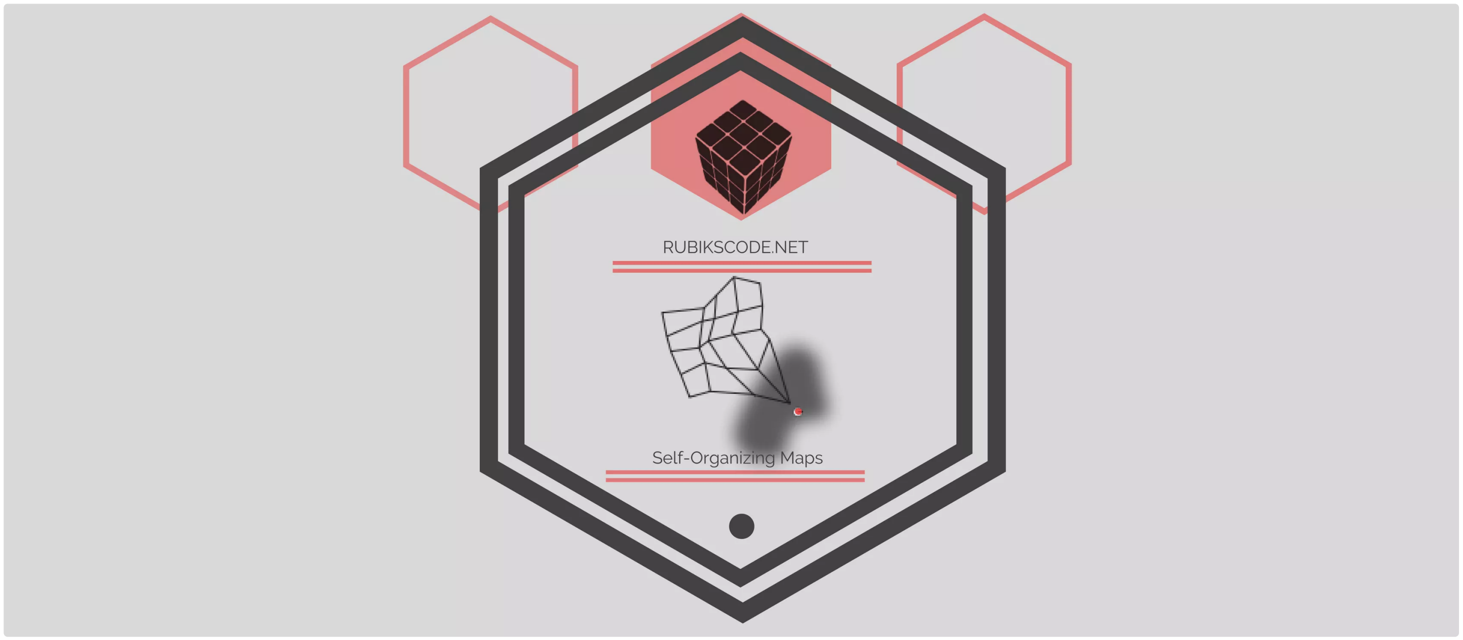
Freedom.
Wisdom.
Excellence.

Implementing Self-Organizing Maps with Python and TensorFlow

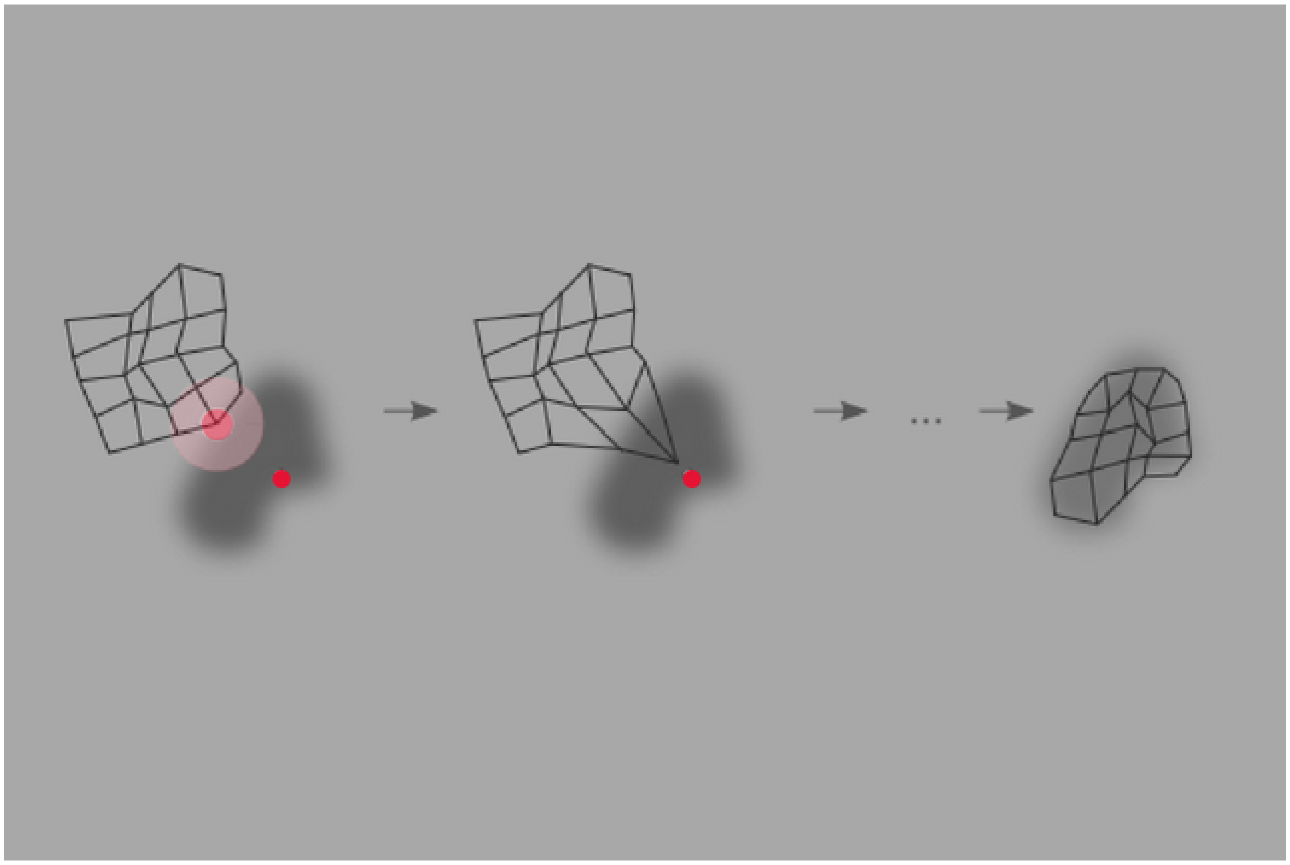
AUGUST 27, 2018 — 2 COMMENTS



In the previous article, we got familiar with the main concepts of Self-Organizing Maps. We explored how they utilize different type of learning than we got a chance to see during our trip through the world of artificial neural networks – unsupervised learning. This is the type of learning in which network doesn't get the expected result for certain input, but it got to figure out inner data relationship on its own. Self-Organizing Maps use this approach for clustering and classification purposes and they are quite good at it.

Another important thing we got a chance to see is that the concepts of neurons, connection and weights are having a different meaning in Self-Organizing Maps world. Neurons are usually organized in two big groups. The first group is a collection of input neurons, and their number corresponds to the number of features that we have in used dataset.

The second group is a collection of output neurons. These neurons are usually organized as one or two-dimensional arrays and are triggered only by certain input values. While here is no concept of locations neuron in artificial neural network, that is not the case with the Self-Organizing Maps. Not only that each neuron has a location, but it is considered that neurons that lie close to each have similar properties and actually represent a cluster.



Every input neuron is connected to every output neuron. The learning process is also different than in standard feed-forward neural networks since unsupervised learning is used. These are the main steps of this process for Self-Organizing Maps:

1. Weight initialization
2. The input vector is selected from the dataset and used as an input for the network
3. BMU is calculated
4. The radius of neighbors that will be updated is calculated
5. Each weight of the neurons within the radius are adjusted to make them more like the input vector
6. Steps from 2 to 5 are repeated for each input vector of the dataset

You can check out the previous article in which this process is explained in details.

There are many existing implementations of Self-Organizing Maps available online, and we will check some of them in the next chapter. However, the idea behind this article is to create our own implementation using TensorFlow and then use it in the future articles for solving the real-world problems.

Existing Implementations

As we already mentioned, there are many available implementations of the Self-Organizing Maps for Python available at PyPI. To name the some:

- SimpleSom
- Kohonen
- MiniSOM

The last implementation in the list – MiniSOM is one of the most popular ones. It is a minimalistic, Numpy based implementation of the Self-Organizing Maps and it is very user friendly. It can be installed using *pip:*

```
pip install minisom
```

or using the downloaded setup:

```
python setup.py install
```

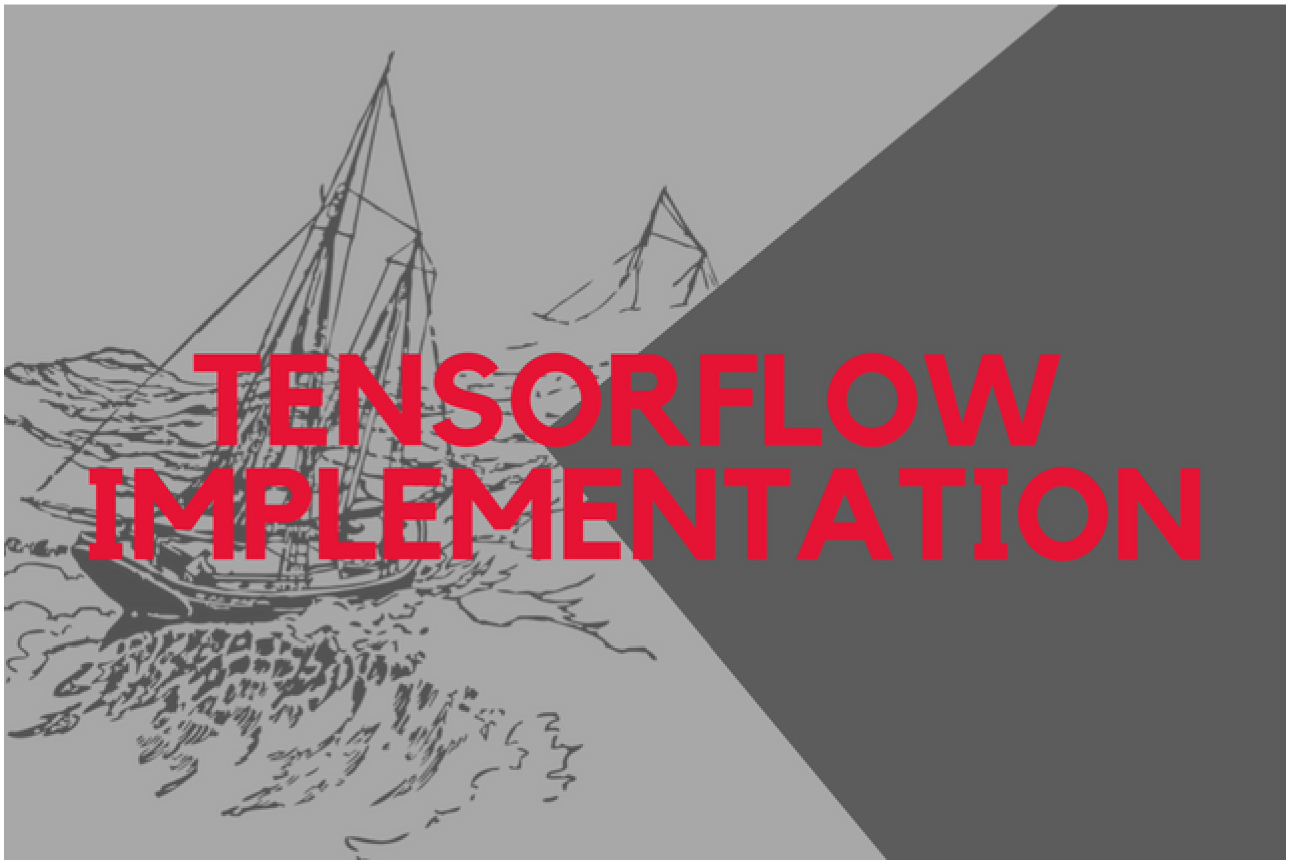
As mentioned, usage of this library is quite easy and straight-forward. In general, all you have to do is create an object of *SOM* class, and define its size, size of the input, learning rate and radius (sigma). After that you can use one of the two options for training that this implementation provides – *train_batch* or *train_random.* The first one uses samples in order in which is recorded in the data set, while the second one shuffles through the samples. Here is an example:

```
from minisom import MiniSom
som = MiniSom(6, 6, 4, sigma=0.5, learning_rate=0.5)
som.train_random(data, 100)
```

minisomuse.py hosted with ❤ by GitHub view raw

In this example, *6×6* Self-Organizing Map is created, with the 4 input nodes (because data set in this example is having 4 features). Learning rate and radius (sigma) are both initialized to 0.5. Than Self-Organizing Map is trained with input data for 100 iterations using *train_random.*

TensorFlow Implementation



For this implementation, TensorFlow 1.10.0 version is used. Here you can find quick guide how to quickly install it and how to start working with it. In general, low level API of this library is used for the implementation. So, lets check out the code:

```
import tensorflow as tf
import numpy as np

class SOM(object):
    def __init__(self, x, y, input_dim, learning_rate, radius, num_iter=111):

        #Initialize properties
        self._x = x
        self._y = y
        self._learning_rate = float(learning_rate)
        self._radius = float(radius)
        self._num_iter = num_iter
        self._graph = tf.Graph()

        #Initialize graph
        with self._graph.as_default():

            #Initializing variables and placeholders
            self._weights = tf.Variable(tf.random_normal([x*y, input_dim]))
            self._locations = self._generate_index_matrix(x, y)
            self._input = tf.placeholder("float", [input_dim])
            self._iter_input = tf.placeholder("float")

            #Calculating BMU
            input_matix = tf.stack([self._input for i in range(x*y)])
            distances = tf.sqrt(tf.reduce_sum(tf.pow(tf.subtract(self._weights, input_matix),
            bmu = tf.argmin(distances, 0)

            #Get BMU location
            mask = tf.pad(tf.reshape(bmu, [1]), np.array([[0, 1]]))
            size = tf.cast(tf.constant(np.array([1, 2])), dtype=tf.int64)
            bmu_location = tf.reshape(tf.slice(self._locations, mask, size), [2])

            #Calculate learning rate and radius
            decay_function = tf.subtract(1.0, tf.div(self._iter_input, self._num_iter))
            _current_learning_rate = tf.multiply(self._learning_rate, decay_function)
            _current_radius = tf.multiply(self._radius, decay_function)

            #Adapt learning rate to each neuron based on position
            bmu_matrix = tf.stack([bmu_location for i in range(x*y)])
            bmu_distance = tf.reduce_sum(tf.pow(tf.subtract(self._locations, bmu_matrix), 2),
            neighbourhood_func = tf.exp(tf.negative(tf.div(tf.cast(bmu_distance, "float32"),
            learning_rate_matrix = tf.multiply(_current_learning_rate, neighbourhood_func)

            #Update all the weights
            multiplytiplier = tf.stack([tf.tile(tf.slice(
                learning_rate_matrix, np.array([i]), np.array([1])), [input_dim])
                                               for i in range(x*y)])
            delta = tf.multiply(
                multiplytiplier,
                tf.subtract(tf.stack([self._input for i in range(x*y)]), self._weights))

            new_weights = tf.add(self._weights, delta)
            self._training = tf.assign(self._weights, new_weights)

            #Initilize session and run it
            self._sess = tf.Session()
            initialization = tf.global_variables_initializer()
            self._sess.run(initialization)

    def train(self, input_vects):
        for iter_no in range(self._num_iter):
            for input_vect in input_vects:
```

```python
                    self._sess.run(self._training,
                                   feed_dict={self._input: input_vect,
                                              self._iter_input: iter_no})

        self._centroid_matrix = [[] for i in range(self._x)]
        self._weights_list = list(self._sess.run(self._weights))
        self._locations = list(self._sess.run(self._locations))
        for i, loc in enumerate(self._locations):
            self._centroid_matrix[loc[0]].append(self._weights_list[i])

    def map_input(self, input_vectors):
        return_value = []
        for vect in input_vectors:
            min_index = min([i for i in range(len(self._weights_list))],
                            key=lambda x: np.linalg.norm(vect - self._weights_list[x]))
            return_value.append(self._locations[min_index])
        return return_value

    def _generate_index_matrix(self, x,y):
        return tf.constant(np.array(list(self._iterator(x, y))))

    def _iterator(self, x, y):
        for i in range(x):
            for j in range(y):
                yield np.array([i, j])
```

somtf.py hosted with ❤ by **GitHub** **view raw**

That is quite a lot of code, so let's dissect it into smaller chunks and explain what each piece means. Majority of the code is in the constructor of class which, similar to the MiniSOM implementation, takes dimensions of the Self-Organizing Map, input dimensions, radius and learning rate as an input parameters. The first thing that is done is initialization of all the fields with the values that are passed into the class constructor:

```python
##Initialize properties
self._x = x
self._y = y
self._learning_rate = float(learning_rate)
self._radius = float(radius)
self._num_iter = num_iter
self._graph = tf.Graph()
```

fieldsinitialization.py hosted with ❤ by **GitHub** **view raw**

Note that we created TensorFlow graph as a *_graph* field. In the next part of the code, we essentially add operations to this graph and initialize our Self-Organizing Map. If you need more information on how TensorFlows graphs and session work, you can find it here. Anyway, the first step that needs to be done is to initialize variables and placeholders:

```
#Initializing variables and placeholders
self._weights = tf.Variable(tf.random_normal([x*y, input_dim]))
self._locations = self._generate_index_matrix(x, y)
self._input = tf.placeholder("float", [input_dim])
self._iter_input = tf.placeholder("float")
```

variables_placeholders.py hosted with ❤ by GitHub view raw

Basically, we created *_weights* as a randomly initialized tensor. In order to easily manipulate the neurons matrix of indexes is created – *_locations*. They are generated by using *_generate_index_matrix*, which looks like this:

```
def _generate_index_matrix(self, x,y):
        return tf.constant(np.array(list(self._iterator(x, y))))

def _iterator(self, x, y):
    for i in range(x):
        for j in range(y):
            yield np.array([i, j])
```

_generate_index_matrix.py hosted with ❤ by GitHub view raw

Also, notice that *_input* (input vector) and *_iter_input* (iteration number, which is used for radius calculations) are defined as placeholders. This is due to the fact that this information is filled during the training phase, not the construction phase. Once all variables and placeholders are initialized, we can start with the Self-Organizing Map learning process algorithm. Firstly, BMU is calculated and it's location is determined:

```
#Calculating BMU
input_matix = tf.stack([self._input for i in range(x*y)])
distances = tf.sqrt(tf.reduce_sum(tf.pow(tf.subtract(self._weights, input_matix), 2), 1))
bmu = tf.argmin(distances, 0)

#Get BMU location
mask = tf.pad(tf.reshape(bmu, [1]), np.array([[0, 1]]))
size = tf.cast(tf.constant(np.array([1, 2])), dtype=tf.int64)
bmu_location = tf.reshape(tf.slice(self._locations, mask, size), [2])
```

get_bmu_and_location.py hosted with ❤ by GitHub view raw

The first part basically calculates the Euclidean distances between all neurons and the input vector. Don't get confused by the first line of this code. In an essence, this input sample vector is repeated and matrix is created, so it can be used for calculations with weights tensor. Once distances are calculated, index of the BMU is returned. This index is used, in the second part of the gist, to get BMU location. We relied on the *slice* function for this. Once that is done, we need to calculate values for learning rate and radius for current iteration. That is done like this:

```
 #Calculate learning rate and radius
decay_function = tf.subtract(1.0, tf.div(self._iter_input, self._num_iter))
_current_learning_rate = tf.multiply(self._learning_rate, decay_function)
_current_radius = tf.multiply(self._radius, decay_function)
```

lrandradius.py hosted with ❤ by GitHub view raw

Variable *decay_function* is created based on iteration number. This "function" is used to determine how much mentioned properties are shrinked in defined iteration. After that, fields *_learning_rate* and *_radius* are updated accordingly. The next step is to create learning rates for all neurons based on iteration number and location in comparison to the BMU location. That is handeled like this:

```
#Adapt learning rate to each neuron based on position
bmu_matrix = tf.stack([bmu_location for i in range(x*y)])
bmu_distance = tf.reduce_sum(tf.pow(tf.subtract(self._locations, bmu_matrix), 2), 1)
neighbourhood_func = tf.exp(tf.negative(tf.div(tf.cast(bmu_distance, "float32"), tf.pow(_curre
learning_rate_matrix = tf.multiply(_current_learning_rate, neighbourhood_func)
```

neuronlr.py hosted with ❤ by GitHub view raw

First matrix of BMU location value is created. Than of the neuron to the BMU is calculated. After that, so called *neighbourhood_func* is created. This function is bacially defining how the weight of concrete neuron will changed. Finally, the weights are updated accordingly and TensorFlow session is initialized and run:

```
#Update all the weights
multiplytiplier = tf.stack([tf.tile(tf.slice(
    learning_rate_matrix, np.array([i]), np.array([1])), [input_dim])
                                   for i in range(x*y)])
delta = tf.multiply(
    multiplytiplier,
    tf.subtract(tf.stack([self._input for i in range(x*y)]), self._weights))

new_weightages = tf.add(self._weights, delta)
self._training = tf.assign(self._weights, new_weightages)

#Initilize session and run it
self._sess = tf.Session()
initialization = tf.global_variables_initializer()
self._sess.run(initialization)
```

updateweights.py hosted with ❤ by GitHub view raw

Apart from *_generate_index_matrix* function that you saw previously, this class has also two important functions – *train* and *map_input.* The first one, as its name suggests, is used to train Self-Organizing Map with proper input. Here is how that function looks like:

```
def train(self, input_vects):
    for iter_no in range(self._num_iter):
        for input_vect in input_vects:
            self._sess.run(self._training,
                           feed_dict={self._input: input_vect,
                                      self._iter_input: iter_no})

    self._centroid_matrix = [[] for i in range(self._x)]
    self._weights_list = list(self._sess.run(self._weights))
    self._locations = list(self._sess.run(self._locations))
    for i, loc in enumerate(self._locations):
        self._centroid_matrix[loc[0]].append(self._weights_list[i])
```

trainsom.py hosted with ❤ by GitHub view raw

Essentially, we have just run defined number of iterations on passed input data. For that we used *_training* operation that we created during class construction. Notice that here placeholders for iteration number and input sample are filled. That is how we run created session with correct data.

The second function that this class has is *map_input*. This function is mapping defined input sample to the correct output. Here is how it looks like:

```
def map_input(self, input_vectors):
    return_value = []
    for vect in input_vectors:
        min_index = min([i for i in range(len(self._weights_list))],
```

```python
                        key=lambda x: np.linalg.norm(vect - self._weights_list[x]))
        return_value.append(self._locations[min_index])
    return return_value
```

mapinput.py hosted with ❤ by **GitHub** view raw

At the end, we got Self-Organizing Map with pretty straight forward API that can be easily used. In the next article, we will use this class to solve one real-world problem. To sum it up, it can be used something like this:

```python
from somtf import SOM

som = SOM(6, 6, 4, 0.5, 0.5, 100)
som.train(data)
```

using_somtf.py hosted with ❤ by **GitHub** view raw

As you can see we tried to keep API very similar to the one from MiniSOM implementation.

Conclusion

In this article we learned how to implement Self-Organizing map algorithm using TensorFlow. We used flexibility of the lower level API so to get in even more details of their learning process and got comfortable with it. To sum it up, we applied all theoretical knowledge that we learned in the previous article. Apart from that, we saw how we can use already available Self-Organizing implementations, namely MiniSOM. Next step would be using this implementation to solve some real-world problems, which we will do in the future.

Thank you for reading!

This article is a part of Artificial Neural Networks Series, which you can check out **here**.

Read more posts from the author at **Rubik's Code**.
