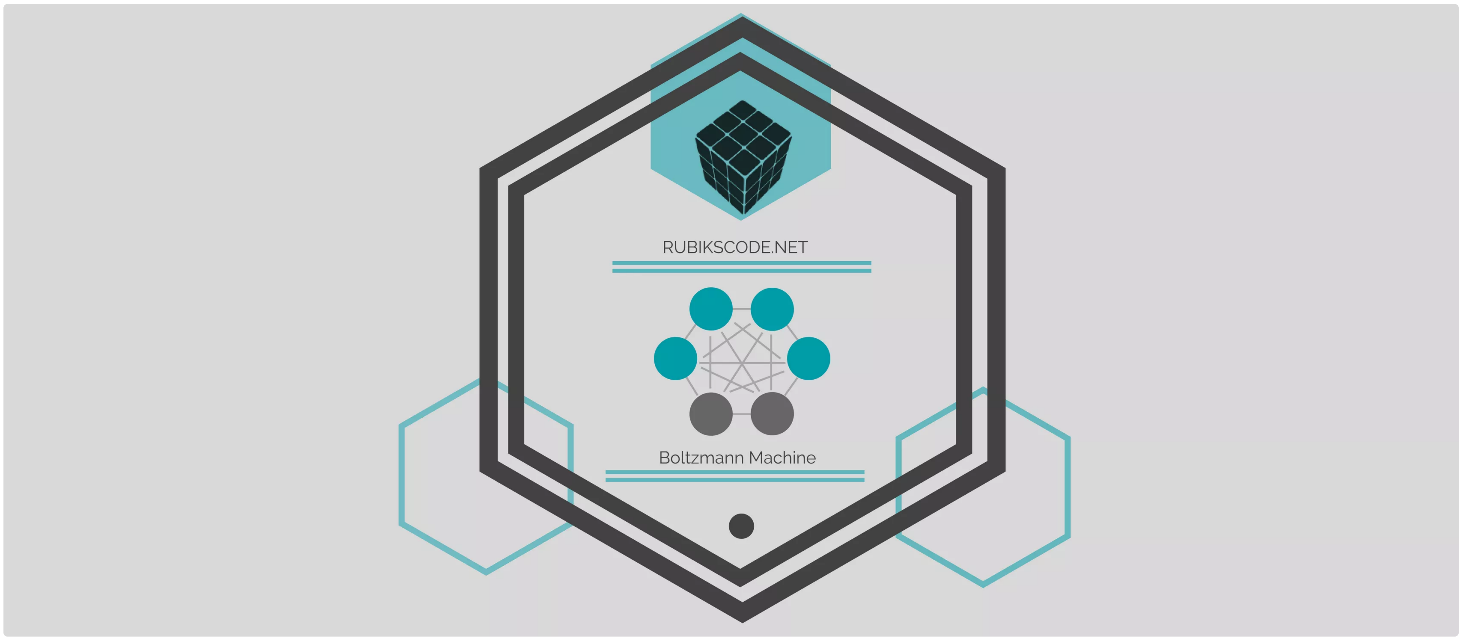
RUBIK'S CODE Freedom. Wisdom. Excellence.

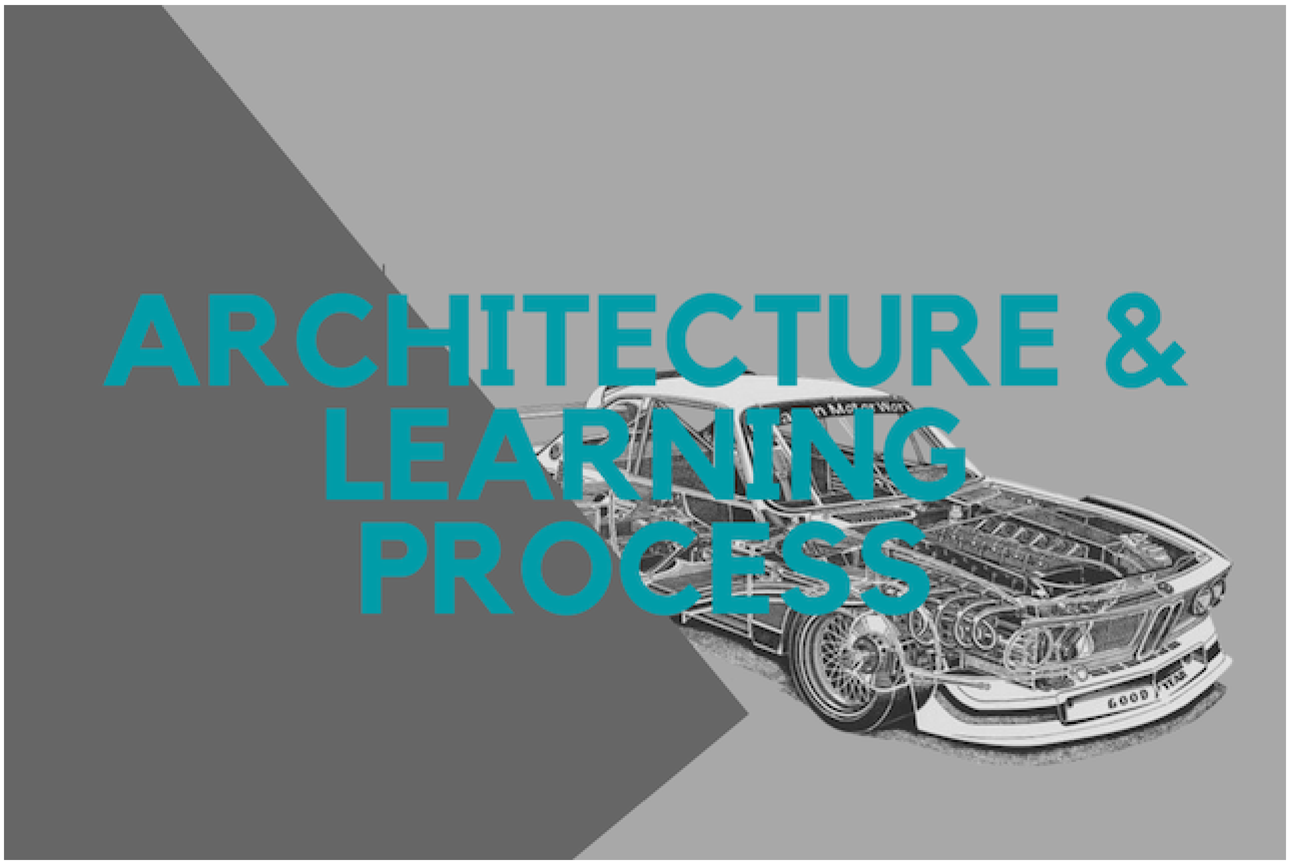
Implementing Restricted Boltzmann Machine with Python and TensorFlow

OCTOBER 22, 2018 — LEAVE A COMMENT



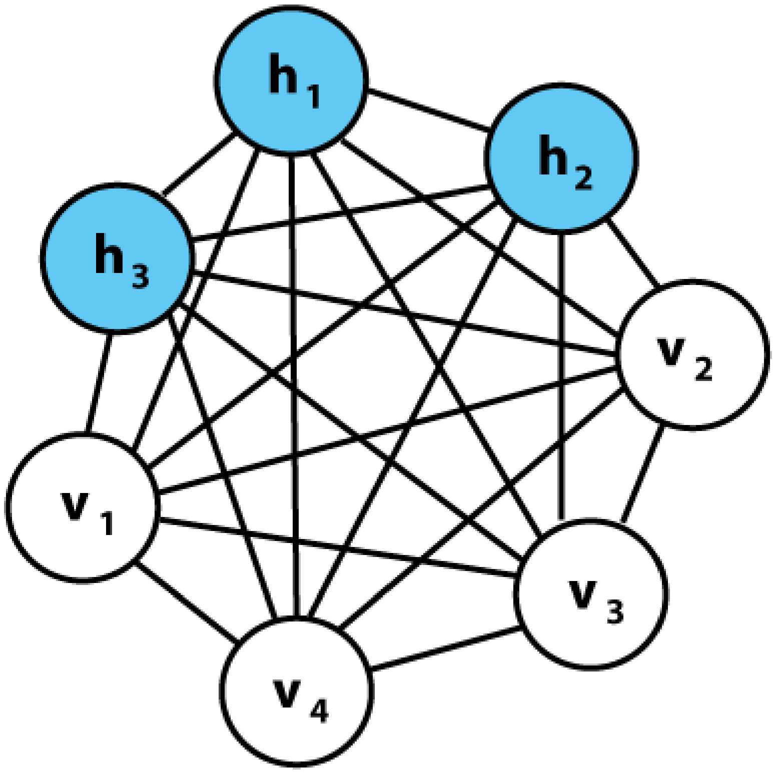
In one of the previous articles, we started learning about Restricted Boltzmann Machine. It was quite a journey since we first had to figure out what energy-based models are, and then to find out how a standard Boltzmann Machine functions. Finally, we discovered the Restricted Boltzmann Machine, an optimized solution which has great performances. At the same time, we touched the subject of Deep Belief Networks because Restricted Boltzmann Machine is the main building unit of such networks. Let's sum up what we have learned so far.

Architecture and Learning Process



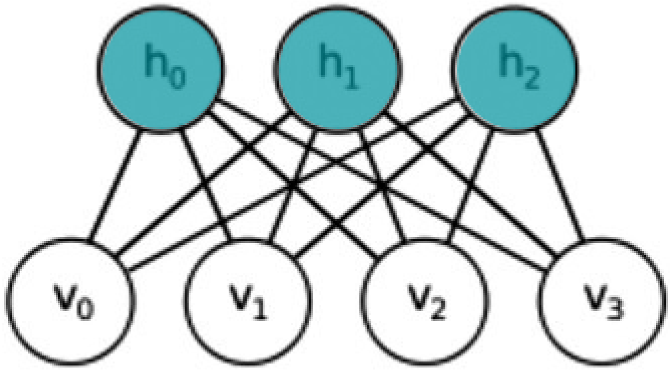
Energy-Based Models are a set of deep learning models which utilize physics concept of energy. They determine dependencies between variables by associating a scalar value, which represents the energy to the complete system. To be more precise, this scalar value actually represents a measure of the probability that the system will be in a certain state. The Boltzmann Machine is just one type of Energy-Based Models.

They consist of symmetrically connected neurons. These neurons have a binary state, i.e they can be either on or off. The decision regarding the state is made stochastically. Since all neurons are connected to each other, calculating weights for all connections is resource-demanding, so this architecture needed to be optimized.



In the end, we ended up with the Restricted Boltzmann Machine, an architecture which has two layers of neurons – visible and hidden, as you can see on the image below. The hidden neurons are connected only to the visible ones and vice-versa, meaning there are no connections between layers in the same layer. This architecture is simple and pretty flexible. It is important to note that data can go both ways, from the visible layer to hidden, and vice-versa.

The learning process of the Restricted Boltzmann Machine is separated into two big steps: Gibbs Sampling and Contrastive Divergence. You can find more on the topic in this article. However, we will run through it either way. First, we need to calculate the probabilities that neuron from the hidden layer is activated based on the input values on the visible layer – Gibbs Sampling. Using this value, we will either turn the neuron on or not.



After Gibbs Sampling is performed, we will use the Contrastive Divergence to update the weights. This process is a bit tricky to explain, so I decided to give it a full chapter in this article. We will use a simple example that will hopefully simplify this explanation. Let’s consider the situation in which we have the visible layer with four nodes in the visible layer and a hidden layer with three nodes. For example, let’s say that input values on the visible layer are [0, 1, 1, 0].

Using the formulas from this article, we will calculate the activation probability for each neuron in the hidden layer. If this probability is high, the neuron from the hidden layer will be activated; otherwise, it will be off. For example, based on current weights and biases we get that values of the hidden layer are [0, 1, 1]. This is the moment when we calculate the so-called positive gradient using the outer product of layer neuron states [0, 1, 1, 0] and the hidden layer neuron states [0, 1, 1]. Outer product is defined like this:

```
v[0]*h[0] v[0]*h[1] v[0]*h[2]
v[1]*h[0] v[1]*h[1] v[1]*h[2]
v[2]*h[0] v[2]*h[1] v[2]*h[2]
v[3]*h[0] v[3]*h[1] v[3]*h[2]
```

where *v* represents a neuron from the visible layer and *h* represents a neuron from the hidden layer. As a result, we get these values for our example:

```
0 0 0
0 1 1
0 1 1
0 0 0
```

This matrix is actually corresponding to all connections in this system, meaning that the first element can be observed as some kind of property or action on the connection between *v[0]* and *h[0]*. In fact, it is exactly that! Wherever we have value 1 in the matrix we add the learning rate to the weight of the connection between two neurons. So, in our example we will do so for connections between *v[1]h[1], v[1]h[2], v[2]h[1]* and *v[2]h[2].*

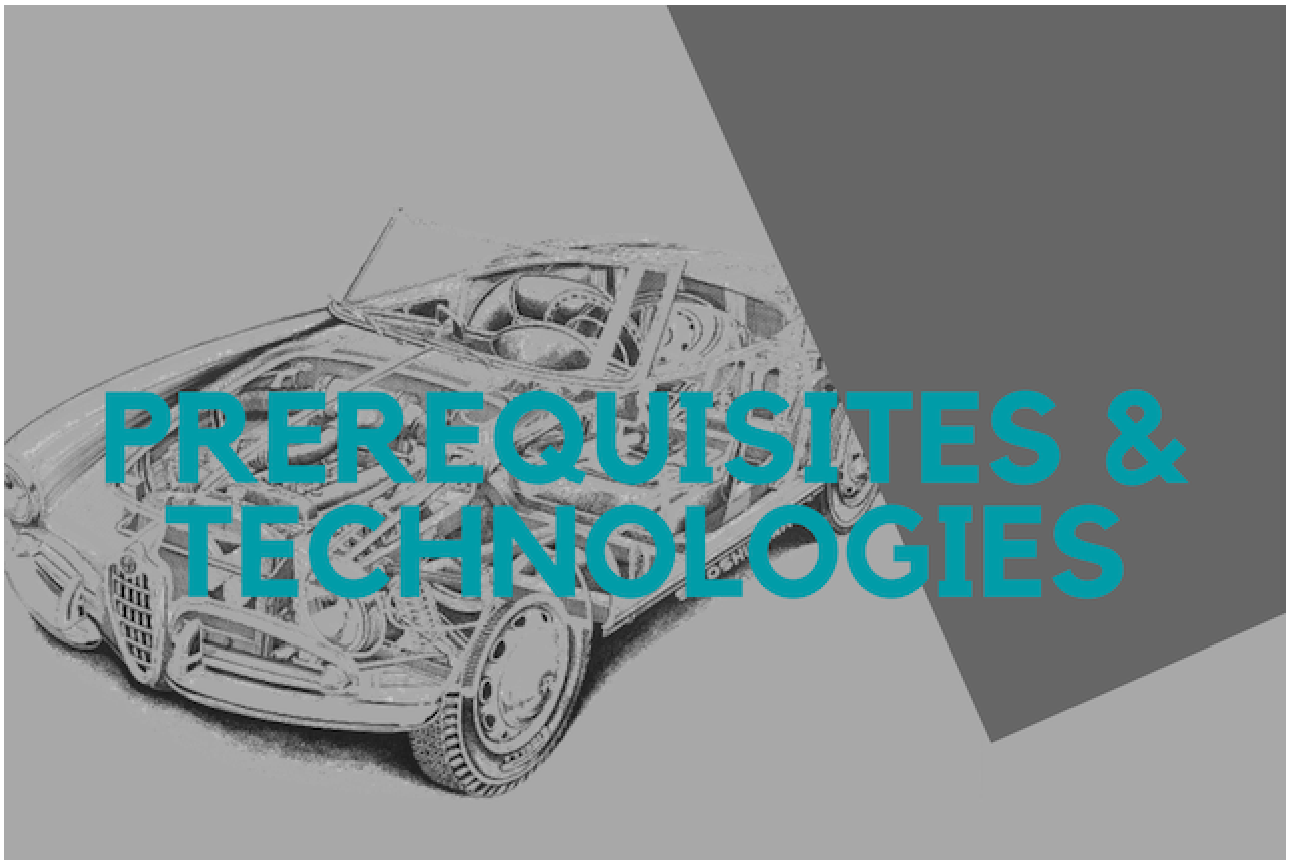
Awesome! We performed the first step in this mystical Contrastive Divergence process. Now, we are once again using formulas from this article to calculate probabilities for the neurons in the visible layer, using values from the hidden layer. Based on these probabilities we calculate the temporary Contrastive Divergence states for the visible layer – *v'[n].* For example, we get the values [0, 0, 0, 1]. Finally, we calculate probabilities for the neurons in the hidden layer once again, only this time we use the Contrastive Divergence states of the visible layer calculated previously. We calculate the Contrastive Divergence states for the hidden layer – – *h'[n],* and for this example get the results [0, 0, 1].

This time we use the outer product of visible layer neuron Contrastive Divergence states [0, 0, 0, 1] and hidden layer neuron states [0, 0, 1] to get this so-called negative gradient:

```
0 0 0
0 0 0
0 0 0
0 0 1
```

Similarly to the previous situation, wherever we have value 1 in this matrix we will subtract the learning rate to the weight between two neurons. So, in our example, we will subtract the learning rate from the weights of the connection between neurons *v[4]h[3].*

Prerequisites & Technologies

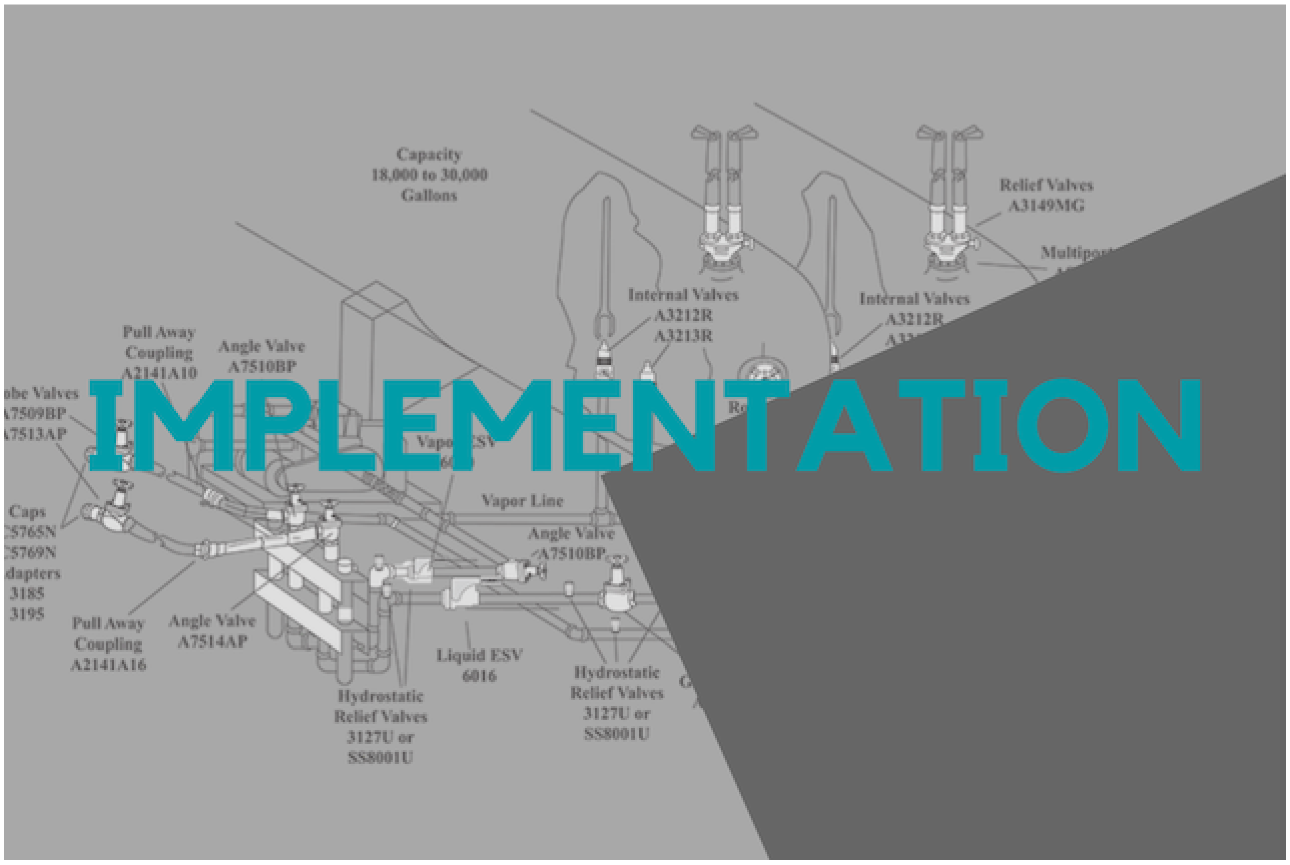


For this implementation, we use these technologies:

- Python 3.6
- TensorFlow 1.10.0
- Spyder IDE

Here you can find a simple guide on how to quickly install TensorFlow and start working with it. As mentioned before, we use Spyder IDE because it is quite good for demonstration purposes. If you find it more convenient, you can use Jupyter as well.

Implementation



Implementation of the Restricted Boltzmann Machine is inside of RBM class. This class has a constructor, *train* method, and one helper method *callculate_state.* Here it is:

```
import tensorflow as tf

class RBM(object):
    def __init__(self, visible_dim, hidden_dim, learning_rate, number_of_iterations):

        self._graph = tf.Graph()

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

            self._num_iter = number_of_iterations
            self._visible_biases = tf.Variable(tf.random_uniform([1, visible_dim], 0, 1, name
            self._hidden_biases = tf.Variable(tf.random_uniform([1, hidden_dim], 0, 1, name =
            self._hidden_states = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name = "h
            self._visible_cdstates = tf.Variable(tf.zeros([1, visible_dim], tf.float32, name
            self._hidden_cdstates = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name =
            self._weights = tf.Variable(tf.random_normal([visible_dim, hidden_dim], 0.01), na
            self._leraning_rate =  tf.Variable(tf.fill([visible_dim, hidden_dim], learning_ra

            self._input_sample = tf.placeholder(tf.float32, [visible_dim], name = "input_samp

            # Gibbs Sampling
            input_matrix = tf.transpose(tf.stack([self._input_sample for i in range(hidden_di
            _hidden_probabilities = tf.sigmoid(tf.add(tf.multiply(input_matrix, self._weights
            self._hidden_states = self.callculate_state(_hidden_probabilities)
            _visible_probabilities = tf.sigmoid(tf.add(tf.multiply(self._hidden_states, self._
            self._visible_cdstates = self.callculate_state(_visible_probabilities)
            self._hidden_cdstates = self.callculate_state(tf.sigmoid(tf.multiply(self._visibl

            #CD
            positive_gradient_matrix = tf.multiply(input_matrix, self._hidden_states)
            negative_gradient_matrix = tf.multiply(self._visible_cdstates, self._hidden_cdsta

            new_weights = self._weights
            new_weights.assign_add(tf.multiply(positive_gradient_matrix, self._leraning_rate)
            new_weights.assign_sub(tf.multiply(negative_gradient_matrix, self._leraning_rate)

            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:
                self._sess.run(self._training,
                               feed_dict={self._input_sample: input_vect})

    def callculate_state(self, probability):
        return tf.floor(probability + tf.random_uniform(tf.shape(probability), 0, 1))


```

rbmtf.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. The majority of the code is in the constructor of the class, which takes dimensions of the hidden and visible layer, learning rate and a number of iterations as input parameters. The first thing we do inside of the constructor is the creation of the graph. After that we initialize variables and placeholders:

```
self._num_iter = number_of_iterations
self._visible_biases = tf.Variable(tf.random_uniform([1, visible_dim], 0, 1, name = "visible_
self._hidden_biases = tf.Variable(tf.random_uniform([1, hidden_dim], 0, 1, name = "hidden_bia
self._hidden_states = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name = "hidden_biases
self._visible_cdstates = tf.Variable(tf.zeros([1, visible_dim], tf.float32, name = "visible_b
self._hidden_cdstates = tf.Variable(tf.zeros([1, hidden_dim], tf.float32, name = "hidden_bias
self._weights = tf.Variable(tf.random_normal([visible_dim, hidden_dim], 0.01), name="weights"
self._leraning_rate =  tf.Variable(tf.fill([visible_dim, hidden_dim], learning_rate), name =

self._input_sample = tf.placeholder(tf.float32, [visible_dim], name = "input_sample")
```

variablesrmbtf.py hosted with ❤ by GitHub view raw

We define biases, states and temporary states for the Contrastive Divergence. Note that states of the visible items are defined by the input array. Apart from that, the weights matrix and learning rate matrix are defined.

Then we perform Gibs Sampling like this:

```
input_matrix = tf.transpose(tf.stack([self._input_sample for i in range(hidden_dim)]))
_hidden_probabilities = tf.sigmoid(tf.add(tf.multiply(input_matrix, self._weights), tf.stack([
self._hidden_states = self.callculate_state(_hidden_probabilities)
_visible_probabilities = tf.sigmoid(tf.add(tf.multiply(self._hidden_states, self._weights), tf
self._visible_cdstates = self.callculate_state(_visible_probabilities)
self._hidden_cdstates = self.callculate_state(tf.sigmoid(tf.multiply(self._visible_cdstates, s
```

gibs_sampling.py hosted with ❤ by GitHub view raw

As we described previously, first we calculate the possibilities for the hidden layer based on the input values and values of the weights and biases. Based on that probability, with the help of *calculate_state* function, we get the states of the hidden layer. After that probability for the visible layer is calculated, and temporary Contrastive Divergence states for the visible layer are defined. Then the process is done for the Contrastive Divergence states of the hidden layer as well. Once this is performed we can calculate the positive and negative gradient and update the weights. Also, we define _*training* operation:

```
positive_gradient_matrix = tf.multiply(input_matrix, self._hidden_states)
negative_gradient_matrix = tf.multiply(self._visible_cdstates, self._hidden_cdstates)

new_weights = self._weights
new_weights.assign_add(tf.multiply(positive_gradient_matrix, self._leraning_rate))
new_weights.assign_sub(tf.multiply(negative_gradient_matrix, self._leraning_rate))

self._training = tf.assign(self._weights, new_weights)
```

update_weights.py hosted with ❤ by GitHub view raw

The final step in the constructor of the class is the initialization of the global variables:

```
self._sess = tf.Session()
initialization = tf.global_variables_initializer()
self._sess.run(initialization)
```

global_var_rbmtf.py hosted with ❤ by GitHub view raw

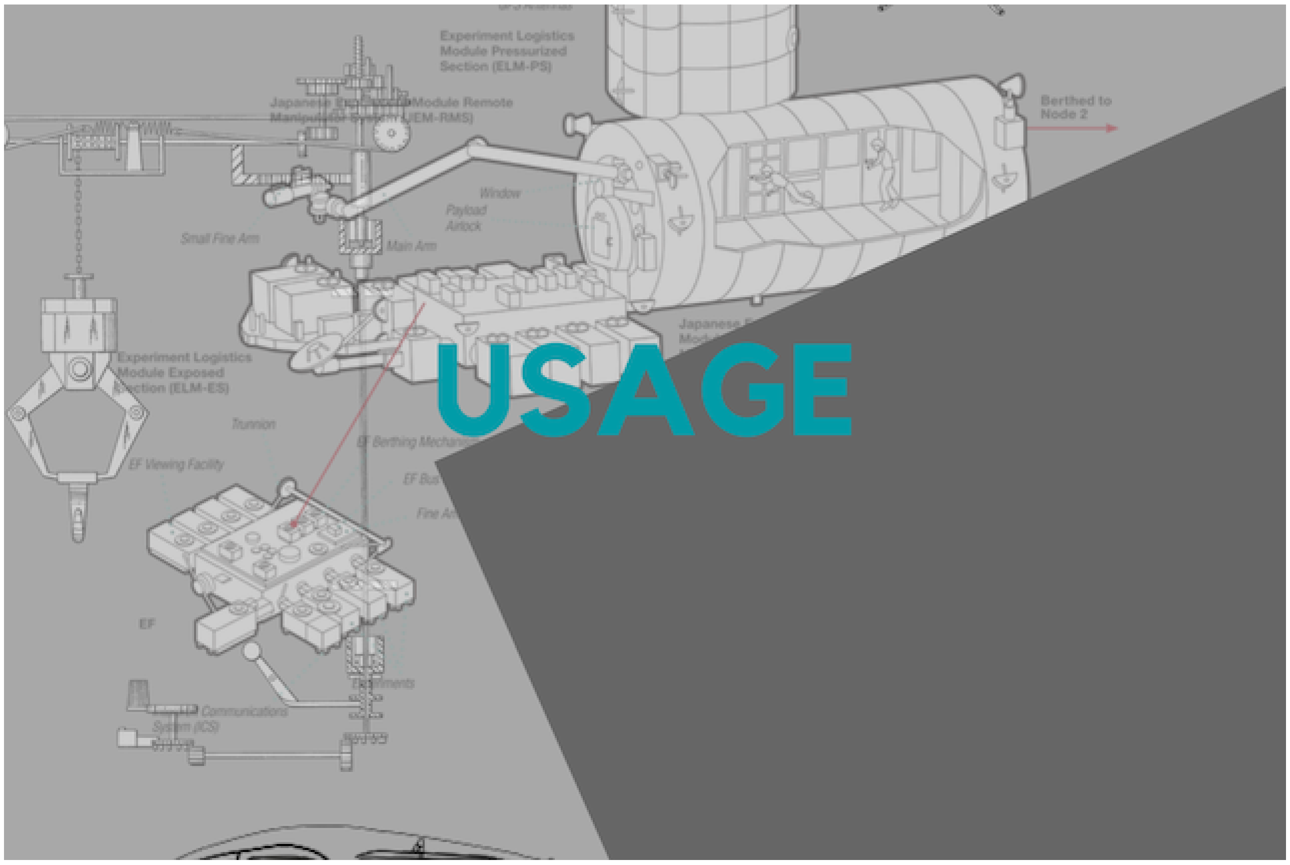
Now, we can examine the *train* method:

```
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_sample: input_vect})
```

train_rbmtf.py hosted with ❤ by GitHub view raw

Here we get an input dataset and we iterate through it. For each array of data in the dataset, we run the training operation in the session. This is done for the number of iterations defined inside of the constructor of the class. This way Restricted Boltzmann Machine is fully trained.

Usage



It is quite easy to use this class we created. Here is an example of how we can use it:

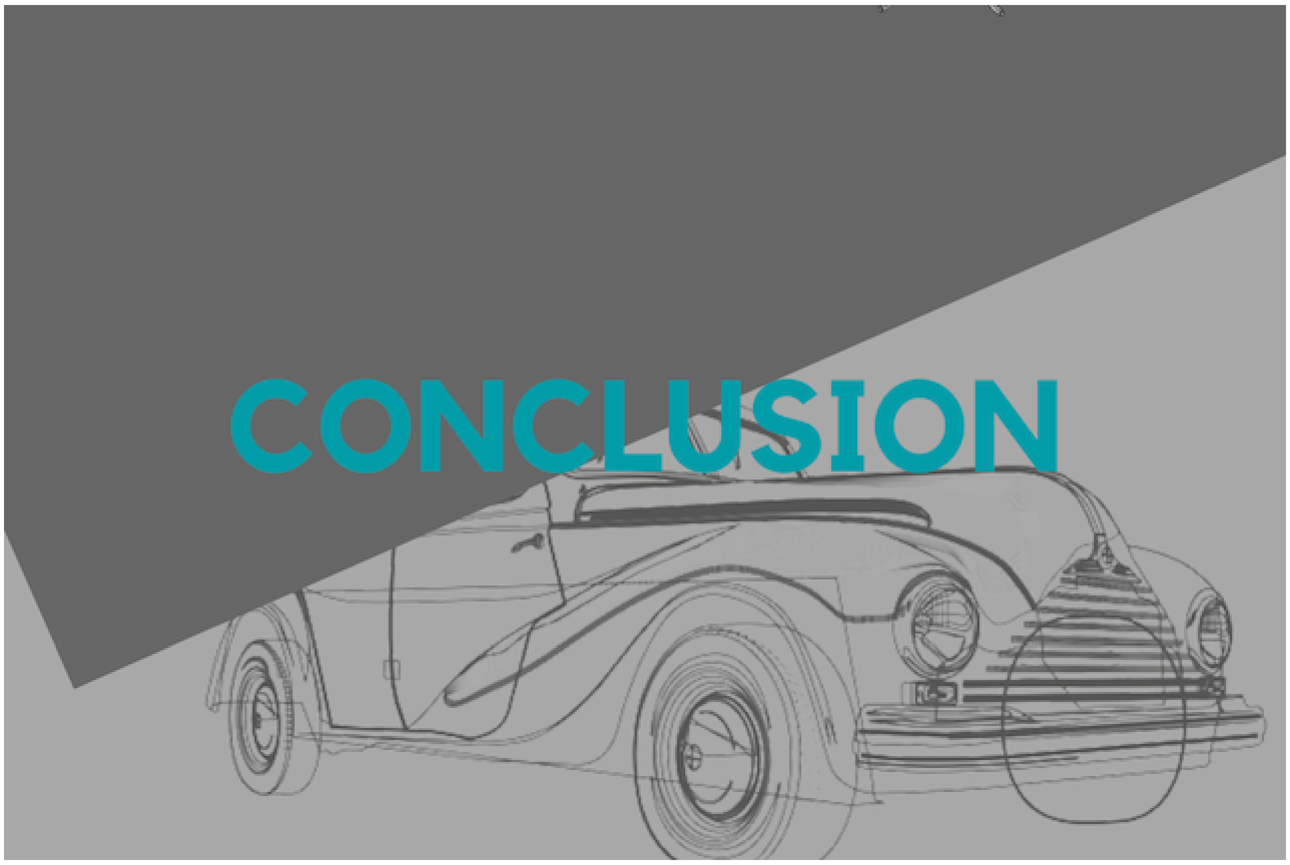
```python
from rbmtf import RBM
import numpy as np

test = np.array([[0,1,1,0], [0,1,0,0], [0,0,1,1]])
rbm = RBM(4, 3, 0.1, 100)
rbm.train(test)
```

rbmtf_usage.py hosted with ❤ by GitHub view raw

First, we import *RBM* from the module and we import *numpy*. With *numpy* we create an array which we call *test*. Then, an object of *RBM* class is created. This object represents our Restricted Boltzmann Machine. To follow the example from the beginning of the article, we use 4 neurons for the visible layer and 3 neurons for the hidden layer. We define values 0.1 and 100 for the learning rate and the number of iterations respectively. Finally, we initiate *train* method and pass *test* array as the input dataset. Of course, in practice, we would have a larger set of data, as this is just for demonstration purposes.

Conclusion



In this article, we learned how to implement the Restricted Boltzmann Machine algorithm using TensorFlow. We used the flexibility of the lower level API to get even more details of their learning process and get comfortable with it. Of course, this is not the complete solution. You can find a more comprehensive and complete solution here. To sum it up, we applied all the theoretical knowledge that we learned in the previous article. The 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**.
