МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСиС»**

Институт ИТКН

Кафедра инженерной кибернетики

Направление подготовки: «01.03.04 Прикладная математика»

Квалификация: бакалавр

Группа: БПМ-17-1

**ОТЧЕТ**

**ПО КУРСОВОЙ РАБОТЕ**

**«ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ»**

на тему: «Самоорганизующиеся карты Кохонена. Структура. Краткое описание алгоритма SOM.»

**Студент** \_\_\_\_\_\_\_\_\_\_/ Рыжков А.П.

**Руководитель** \_\_\_\_\_\_\_\_\_\_/ старший преподаватель, Кондыбаева А.Б.

**Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Москва 2020**

**Содержание**

1. Введение 3

2. Теоретическая часть 4

2.1. Самоорганизующаяся карта Кохонена 4

2.2. Алгоритм обучения SOM 7

2.3. Эвристики алгоритма обучения SOM 13

3. Реализация модели 14

3.1. Используемые программные средства 14

3.2. Описание реализации ПО 14

3.3. Тестирование модели 17

4. Заключение 20

5. Список использованных источников 21

Приложение. 22

1. **Введение**

Обучение нейронной сети – это важный этап ее функционирования. Один из эффективных методов обучения – использование самоорганизующихся карт Кохонена. они нашли широкое применение в различных областях. Метод был предложен финским учёным Теуво Кохоненом в 1984 году. Существует множество модификаций исходной модели.

Самоорганизующаяся карта Кохонена— нейронная сеть с обучением без учителя, выполняющая задачу визуализации и кластеризации. Является методом проецирования многомерного пространства в пространство с более низкой размерностью (чаще всего, двумерное), применяется также для решения различных задач:

* моделирования,
* прогнозирования,
* выявление наборов независимых признаков,
* поиска закономерностей в больших массивах данных,
* разработке компьютерных игр,
* квантизации цветов к их ограниченному числу индексов в цветовой палитре:
* при печати на принтере и ранее на ПК или же на приставках с дисплеем с пониженным числом цветов,
* для архиваторов или видео-кодеков, и прочего.

1. **Теоретическая часть**
   1. **Самоорганизующаяся карта Кохонена**

Для начала разберемся что из себя представляет карта самоорганизации (Self-orginizing Map), или просто SOM. SOM – это искусственная нейронная сеть основанная на обучении без учителя. В картах самоорганизации нейроны помещены в узлах решетки, обычно одно- или двумерной.

Все нейроны этой решетки связаны со всеми узлами входного слоя.

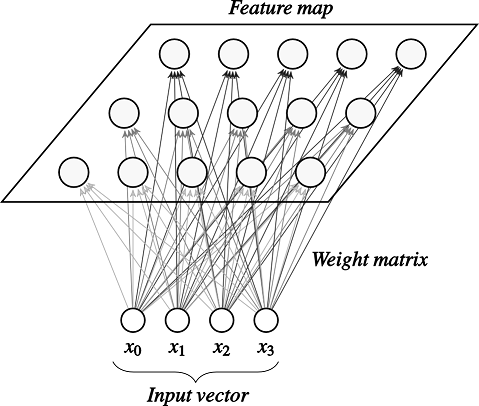


Рисунок 1. Изображение SOM

SOM преобразует непрерывное исходное пространство X в дискретное выходное пространство A.

Самоорганизующаяся карта состоит из компонентов, называемых узлами или нейронами. Их количество задаётся аналитиком. Каждый из узлов описывается двумя векторами. Первый — т. н. вектор веса *m*, имеющий такую же размерность, что и входные данные. Второй — вектор *r*, представляющий собой координаты узла на карте. Карта Кохонена визуально отображается с помощью ячеек прямоугольной или шестиугольной формы; последняя применяется чаще, поскольку в этом случае расстояния между центрами смежных ячеек одинаковы, что повышает корректность визуализации карты.

Изначально известна размерность входных данных, по ней некоторым образом строится первоначальный вариант карты. В процессе обучения векторы веса узлов приближаются к входным данным. Для каждого наблюдения (семпла) выбирается наиболее похожий по вектору веса узел, и значение его вектора веса приближается к наблюдению. Также к наблюдению приближаются векторы веса нескольких узлов, расположенных рядом, таким образом если в множестве входных данных два наблюдения были схожи, на карте им будут соответствовать близкие узлы. Циклический процесс обучения, перебирающий входные данные, заканчивается по достижении картой допустимой (заранее заданной аналитиком) погрешности, или по совершении заданного количества итераций. Таким образом, в результате обучения карта Кохонена классифицирует входные данные на кластеры и визуально отображает многомерные входные данные в двумерной плоскости, распределяя векторы близких признаков в соседние ячейки и раскрашивая их в зависимости от анализируемых параметров нейронов.

В результате работы алгоритма получаются следующие карты:

* **карта входов нейронов** — визуализирует внутреннюю структуру входных данных путём подстройки весов нейронов карты. Обычно используется несколько карт входов, каждая из которых отображает один из них и раскрашивается в зависимости от веса нейрона. На одной из карт определенным цветом обозначают область, в которую включаются приблизительно одинаковые входы для анализируемых примеров.
* **карта выходов нейронов** — визуализирует модель взаимного расположения входных примеров. Очерченные области на карте представляют собой кластеры, состоящие из нейронов со схожими значениями выходов.
* **специальные карты** — это карта кластеров, полученных в результате применения алгоритма самоорганизующейся карты Кохонена, а также другие карты, которые их характеризуют.
  1. **Алгоритм обучения SOM**

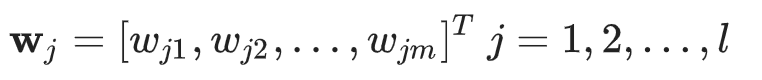
Обучение сети состоит из трех основных процессов:

* конкуренция;
* кооперация;
* адаптация.

Ниже описаны все шаги алгоритма обучения SOM.

**Шаг 1: Инициализация.**

Для всех векторов синаптических весов wj:



где l — общее количество нейронов, m — размерность входного пространства, выбирается случайное значение от -1 до 1.  
  
**Шаг 2: Подвыборка.**

Выбираем вектор x из входного пространства:

Изображение выглядит как текст

Автоматически созданное описание

**Шаг 3: Поиск победившего нейрона или процесс конкуренции.**

Находим наиболее подходящий (победивший нейрон)  на шаге n, используя критерий минимума Евклидова расстояния (что эквивалентно максимуму скалярных произведений):

Изображение выглядит как текст

Автоматически созданное описание

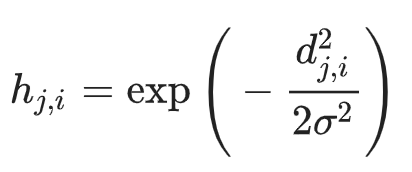
где Евклидово расстояние (ℓ2norm) определятся как:

Изображение выглядит как текст, часы

Автоматически созданное описание

**Шаг 4: Процесс кооперации.**

Нейрон-победитель находится в центре топологической окрестности «сотрудничающих» нейронов. Следующая задача состоит в определении топологической окрестности (topological neighbourhood для победившего нейрона. Для удобства обозначим ее символом: hj,i, с центром в победившем нейроне i. Топологическая окрестность должна быть симметричной относительно точки максимума, определяемой при dj,i = 0, dj,i — это латеральное расстояние (lateral distance) между победившим i и соседними нейронами j.  
 Типичным примером, удовлетворяющим условию выше, hj,i является функция Гаусса:



где σ — эффективная ширина (effective width). Латеральное расстояние определяется как:

Изображение выглядит как текст

Автоматически созданное описание

в одномерном, и:

Изображение выглядит как текст

Автоматически созданное описание

в двумерном случае, Где rj определяет позицию возбуждаемого нейрона, а ri — позицию победившего нейрона (в случае двумерной решетки r=(x,y), где x и y координаты нейрона в решетке).

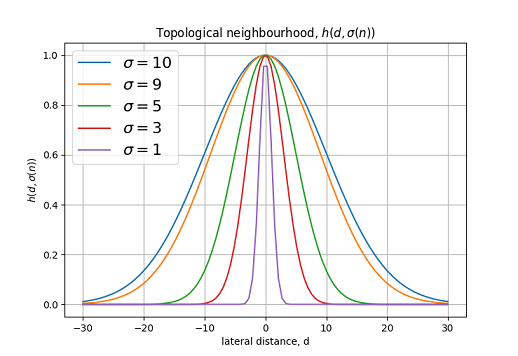
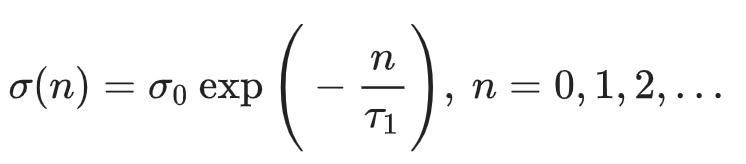


Рисунок 2. График функции топологической окрестности для различных σ

Для SOM характерно уменьшение топологической окрестности в процессе обучения. Достичь этого можно изменяя σ по формуле:



где τ1 — некоторая константа, n — шаг обучения, σ0 — начальное значение σ.

Покажем график изменения σ в процессе обучения:

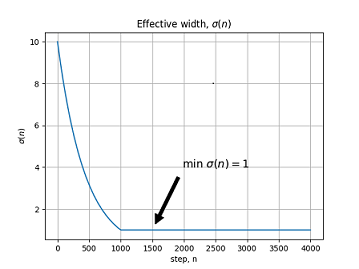


Рисунок 3. График изменением σ в процессе обучения.

Функция hj,i по окончании этапа обучения должна охватывать только ближайших соседей. На рисунках ниже приведены графики функции топологической окрестности для двумерной решетки.

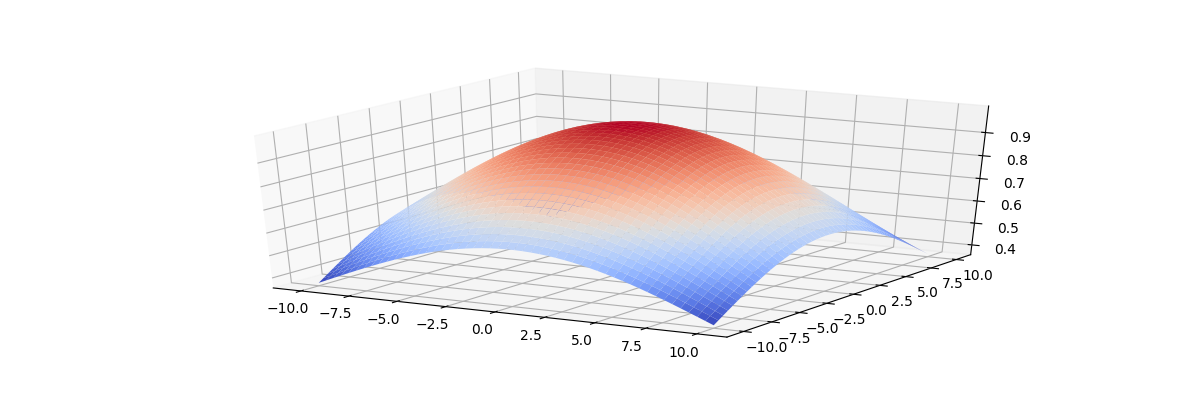


Рисунок 4. Топологическая окрестность в начале обучения.

Из данного рисунка видно, что в начале обучения топологическая окрестность охватывает практически всю решетку.

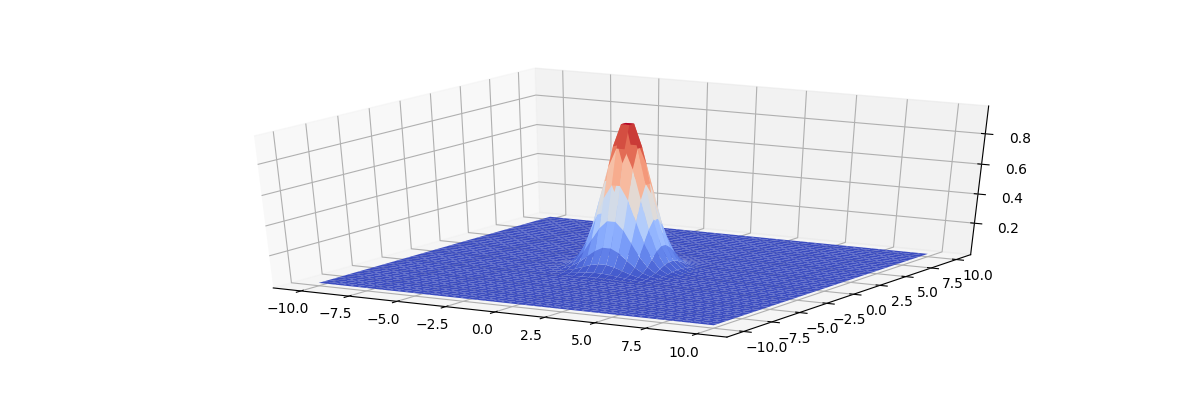


Рисунок 5. Топологическая окрестность в конце обучения.

В конце обучения hj,i сужается до ближайших соседей.

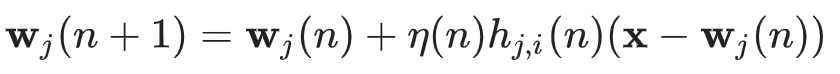
**Шаг 5: Процесс адаптации.**

Процесс адаптации включает в себя изменение синаптических весов сети. Изменение вектора весов нейрона j в решетке можно выразить следующим образом:

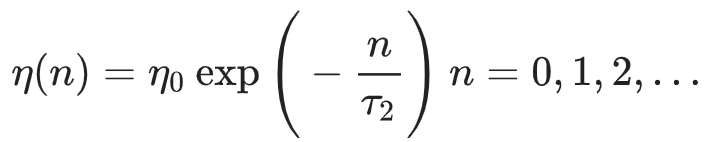
Изображение выглядит как текст

Автоматически созданное описание

где η — параметр скорости обучения.  
 В итоге имеем формулу обновленного вектора весов в момент времени n:



В алгоритме обучения SOM также рекомендуется изменять параметр скорости обучения η в зависимости от шага.



где τ2 — еще одна константа алгоритма SOM.

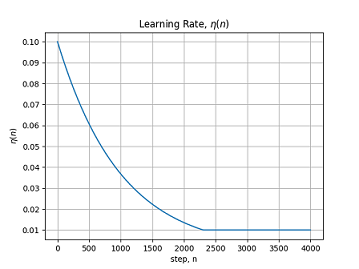


Рисунок 6. График изменения η в процессе обучения.

После обновления весов возвращаемся к шагу 2 и так далее.

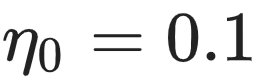
* 1. **Эвристики алгоритма обучения SOM**

Обучения сети состоит из двух этапов:

* **Этап самоорганизации** — может занять до 1000 итераций. В некоторых случаях может занимать больше 1000 итераций;
* **Этап сходимости** — требуется для точной подстройки карты признаков.

Как правило, количество итераций, достаточное для этапа сходимости может превышать количество нейронов в сети в 500 раз.

**Эвристика 1.**  Начальное значение параметра скорости обучения лучше выбрать близким к значению:

****

**Изображение выглядит как текст

Автоматически созданное описание**

При этом оно не должно опускаться ниже значения 0.01.

**Эвристика 2.**  Исходное значение σ0 следует установить примерно равной радиусу решетки, а константу τ1 определить как:

Изображение выглядит как текст

Автоматически созданное описание

На этапе сходимости следует остановить изменение σ.

1. **Реализация модели**
   1. **Используемые программные средства.**

Реализация модели в рамках данной работы производилась в среде разработки VSCode , на языке Python v3.8.

Список используемых библиотек:

* TensorFlow - библиотека для машинного обучения, разработанная для решения задач построения и тренировки нейронной сети с целью автоматического нахождения и классификации образов;
* Matplotlib – библиотека для визуализации данных двумерной графикой;
* NumPy – библиотека, поддерживающая работу с многомерными массивами, высокоуровневыми математическими функциями, предназначенными для работы с многомерными массивами.
  1. **Описание реализации ПО**

Разработанная программа принимает на вход трехмерный вектор, который можно представить как цвет из трех (r,g,b) компонент. После обучения сеть преобразует непрерывное входное пространство цветов в дискретную градиентную карту, при подаче одной гаммы цветов всегда будут активироваться нейроны из одной области карты соответствующей данному цвету.

Перейдем от теории к практической реализации самоорганизующейся карты (SOM) с помощью Python и TensorFlow. Ниже будет описание программного кода с комментариями реализации внутри программы.  
 Для начала создадим класс SOMNetwork и создадим операции TensorFlow для инициализации всех констант:

**Изображение выглядит как текст

Автоматически созданное описание**

Рисунок 7. Реализация класса SOMNetwork

Далее создадим функцию для создания операции процесса конкуренции:

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 8. Функция процесса конкуренции

Нам осталось создать основную операцию обучения для процессов кооперации и адаптации:

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 9. Функция процесса обучения

В итоге была реализована SOM и получена операция training\_op с помощью которой можно обучить нейронную сеть, передавая на каждой итерации входной вектор и номер шага как параметр. Ниже приведен граф операций TensorFlow построенный с помощью Tesnorboard.

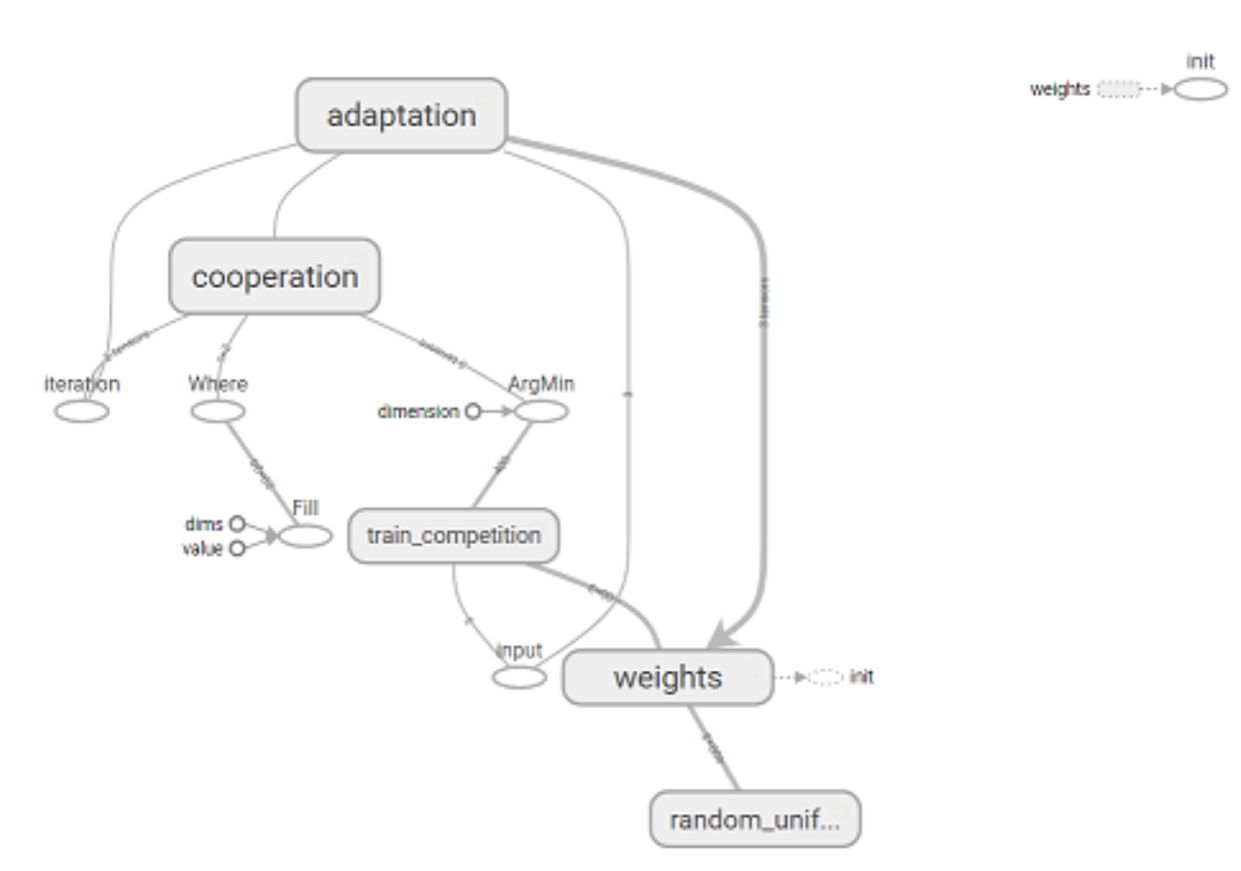


Рисунок 10. Граф операций TensorFlow

* 1. **Тестирование модели**

Для тестирования работы программы, будем подавать на вход трехмерный вектор:



Данный вектор можно представить как цвет из трех (r,g,b) компонент.

Создадим экземпляр нашей SOM сети и массив из случайных векторов (цветов):

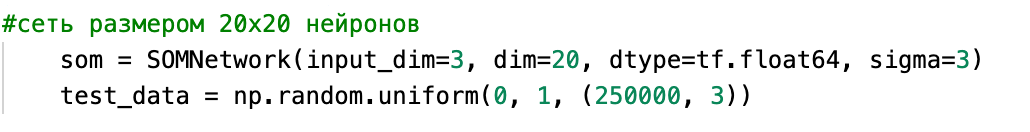


Рисунок 11. Создание экземпляра SOM

После создания экземпляра можно реализовать основной цикл обучения сети:

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 12. Обучение сети

После обучения сеть преобразует непрерывное входное пространство цветов в дискретную градиентную карту, при подаче одной гаммы цветов всегда будут активироваться нейроны из одной области карты соответствующей данному цвету (активируется один нейрон наиболее подходящий подаваемому вектору). Для демонстрации можно представить вектор синаптических весов нейронов как цветовое градиентное изображение.

На рисунке представлена карта весов для сети 20х20 нейронов, после 200тыс. итераций обучения. Результат программы изображен на рисунке ниже.

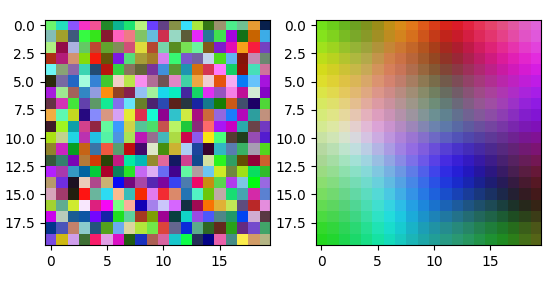


Рисунок 13. Карта весов в начале обучения (слева) и в конце обучения (справа).

После была создана карта весов для сети 100x100 нейронов, после 350тыс. итераций обучения. Результат программы изображен на рисунке ниже.

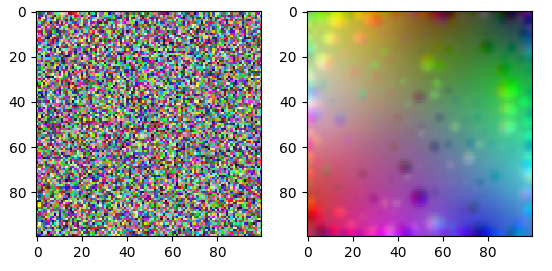


Рисунок 14. Карта весов для сети 100x100 нейронов, после 350тыс. итераций обучения.).

1. **Заключение**

В итоге создана самоорганизующаяся карта и показан пример ее обучения на входном векторе состоящим из трех компонент. Для обучения сети можно использовать вектор любой размерности. Так же можно преобразовать алгоритм для работы в пакетном режиме. При этом порядок представления сети входных данных не влияет на окончательную форму карты признаков и нет необходимости в изменении параметра скорости обучения со временем.

1. **Список использованных источников**

* <https://medium.com/@abhinavr8/self-organizing-maps-ff5853a118d4> - Self Organizing Map, Abhinav Ralhan;
* Introduction to Neural Networks, John A. Bullinaria, 2004;
* <https://en.wikipedia.org/wiki/Self-organizing_map>;
* <https://www.sciencedirect.com/science/article/abs/pii/S0893608012002596> - Essentials of the self-organizing map, Tuevo Kohonen.

**Приложение**

import time

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

class SOMNetwork():

def \_\_init\_\_(self, input\_dim, dim=10, sigma=None, learning\_rate=0.1, tay2=1000, dtype=tf.float32):

if not sigma:

sigma = dim / 2

self.dtype = dtype

#constants

self.dim = tf.constant(dim, dtype=tf.int64)

self.learning\_rate = tf.constant(learning\_rate, dtype=dtype, name='learning\_rate')

self.sigma = tf.constant(sigma, dtype=dtype, name='sigma')

self.tay1 = tf.constant(1000/np.log(sigma), dtype=dtype, name='tay1')

self.minsigma = tf.constant(sigma \* np.exp(-1000/(1000/np.log(sigma))), dtype=dtype, name='min\_sigma')

self.tay2 = tf.constant(tay2, dtype=dtype, name='tay2')

#input vector

self.x = tf.placeholder(shape=[input\_dim], dtype=dtype, name='input')

#iteration number

self.n = tf.placeholder(dtype=dtype, name='iteration')

#variables

self.w = tf.Variable(tf.random\_uniform([dim\*dim, input\_dim], minval=-1, maxval=1, dtype=dtype),

dtype=dtype, name='weights')

#helper

self.positions = tf.where(tf.fill([dim, dim], True))

def feed(self, input):

init = tf.global\_variables\_initializer()

with tf.Session() as sess:

init.run()

win\_index = sess.run(self.\_\_competition(), feed\_dict={self.x: input})

win\_index\_2d = np.array([win\_index//self.dim.eval(), win\_index-win\_index//self.dim.eval()\*self.dim.eval()])

return win\_index\_2d

def training\_op(self):

win\_index = self.\_\_competition('train\_')

with tf.name\_scope('cooperation') as scope:

coop\_dist = tf.sqrt(tf.reduce\_sum(tf.square(tf.cast(self.positions -

[win\_index//self.dim, win\_index-win\_index//self.dim\*self.dim],

dtype=self.dtype)), axis=1))

sigma = tf.cond(self.n > 1000, lambda: self.minsigma, lambda: self.sigma \* tf.exp(-self.n/self.tay1))

sigma\_summary = tf.summary.scalar('Sigma', sigma)

tnh = tf.exp(-tf.square(coop\_dist) / (2 \* tf.square(sigma))) # topological neighbourhood

with tf.name\_scope('adaptation') as scope:

lr = self.learning\_rate \* tf.exp(-self.n/self.tay2)

minlr = tf.constant(0.01, dtype=self.dtype, name='min\_learning\_rate')

lr = tf.cond(lr <= minlr, lambda: minlr, lambda: lr)

lr\_summary = tf.summary.scalar('Learning rate', lr)

delta = tf.transpose(lr \* tnh \* tf.transpose(self.x - self.w))

training\_op = tf.assign(self.w, self.w + delta)

return training\_op, lr\_summary, sigma\_summary

def \_\_competition(self, info=''):

with tf.name\_scope(info+'competition') as scope:

distance = tf.sqrt(tf.reduce\_sum(tf.square(self.x - self.w), axis=1))

return tf.argmin(distance, axis=0)

#== Test SOM Network ==

def test\_som\_with\_color\_data():

som\_dim = 100

som = SOMNetwork(input\_dim=3, dim=som\_dim, dtype=tf.float64, sigma=3)

test\_data = np.random.uniform(0, 1, (250000, 3))

training\_op, lr\_summary, sigma\_summary = som.training\_op()

init = tf.global\_variables\_initializer()

writer = tf.summary.FileWriter('./logs/', tf.get\_default\_graph())

with tf.Session() as sess:

init.run()

img1 = tf.reshape(som.w, [som\_dim,som\_dim,-1]).eval()

plt.figure(1)

plt.subplot(121)

plt.imshow(img1)

start = time.time()

for i, color\_data in enumerate(test\_data):

if i % 1000 == 0:

print('iter:', i)

sess.run(training\_op, feed\_dict={som.x: color\_data, som.n:i})

end = time.time()

print(end - start)

img2 = tf.reshape(som.w, [som\_dim,som\_dim,-1]).eval()

plt.subplot(122)

plt.imshow(img2)

writer.close()

plt.show()

test\_som\_with\_color\_data()