گزارش تمرین دوم هوش محاسباتی

غزاله محمودی

Section one - MLP and RBF network

• MLP

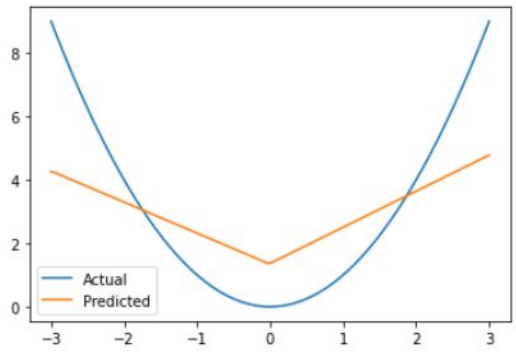
در این سوال قصد داریم به کمک شبکه MLP مدلی طراحی کنیم که تابع y = x^2 را در بازه مشخص یاد بگیرد .

به این منظور ابتدا داده های آزمایشی را نرمال می کنیم .(در رنج صفر و یک می کنیم تا مقادیر عدد داده ها خطاهای غیر منتظره در مدل ما به وجود نیاورد)

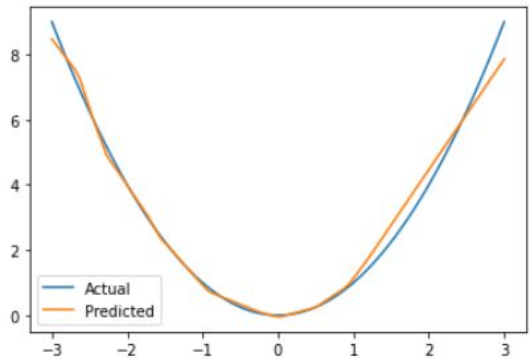
برای نرمال کردن از ماژول Min Max Scaler کتابخانه sklearn.preprocessing استفاده کردم.

شبکه ای که در نظر گرفتم به صورت متوالی (sequential) و dense می باشد به این معنی که همه نورون های هر لایه به لایه بعدی وصل هستند . لازم به ذکر است در این شبکه لایه ورودی و خروجی 1 نورون دارند و با تجربه به دست آمد relu برای این شبکه بهتر از بقیه جواب می دهد.همچنین loss function شبکه می‌تواند mean squared error باشد و optimizer استفاده شده adam است.

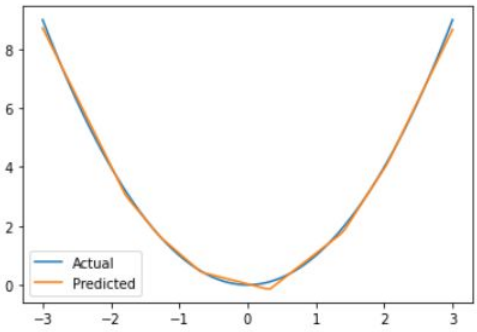
به صورت آزمون و خطا تعدادی مدل با تعداد لایه و نورون های متفاوت را امتحان کردم تا بهترین مدل را پیدا کنم و نتایج به صورت زیر حاصل شد :



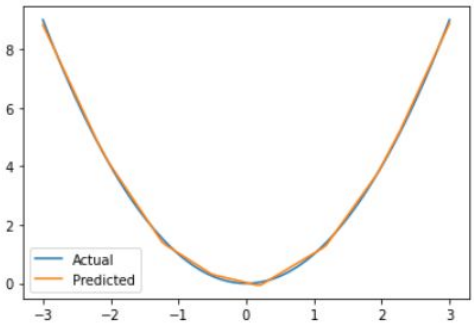
1 لایه هیدن با 20 نورون و 500 اپوک



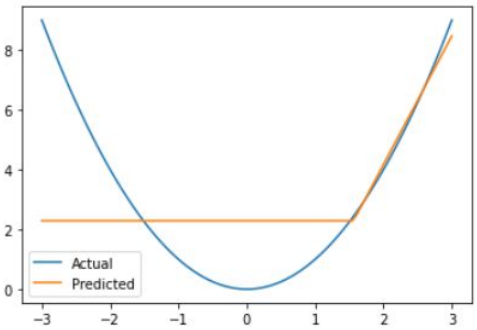
2 لایه هیدن هر کدام 20 نورون 100 اپوک



2 لایه هیدن هر کدام 20 نورون 500 اپوک



2 لایه هیدن هر کدام 16 نورون با 500 اپوک "نهایی"



2 لایه هیدن هر کدام 5 نورون با 500 اپوک

مشاهده شد بهترین حالت 2 لایه با 16 نورون است و تعداد لایه کمتر یا نورون کمتر یا بیشتر دقت مدل را کم می کند .

• RBF

در این قسمت قصد داریم به کمک شبکه RBF مدلی طراحی کنیم که تابع y = x^2 را در بازه مشخص یاد بگیرد .

می دانیم RBF شبکه ای سه لایه است که در واقع تک لایه هیدن تابع شعاعی را بر روی ورودی اعمال می کند .

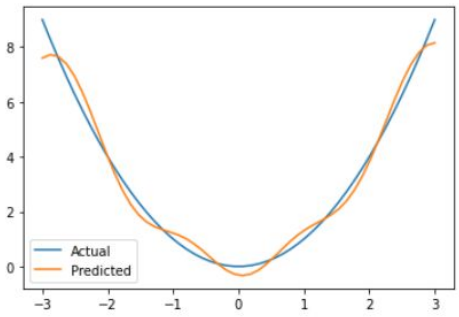
و به این منظور کلاس SOM را به صورت زیر تعریف می کنیم :

```
class RBF :
    def __init__(self, input_dim, output_dim, centers_number , beta = 0.5):
        self.input_dim = input_dim
        self.hidden_dim = output_dim
        self.centers_number = centers_number
        self.center  = [random.uniform(-3, 3) for i in range(self.centers_number)]
        self.weight = []
        self.beta = beta
```

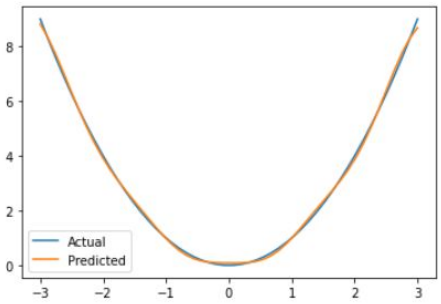
در اینجا قصد داریم مراکزی را انتخاب کنم که می شود از الگوریتم K-mean هم استفاده هر ولی در اینجا به انتخاب رندوم بسنده کردم .

در این مدل وزن یال بین ورودی و میانی 1 است و وزن بین لایه میانی و خروجی هم از ضرب وارون ماتریس activation function در خروجی مطلوب به دست می آید و با توجه به این توضیحات الگوریتم تنها 1 بار اجرا می شود و آنچه تعیین کننده است تعداد مراکز است .

نتیجه RBF :



مدل با 8 مرکز



مدل با 10 مرکز

مقایسه نتایج حاصل از MLP و RBF :

RBF سریع تر از شبکه mlp به نظر می آید دقت مدل هم بهتر است .

و همچنین تعداد epoch هه در RBF به میزان قابل توجهی کمتر است .

در کل استفاده از RBF برای Function approximation بهتر به نظر می رسد گرچه MLP هم با زمان بیشتر نتیجه مطلوب به ما میدهد .

Section Two - self organization map

سوال اول :

در یادگیری بدون ناظر در واقع مدل با استفاده از شباهت بین داده ها و فاصله بین آن ها (مثلا فاصله اقلیدسی) سعی در گروه بندی آن ها دارد به طوری که داده های مشابه در یک گروه قرار گیرند و داده های متفاوت در یک گروه نباشند .

سوال دوم :

S^2 => فاصله اقلیدسی بین هر نود مپ نسبت به نزدیکترین نود مپ به داده ورودی

```
dist = np.sum(np.power(np.array(winner) - np.array(k), 2)) ** 0.5
```

Sigma => فاصله شعاعی مربوط به تابع گوسی

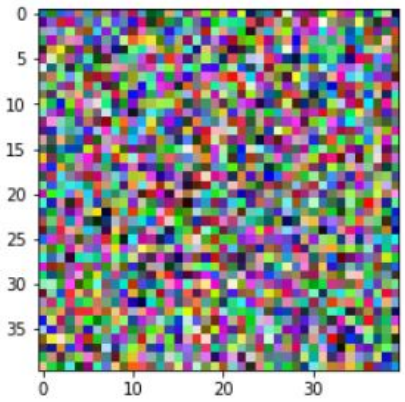
Eta => ضریب یادگیری

T i, j => مقدار تابع گوسی - نشان دهنده مقدار همسایگی

SOM - Kohonen :

در این قسمت ابتدا به کمک قطعه کد زیر 1600 رنگ ساخته و به فضا 40 در 40 می بریم .

```
def generate_random_color(dim = 1600, input_shape=(40, 40, 3)):
    color = np.array([np.random.choice(np.arange(0, 256), 3) for x in range (dim)])
    color = color.reshape(input_shape)
    plt.imshow(color)
    return color
```



نتیجه map ای مانند شکل مقابل می شود .

این رنگ ها به صورت کاملا رندوم generate شده و حال قصد داریم شبکه ای طراحی کنیم که رنگ های مشابه را کنار هم قرار دهد .

از این رو از الگوریتم kohonen استفاده کرده و قدم به قدم آن را پیاده سازی می کنیم .

به این منظور کلاس SOM را به صورت مقابل تعریف می کنیم .

```
def __init__ (self, input_data, input_shape=(40, 40, 3), sigma=4) :

    self.input_data = input_data

    self.weight = self.initial_weight(input_shape)

    self.map_index = [(i, j) for i in range(input_shape[0]) \
                      for j in range(input_shape[1])]

    self.sigma = sigma
```

مراحل الگوریتم

1. مرحله اول در این الگوریتم مقدار دهی اولیه به بردارهای وزن است که به صورت مقابل پیاده سازی می کنیم .

```
#step 1 : initialization
def initial_weight(self, input_shape) :
    return np.array([np.random.choice(np.arange(0, 256), input_shape[2])\
                     for x in range (input_shape[0]*input_shape[1])]).reshape(input_shape).astype(float)
```

متغیر weight در کانستراکتور این کلاس تعریف می شود .

2. مرحله بعدی پیدا کردن نزدیک ترین نود با توجه به نقطه ورودی می باشد .

```
#step 2 : competition
def calculate_nearest_node(self, input_):
    min_value, min_value_index = np.linalg.norm(input_ - self.weight[0][0] ), (0, 0)
    for j in self.map_index :
        _min_value = np.linalg.norm(input_ - self.weight[j])
        if min_value > _min_value :
            min_value = _min_value
            min_value_index = j
    return  min_value_index
```

3. پس از پیدا کردن نزدیک نقطه وقت آن است میزان همسایگی (نزدیک بودن) سایر نقاط نسبت به این نقطه را با کمک تابع گوسی محاسبه کنیم .

```
#step 3 : Cooperation
def neighborhood_function (self, dist) :
    return np.exp(-(dist**2) / (2*self.sigma**2))
```

4. آپدیت وزن ها هم بر اساس مولفه های زیر صورت می پذیرد .

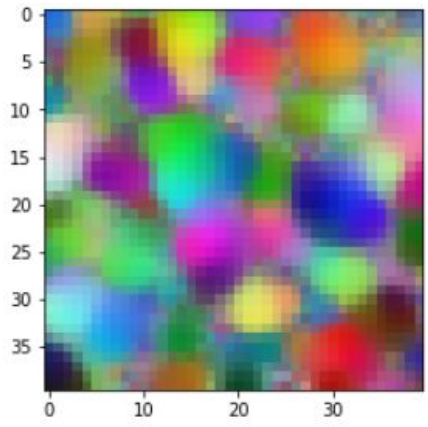
```
# step 4 : Adaptation
def update_weight(self, learning_rate, h, x):
    return learning_rate*h*x
```

و به طور کلی هر epoch از train این مدل شامل 4 مرحله بالاست .

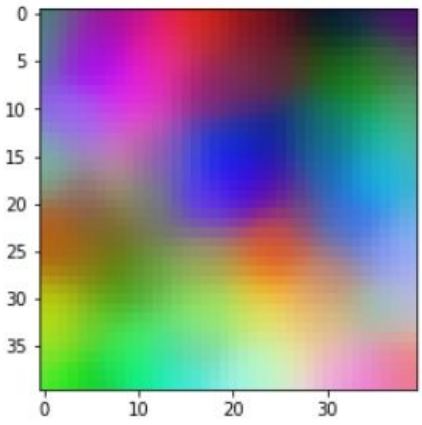
```
def train(self, epoch, learning_rate) :
    for e in range(epoch):
        print(e)
        for i in self.map_index :
            winner = self.calculate_nearest_node(self.input_data[i])
            for k in  self.map_index :
                dist = np.sum(np.power(np.array(winner) - np.array(k), 2)) ** 0.5
                h = self.neighborhood_function(dist)
                self.weight[k] += self.update_weight(learning_rate, h, self.input_data[i] - self.weight[k])
```

برای پیدا کردن بهترین حالت مدل پارامتر های learning rate , epoch , sigma رو با مقادیر مختلف امتحان کردم . طبق مشاهدات پارامتر سیگما خیلی تاثیر گذار تر به نظر می آید . به ازای چند مقدار مختلف sigma شبکه را train کردم و نتایج زیر به دست آمد .

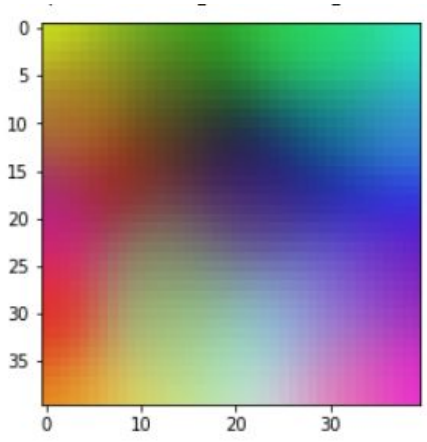
لازم به ذکر است با افزایش سیگما کم کم مرز بین رنگ ها هموار تر می شود و به جایی می رسد که رنگ های تشخیص داده شده کاهش می یابد.



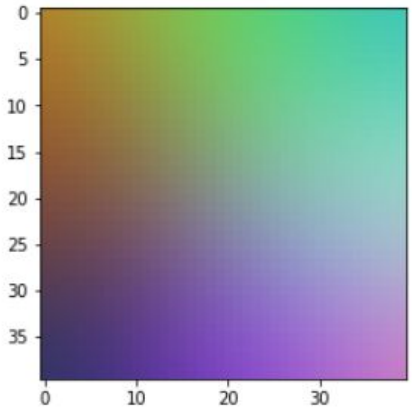
sigma = 1



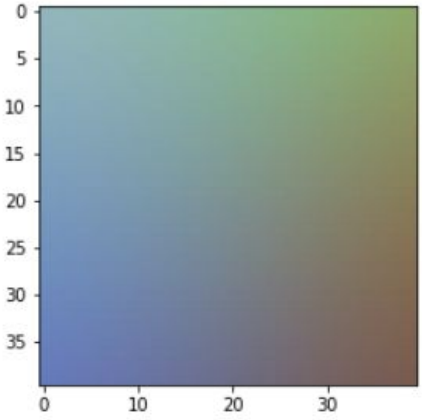
Sigma = 2



sigma = 4



Sigma = 10



sigma = 20