Universidade de São Paulo

Instituto de Física de São Carlos

Análise e reconhecimento de Padrões

Docente: Prof. Luciano Fontoura da Costa

Projeto 4: Mapa Auto-Organizado (SOM)

Beatriz de Camargo Castex Ferreira

10728077

bcastex@usp.br

São Paulo - 10/07/2020

Índice

1. Resumo

Após a geração de padrões e o tratamento de atributos, é importante aprender a reconhecer padrões e categoriza-los corretamente. Seres humanos fazem isso instintivamente, identificando objetos semelhantes e agrupando-os em nossas mentes. Nós somos particularmente eficazes em reconhecimento de imagens, conseguimos, por exemplo, olhar para textos escritos em diversas caligrafias diferentes e ainda identificar o que está escrito. Um computador, por outro lado, tem mais dificuldades, pois a maioria dos programas dependem de busca de valores específicos, ele não consegue ver um dígito 1 escrito de forma ligeiramente diferente e ainda identificá-lo como um 1 normalmente, pois o padrão da imagem (que fala para as luzes em sua tela acenderem ou apagarem) não vai ser igual para os dois. Por isso, quando procura-se criar programas que reconhecem dígitos manuscritos utilizam-se programas de aprendizado de máquina e aprendizado profundo, pois assim pode-se criar uma rede grande de padrões conhecidos e quando é necessário reconhecer um padrão novo ao invés de procurar um padrão igual o programa pode buscar características específicas ou buscar o valor mais parecido dentro de sua rede. Um algoritmo que pode ser utilizado para esse propósito é o mapa auto-organizado, ou SOM (Self-Organizing Map), que tenta mimicar o funcionamento de neurônios, com uma camada de retina, que lê a entrada, e uma camada córtex, que identifica o padrão. Nesse trabalho iremos explorar a arquitetura SOM, tentando entender seu funcionamento e então iremos aplicar o algoritmo em Pyhton

2. Introdução

Como você consegue ler esse texto? Como você sabe que as pequenas linhas desenhadas na tela representam um letras que por sua vez formam palavras? Você ainda conseguiria entender esse texto mesmo se ele fosse escrito em outra fonte?

Essas podem ser perguntas sobre as quais você nunca pensou antes, para nós, seres humanos, o ato de observar e reconhecer padrões é simplesmente a forma com que processamos o mundo. É instintivo, e utilizamos todos os nossos sentidos para tentar diferenciar e compreender o que está a nossa volta, porém, por mais fácil que esse processo seja para nós, reproduzi-lo artificialmente é uma tarefa desafiadora que gerou diversas áreas de pesquisa acadêmica.

Criar uma máquina que seja capaz de observar qualquer objeto que seja apresentado a ela e identificá-lo corretamente do zero levaria décadas, e torná-la eficiente seria ainda mais difícil. Com sorte, temos um modelo já pronto, criado por milhões de anos de evolução e criar tal máquina pode ser bem mais fácil se primeiro aprendermos como nossos próprios cérebros funcionam.

Para isso, precisamos entender o que são neurônios e como eles são estruturados em nosso corpo. Essas simples células são responsáveis por grande parte da nossa habilidade cognitiva, tendo a função de transmitir informações pelo organismo.

Eles tem um formato alongado e são formados por duas partes, o axion e os dendritos.

Os dendritos tem um formato parecido com uma árvore, seus galhos se estendendo a partir do núcleo da célula, e ele é responsável por receber as informações vindas de outros neurônios. Já o axion se assemelha a um lindo tentáculo que se conecta com um dos denários de outro neurônio e é responsável por transmitir a informação, como na imagem abaixo:

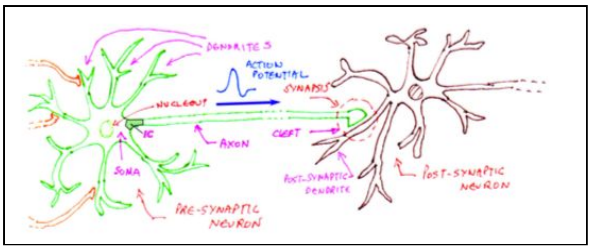

Figura 1. Anatomia de um neurônio[1].

íons são transportados de um neurônio para o outro criando um potêncial de ação, em um processo chamado sinapse, que representa a transmissão de informação de um neurônio para o outro.

[1] Referência [1]. Disponível em: <https://www.researchgate.net/publication/339599069>

Mas como podemos traduzir isso para programação? Um jeito é criar um modelo matemático de um neurônio. O modelo que será utilizado hoje é relativamente simples.

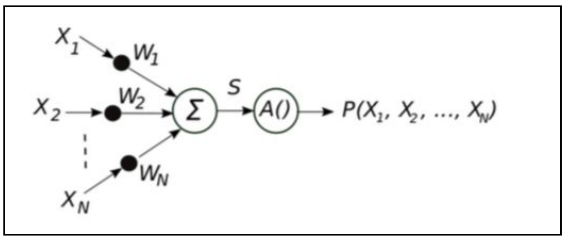

Figura 2. Modelo de um neurônio[1].

Nele, temos uma camada de entrada, por onde ele recebe informações x_i, que representa os dendritos. Cada um destes tem um peso, que mostra o quão efetiva foi a transferência de sinal por esse dendrito. O neurônio então processa toda a informação recebida, multiplicando os dados de input pelos pesos.

$$S = \sum_{i=1}^{N} X_i W_i$$

A função $A()$ então limita o sinal do neurônio, resultando numa função $P(X)$ que seria o sinal transmitido pelo axion para outros neurônios.

Tendo este modelo nos resta entender como os neurônios podem funcionar para reconhecimento de padrões. Uma forma que se pode considerar é como identificador de templates.

Como no modelo existe um peso para cada input que o neurônio recebe, podemos programá-lo para verificar o quão parecidos eles são, assim podemos ter um número que identifique se o peso é uma boa representação dos dados adquiridos.

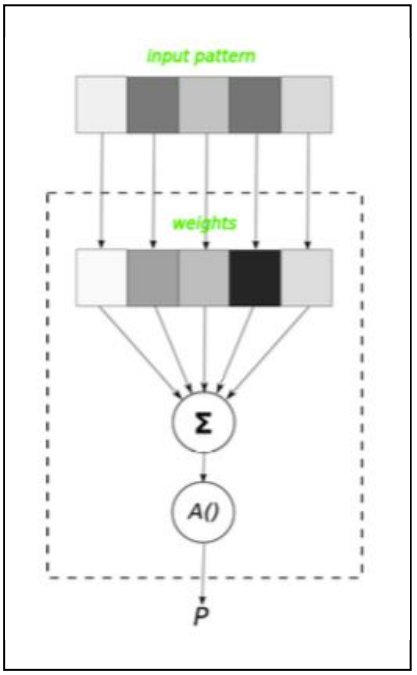

Figura 3. Identificador de template[1].

Na imagem acima por exemplo, podemos ter uma vizualização de como o neurônio compara cada dado do input com seu respectivo peso.

Pode-se também mudar o peso para que esse se torne ligeiramente mais semelhante ao dado de input. Isso pode ser feito aplicando uma função de morfagem, como:

$$\vec{r}(\alpha) = \vec{u} + \alpha(\vec{u} - \vec{v})$$

Com essa informação é possível montar vários modelos de reconhecimento de padrão. Nesse trabalho será utilizado o modelo de mapa auto organizado, ou SOM, que utiliza a possibilidade de verificar a semelhança entre um dado e o peso pré-estabelecido de um neurônio para criar uma rede particionada, em que

cada área identificável se dedica a uma classe de objetos.

3. Mapa Auto Organizado

O SOM é formado de duas camadas, uma camada de retina (que é essencialmente o objeto inserido como input X) e uma camada córtex, que é uma matriz NxN em que cada elemento representa um neurônio com peso W .

Cada peso é um vetor com a mesma dimensão que o input X e que começa tendo valores aleatóriamente distribuidos. Conforme mais inputs passam pela rede esses pesos vão mudando e se tornando mais parecidos com os dados dos objetos até que praticamente cada elemento tenha um dos valores de input como template.

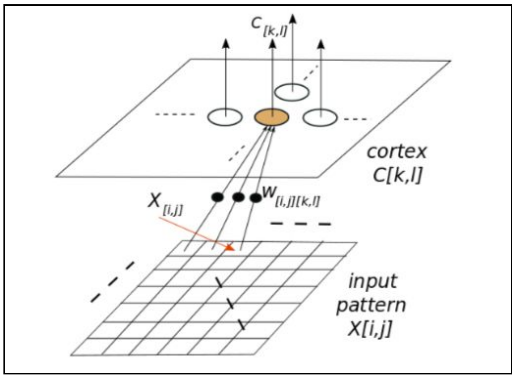

Figura 4. Arquiterura SOM[1].

O algoritmo para a implementação do Som é relativamente simples:

1 - Inicializar uma matriz MxM com pesos aleatórios.

2 - Escolher um dado de treinamento.

3 - Verificar qual elemento da matriz possuí o peso mais parecido com o dado de treinamento.

4 - Morfar o peso mais parecido e seus vizinhos para que fiquem mais similares ao dado de treinamento.

5 - Repetir N vezes.

No passo 3, existem várias formas com que podemos verificar a semelhança entre dois dados, mas para os propósitos desse projeto será utilizada a distância Euclidiana:

$$d_{u,v} = \sqrt{\sum (\vec{u} - \vec{v})^2}$$

Além disso algumas considerações e definições devem ser feitas em relação ao passo 4, particularmente como definir os vizinhos de um ponto na matriz e quanto modificar cada peso.

Podemos considerar vizinhos quaisquer elementos da matriz que estejam dentro de certo raio r de influência do elemento mais parecido, e podemos deixar essa influência mais forte quanto mais perto eles estiverem um do outro. Dessa forma elementos mais próximo do elemento mais parecido (best matching unit ou bmu) serão modificados com maior intensidade, enquanto os mais longe permanecerão iguais. Esse efeito

pode ser obtido utilizando uma exponecial para o cálculo da área de influência I:

$$I = e^{-d/2r^2}$$

Finalmente devemos escolher a constante α para a função de morfagem dos pesos. Essa constante é chamada de constante de aprendizado e indica quão rápido a rede irá se adaptar. É interessante no começo fazer esse valor ser alto, para que aja mais variação nos pesos e a rede se proxime dos dados de treino mais rápido, porém conforme ela se torna mais acurada valores menores de α são mais desejáveis, pois impedem que as morfagens os retirem do estado ótimo ao invés de alcançá-lo.

Por isso, é uma boa ideia fazer com que a constante de aprendizado (e pelos mesmos motivos o raio de influência) diminuam com o tempo. Para isso podemos usar uma função de decaímento como:

$$\alpha_i = \alpha_{inicial} \cdot e^{-i/N},\ i = 1,\ 2,\ 3, \ldots,\ N$$

Assim, temos todos os elementos necessários para implementar uma arquitetura SOM.

4. Implementação em Python

Para esse projeto iremos treinar nossa rede utilizando a base de dígitos manuescritos da Universidade da Califórnia: Irvine[2], que é facilmente inserida no seu programa usando a função *datasets* da biblioteca SciKit Learn[3] :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

''' Importar um dataset
 Você pode trocar essa parte e importar qualque base
 dados da forma que quiser, mas vamos usar a base de dígitos escritos a
mão
 da UC Irvine que está disponível na biblioteca sci-kit learn. Outras
bases de
 dados podem precisar de uma forma diferente de importar. '''

print(
    'How many classes of digits do you want to train the network with?
[1 to 10]')
number_of_classes = int(input())
```

[2] Referência [9].Disponível em: <https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits>
[3] Referência[2].Disponível em: <https://scikit-learn.org/stable/modules/classes.html?highlight=datasets#module-sklearn.datasets>

```
database = 'Digitos Manuscritos com ' + str(number_of_classes) + '
classes'
# Importa base de dados.
digits = datasets.load_digits(n_class=number_of_classes)
data = digits.data  # Coloca os dígitos em uma matriz (sem a label)
label = digits.target  # Coloca os índices em uma matriz (sem os
dígitos)

# Normalizar dados
data = data / data.max()
```

Em seguida inicializamos as variáveis e a rede, aplicando pesos aleatórios para cada um de seus elementos:

```
# Definir variáveis iniciais
N = 5000  # Número de vezes que vai repetir
net_dim = 20  # Tamanho do córtex
initial_learning_rate = 0.2  # rítimo de aprendizado (alpha)
initial_radius = net_dim / 2  # raio de influência dos nódulos
time = N / np.log(initial_radius)  # tempo de deterioração das
constantes
dim = data.shape[1]  # Número de atributos de cada objeto


# 1 - Criar córtex NxN em que cada nódulo tem um vetor de pesos tamanho
dim aleatórios
network = np.random.random((net_dim, net_dim, dim))
start_network = network
```

Então treinamos a rede N vezes, escolhendo um dado aleatório da nossa base de dados a cada vez e decaíndo as constantes conforme o tempo passa:

```
for t in range(N):
    # 2 - Decair o rítimo de aprendizado e o raio com o tempo:
    learning_rate = initial_learning_rate * np.exp(-t / N)
    radius = initial_radius * np.exp(-t / time)

    # 3 - Escolher um ponto aleatório da base de dados para ser o teste
    test = data[np.random.randint(0, data.shape[0]), :]
```

O dado teste então é comparado a todos os pesos da rede e obtem-se o índice do peso mais parecido:

```
# 4 - Achar a unidade de melhor semelhança (bmu)
```

```
    bmu_index = find_bmu(test, network)
```

```
def find_bmu(test_data, net):
    # Encontra o valor de menor distância na rede
    min_distance = 10**20  # um valor absurdo
    i = 0
    for line in net:
        j = 0
        for test_weight in line:
            # Calculamos a distância entre o peso e o test
            distance = np.sum((test_data - test_weight)**2)
            # Encontramos a menor distância:
            if distance < min_distance:
                min_distance = distance
                best_index = [i, j]
            j += 1
        i += 1
    return best_index
```

Então encontramos todos os vizinhos desse nódulo e transformamos tanto ele quanto seus vizinhos de acordo com sua influência:

```
# 5 - Achar os nódulos da rede que estão dentro da área de influência do
bmu:
    # Olhamos todos os nódulos da rede
    for i in range(net_dim):
        for j in range(net_dim):
            # Vemos a distância entre ele e o bmu:
            t_dist = np.sum((np.array([i, j]) - bmu_index)**2)
            # Se estiver dentro do raio:
            if t_dist <= radius**2:
                # Calcular influência:
                influence = np.exp(-(t_dist) / (2 * (radius**2)))
                # Morfar vetor para se parecer mais com o dado teste
                to_morph = network[i, j, :]
                weight_morph = to_morph + \
                    (learning_rate * influence * (test - to_morph))
                network[i, j, :] = weight_morph
```

Após rodar todas as N vezes teremos uma rede de pesos particionada de acordo com o dígito que cada peso representa, como veremos nos resultados abaixo.

O código fonte está disponível por completo, justamente com todos os arquivos utilizados para gerar os arquivos abaixo, caso haja interesse[4].

[4] Código fonte. Disponível em: <https://drive.google.com/drive/folders/1Oou6OwhfBmK8s-g1o4DYCS7c-7oDNzEL?usp=sharing >

4. Resultados

Ao rodar o programa para a base de dados com apenas quatro dígitos obteve-se o seguinte resultado:

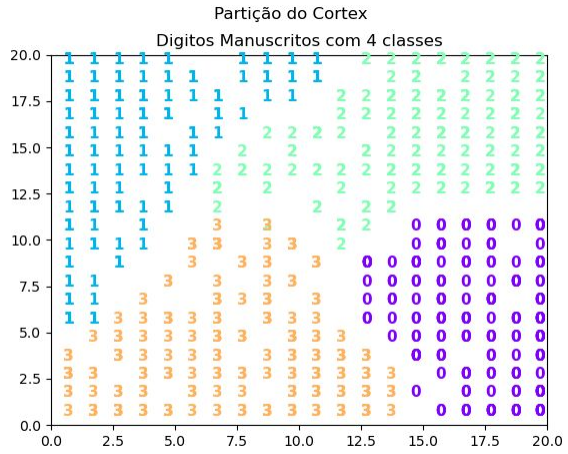

Figura 5. Resultado com 4 dígitos.

Onde se pode ver claramente a partição da rede, de forma que cada área está dedicada à apenas um dígito. Deste modo, é possível inserir um dado teste na rede e ver o índice do bmu dele. Dependendo da repartição em que o bmu se encontra, podemos definir que dígito era o dado teste.

Imprimindo a imagem em greyscale dos pesos da rede podemos ver claramente que os pesos agora se aproximam com os dígitos manuescritos com os quais a rede foi treinada:

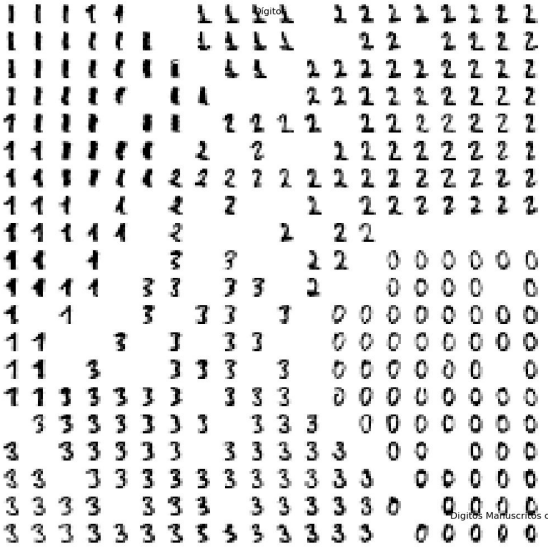

Figura 6. Greyscale dos pesos.

Porém esse método não é perfeito. Nem sempre iremos obter uma partição limpa, como pode-se perceber ao ver o gráfico da partição formada ao utilizarmos 5 dígitos:

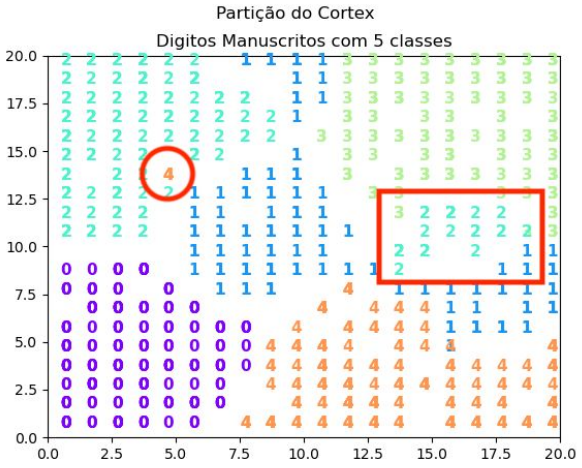

Figura 7. Resultado com 5 dígitos.

Como podemos ver pela área circulada, um único quatro foi salvo na partição que seriam os dígitos 2, e na área demarcada por um retângulo temos um pequeno conjunto de dois separado dos demais. Isso pode levar a classificação incorreta de um certo dado,

pois seu bmu está em uma determinada partição mas indica outro dígito.

Olhando o mapa em greyscale fica claro o porque isso ocorre:

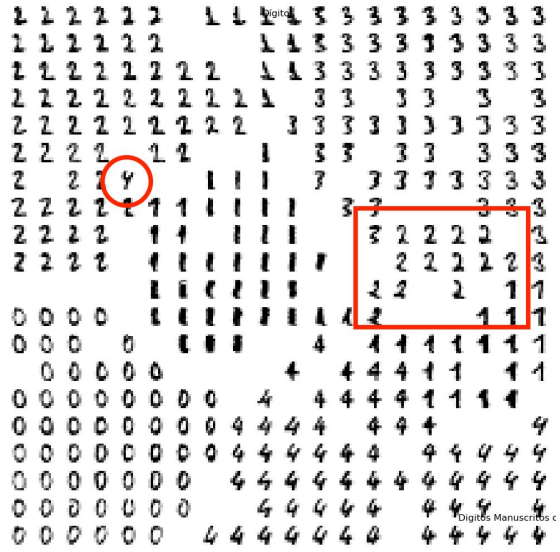

Figura 8. Greyscale dos pesos com 5 dígitos.

O quatro que ficou na partição dos 2, é um dígito particularmente estilizado, e os 2 que se separaram de sua partição são vizualmente similares aos 3, dos quais eles estão próximos. Também é possível notar agora que houve ainda outra separação, entre números 1 que são escritos de forma reta e números 1 que tem um "chapéu", mais parecidos com 4.

5. Conclusão

A arquitetura SOM é extremamente eficiênte para reconhecimento de imagens, criando partições bem demarcadas para cada classe de objetos apresentados à ela. Porém esta não é infalível, podendo ter erros em casos de imagens similares, mas de classes diferentes.

Referências:

[1] COSTA. L. da F. **Neurons as Pattern Recognizers.**Researchgate.Março,2020Disponível em: <https://www.researchgate.net/publication/339599069>

[2] API Reference¶. Disponível em: <https://scikit-learn.org/stable/modules/classes.html?highlight=datasets#module-sklearn.datasets>

[3] ABHINAVRALHAN. **abhinavralhan/kohonen-maps**. Disponível em: <https://github.com/abhinavralhan/kohonen-maps/blob/master/gsom-iris-python.ipynb>

[4] JUSTGLOWING. **JustGlowing/minisom**. Disponível em: <https://github.com/JustGlowing/minisom/blob/master/examples/HandwrittenDigits.ipynb>

[5] RALHAN, A. **Self Organizing Maps**. Disponível em:

<https://towardsdatascience.com/self-organizing-maps-ff5853a118d4>

[6] Self Organizing Maps. Disponível em: <https://glowingpython.blogspot.com/2013/09/self-organizing-maps.html>

[7] numpy.ndarray.shape¶. Disponível em: <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.shape.html>

[8] ŷhat: Self-Organising Maps: In Depth. Disponível em: <http://blog.yhat.com/posts/self-organizing-maps-2.html>

[9] ALPAYDIN ,E.; Kaynak, C..

Optical Recognition of Handwritten Digits Data Set. UCI. Disponível em: <https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits>

[10] SuperDataScience. **Self Organizing Maps: Introduction.** Disponível em: <https://www.youtube.com/watch?v=0qtvb_Nx2tA>

[11] 3Blue1Brown. **But what is a Neural Network? | Deep learning, chapter 1.** Disponível em: <https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2&t=0s>