Aula3

September 8, 2023

1 Operações de Álgebra Linear e Cálculo

1.1 1. Operações de Álgebra Linear

1.1.1 1.1. Somas e Reduções de Dimensionalidade ; Transmissão (“BroadCasting”)

Vamos importar o tensorflow e criar uma matriz

[1]:
```
import tensorflow as tf
A=tf.reshape(tf.range(6),(2,3))
A
```

[1]:
```
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[0, 1, 2],
       [3, 4, 5]])>
```

Observe que é um array bidimensional 2x3. Frequentemente precisamos somar as linhas ou colunas de matrizes (para fazer normalizações, por exemplo).

[2]:
```
tf.reduce_sum(A)
```

[2]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=15>
```

O `reduce_sum`, sem parâmetros, soma todos os elementos da matriz A, e devolve um escalar (dimensão 0).

[3]:
```
tf.reduce_sum(A,axis=0)
```

[3]:
```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([3, 5, 7])>
```

Indicando-se a dimensão 0 (axis), apenas as linhas são colapsadas, isto é, temos um vetor de tamanho 3 (array monodimensional) constituído pela soma de cada coluna. Observe como a função `reduce_suma` princípio diminui a dimensionalidade do objeto. A função `reduce_mean` tem comportamento semelhante, mas com a média (e não a soma) de elementos.

Esta diminuição de dimensionalidade nem sempre é desejável. Digamos que calculemos a soma das colunas de A, para normalizá-las (fazer toda coluna ter soma 1). Não podemos fazer simplesmente `A/reduce_sum(A,axis=0)` porque estaríamos dividindo uma matriz (dimensão 2) por um vetor (dimensão 1). A solução é usar a opção `keepdims=True` para que as dimensões de tamanho 1 não sejam colapsadas. Em outras palavras, para que esta soma de colunas seja representada em uma matriz 3x1 e não um vetor 3.

[4]:
```
SomaLinhas=tf.reduce_sum(A,axis=1)
SomaLinhas2=tf.reduce_sum(A,axis=1,keepdims=True)
SomaLinhas,SomaLinhas2
```

[4]:
```
(<tf.Tensor: shape=(2,), dtype=int32, numpy=array([ 3, 12])>,
 <tf.Tensor: shape=(2, 1), dtype=int32, numpy=
 array([[ 3],
        [12]])>)
```

[5]:
```
A/SomaLinhas2
```

[5]:
```
<tf.Tensor: shape=(2, 3), dtype=float64, numpy=
array([[0.        , 0.33333333, 0.66666667],
       [0.25      , 0.33333333, 0.41666667]])>
```

Mas dissemos anteriormente que a divisão / é realizada elemento a elemento. A e SomaLinhas 2 têm dimensão 2, mas os tamanhos não são os mesmos (2x3 contra 2x1). Como a operação pôde ser realizada corretamente. A resposta está na importante característica de **Broadcasting** (transmissão) pressuposta nas operações matriciais no TensorFlow. Antes de realizar a operação, as dimensões de tamanho 1 são replicadas para que tenhamos tamanhos compatíveis. Em outras palavras, interpreta-se que o que se deseja é fazer 3 cópias (em colunas) da matriz 2x1 para que se tenha uma matriz 2x3. Observe que a operação é exatamente o que queríamos: a soma das linhas é 1.

"Broadcasting" será muito útil na manipulação de grandes tabelas de dados em Aprendizado de Máquina.

1.1.2 1.2. Produto interno, produto matriz-vetor e matriz-matriz

A função `tf.tensordot` realiza o produto interno de vetores.

[6]:
```
x,y=tf.range(3),tf.range(3)
x,y,tf.tensordot(x,y,axes=1)
```

[6]:
```
(<tf.Tensor: shape=(3,), dtype=int32, numpy=array([0, 1, 2])>,
 <tf.Tensor: shape=(3,), dtype=int32, numpy=array([0, 1, 2])>,
 <tf.Tensor: shape=(), dtype=int32, numpy=5>)
```

Já a função `tf.matmul`realiza o produto matriz x vetor.

[7]:
```
W=tf.random.normal(shape=(3,3))
W
```

[7]:
```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[ 0.7075696 , -0.83752555,  0.27156076],
       [-0.5742506 , -0.605356  ,  1.0385914 ],
       [ 0.53732204, -0.40703335, -0.7355571 ]], dtype=float32)>
```

[8]:
```
Y=tf.random.normal(shape=(3,3))
tf.matmul(W,Y)
```

[8]:
```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[ 0.8055265 ,  2.448222  , -0.09804476],
       [ 0.51324964,  1.318014  ,  1.2131957 ],
       [ 0.71605265,  0.44860893, -0.17497365]], dtype=float32)>
```

1.1.3 1.3. Normas

A função `tf.norm` calcula a norma L2 (ou euclidiana) de um vetor.

[9]:
```
A=tf.constant([3.0,4.0])
tf.norm(A)
```

[9]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

O ".0" ao final faz com que A seja um vetor de `float32`. A função `tf.norm`não aceita inteiros como argumentos. É possível contornar isso com a função `tf.cast`

[10]:
```
A = tf.constant([3, 4])
tf.norm(tf.cast(A, dtype=tf.float32))
```

[10]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=5.0>
```

Se aplicada a uma matriz, `tf.norm`apresenta a norma de Frobenius da matriz (raiz quadrada da soma dos quadrados de todos os elementos).

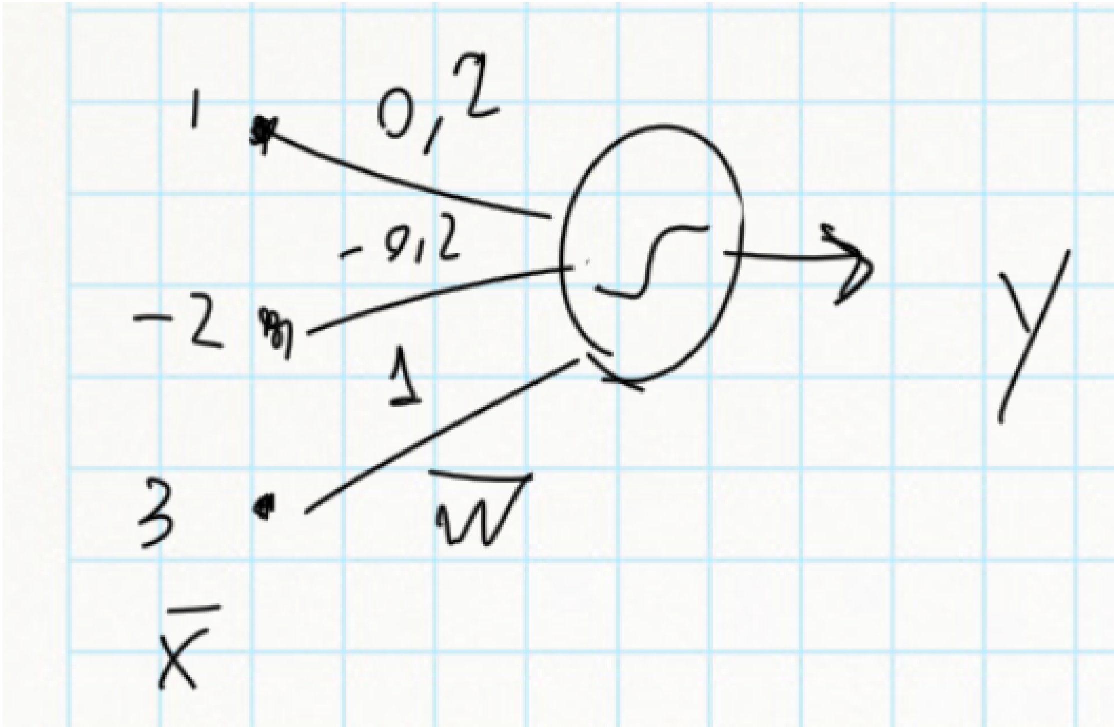
[11]:
```
A=tf.ones((3,3))
tf.norm(A)
```

[11]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=3.0>
```

Observe que a função `tf.ones` devolve um vetor de `float32`, não sendo necessário o "casting".

1.2 2. Cálculo: Diferenciação automática

Na Figura abaixo, temos um neurônio artificial com entradas $\vec{x}$, pesos sinápticos $\vec{w}$ e saída y.



Diferente do modelo que vimos aula passada, este neurônio não computa apenas o produto interno entre $\vec{w}$ e $\vec{x}$ (ou dito de outra forma, faz a soma das entradas $\vec{x}$ ponderada pelos pesos $\vec{w}$. Ele passa a soma resultante por uma função não-linear, conhecida como logística ou sigmoide:

$$y = \frac{1}{1 + \exp(-s)} \tag{1}$$

, onde s é a soma.

Frequentemente queremos saber qual a derivada de y com relação a cada peso w_i. Isto será usado para modificar w de acordo com os erros que a rede neural comete.

`tensorflow` permite o cálculo automático das derivadas de variáveis computadas.

Primeiro, vamos simular um passo do cálculo da saída do neurônio.

[12]:
```
x=tf.Variable(tf.constant([1.0,-2.0,3.0]))
w=tf.Variable(tf.constant([0.2,-0.2,1]))
```

Encontramos pela primeira vez a função `tf.Variable`. Esta é uma ferramenta de economia de memória. Python usa gerenciamento dinâmico de memória, o que significa que ele vai alocando memória ao longo da execução. Isto pode ser útil no gerenciamento de memória (podemos ir descartando a memória que não será mais usada), mas se usado sem cuidado pode ter o efeito contrário. Quando fazemos `Y=X+Y` alocamos memória para `X+Y` mesmo sendo o objetivo apenas atualizar Y, por exemplo.

Para garantir que algo que será calculado várias vezes (como os parâmetros de uma rede neural) sejam alocados em uma mesma porção de memória, existe a função `tf.Variable`. Ela recebe o alor inicial da variável e, futuramente, o valor pode ser mudado sem alocar nova memória.

Resolvida a questão da alocação, podemos calcular o valor da saída. Mas queremos que este cálculo seja usado para que o gradiente da saída em relação aos pesos $\vec{W}$ seja calculado automaticamente. Para isto, colocamos o código onde a conta é feito sob a declaração `with tf.GradientTape() as t:`. A declaração `with` em Python define um encapsulamento e um contexto para o código. É comum para arquivos (veja a aula anterior), que podem ser fechados após a leitura, ou justamente para declarar que as operações devem ser memorizadas para que se compute o gradiente, como aqui.

[13]:
```
with tf.GradientTape() as t:
    y=tf.sigmoid(tf.tensordot(x,w,axes=1))
y
```

[13]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.973403>
```

Observe que a computação está correta. O produto interno nos fornece $1 \times 0.2 + (-2) \times (-0.2) + 3 \times 1 = 3,6$. Verifique que a função logística definida acima, vale 0.97 para $s = 3,6$.

Agora vamos ao gradiente.

[14]:
```
dydw = t.gradient(y, w)
dydw
```

[14]:
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([ 0.02588962, -0.05177924,
0.07766887], dtype=float32)>
```

De novo, verifique que a computação está correta. Para calcular a derivada parcial de y com relação a cada w_i aplique a regra da cadeia.

$$\frac{\partial y}{\partial w_i} = \frac{dy}{ds} \cdot \frac{\partial s}{w_i} \tag{2}$$

Compute a derivada de y com relação a s, verificando que vale $s(1-s)$. O outro fator, já que s é simplesmente a soma ponderada, vale x_i. Multiplique os fatores e verifique o resultado.

[ ]:
```
```