Aula2

September 3, 2023

1 Vetores, matrizes e Tensores com TensorFlow

1.1 1. Introdução

Veremos que as operações das redes neurais artificiais podem ser convenientemente descrita na forma de vetores e matrizes (arrays) de várias dimensões . Na área de IA, especialmente na comunidade de computação, as matrizes de dimensão maior que 2 são geralmente chamadas tensores. Existe alguma polêmica nisso, já que tensores são objetos de física-matemática que generalizam a ideia de vetores e matrizes, que só fazem sentido a partir de um sistema de coordenadas de referência, e que devem satisfazer certas propriedades de transformação. Eu particularmente acho que deveríamos continuar a chamá-lo arrays multidimensionais.

Este caderno é uma adaptação de cadernos do ambiente d2l de aprendizado em "deep learning". Eu modifiquei ligeiramente alguns exemplos e incluí alguns comentários para quem não é tão familiarizado com python e tensorflow.

1.2 2. Arrays de várias dimensões.

1.2.1 2.1. Criação e Manipulação

[1]:
```
import tensorflow as tf
```

Primeiro, vejamos a criação com tf.range(N). Com parâmetros default, a função cria uma sequência de valores de 0 a N-1, em passos de 1.

[2]:
```
x = tf.range(12, dtype=tf.float32)
x
```

[2]:
```
<tf.Tensor: shape=(12,), dtype=float32, numpy=
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.],
      dtype=float32)>
```

Observe que, internamente, Python trata o vetor como uma lista de números. Python é todo orientado para tratar com listas, e elas são definidas entre colchetes.

Agora vejamos a criação de um array com a função tf.zeros . Ela cria um array de zeros com o tamanho especificado.

[3]:
```
tf.zeros((2,3))
```

[3]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)>
```

Observe os parênteses duplos. A função aceita o tamanho do array a ser criado como uma "tupla". Tupla é uma lista imutável, não pode ser manipulada depois de criada. Observe também que o atributo "shape" do objeto agora indica duas dimensões. Trata-se de uma matriz 2x3 (ou, se preferir uma lista de 2 vetores de 3 elementos). Veja também que a matriz (de dimensão 2) é descrita como uma lista de listas.

Como você deve ter imaginado, tf.ones cria arrays preenchidos com 1s.

[4]:
```
tf.ones((2,3,2))
```

[4]:
```
<tf.Tensor: shape=(2, 3, 2), dtype=float32, numpy=
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]]], dtype=float32)>
```

Arrays de dimensão 3 (ou tensores, lembre-se da polêmica citada acima....) são obviamente listas de listas de ... listas. Finalmente, podemos (e frequentemente queremos) gerar arrays com valores aleatórios. Isso pode ser feito, por exemplo, com a função tf.random.normal (neste caso, a distribuição é normal com média 0 e desvio-padrão 1)

[5]:
```
tf.random.normal((2,2))
```

[5]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[-0.01023923,  0.12669297],
       [-2.3172934 , -1.1132221 ]], dtype=float32)>
```

A função tf.constant pode ser usado para especificar os elementos do array um a um.

[6]:
```
tf.constant([[1,2],[3,4],[5,6]])
```

[6]:
```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[1, 2],
       [3, 4],
       [5, 6]])>
```

1.2.2 2.2. Indexação, fatiamento e reforma

O sistema de indexação para listas em Python (e consequentemente para arrays) causa alguma estranheza a princípio. O primeiro elemento é o 0. Índices negativos podem ser usados para acessar valores a partir do fim da lista.

[7]:
```
x=tf.constant([1,2,3,4,5])
x[0],x[4],x[-1],x[-2]
```

[7]:
```
(<tf.Tensor: shape=(), dtype=int32, numpy=1>,
 <tf.Tensor: shape=(), dtype=int32, numpy=5>,
 <tf.Tensor: shape=(), dtype=int32, numpy=5>,
 <tf.Tensor: shape=(), dtype=int32, numpy=4>)
```

As faixas a:b como índice selecionam os elementos de a até o **antecessor de b**. Se a for omitido, entende-se "do começo", se b for omitido, "até o final".

[8]:
```
x[0:3], x[-3:-1],x[0:-1],x[:2],x[2:]
```

[8]:
```
(<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3])>,
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 4])>,
 <tf.Tensor: shape=(4,), dtype=int32, numpy=array([1, 2, 3, 4])>,
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([1, 2])>,
 <tf.Tensor: shape=(3,), dtype=int32, numpy=array([3, 4, 5])>)
```

[9]:
```
x=tf.reshape(tf.range(9),(3,3))
x
```

[9]:
```
<tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])>
```

Acima usamos tf.reshape para transformar o vetor de 9 elementos em uma matriz 3x3. Se omitirmos uma das dimensões, selecionamos todo o elemento. Por exemplo, se o índice tem uma dimensão apenas, entende-se toda a linha:

[10]:
```
x[-1],x[:2]
```

[10]:
```
(<tf.Tensor: shape=(3,), dtype=int32, numpy=array([6, 7, 8])>,
 <tf.Tensor: shape=(2, 3), dtype=int32, numpy=
 array([[0, 1, 2],
        [3, 4, 5]])>)
```

1.2.3 2.3. Operações elemento a elemento

As operações como exp(.) , +, *, / e ** são feitas elemento a elemento.

[11]:
```
x=tf.constant([[1,2],[3,4]])
y=tf.constant([[0,1],[2,3]])
x+y, x*y, y/x, x**y
```

[11]:
```
(<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
 array([[1, 3],
        [5, 7]])>,
```

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 0,  2],
       [ 6, 12]])>,
<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
array([[0.        , 0.5       ],
       [0.66666667, 0.75      ]])>,
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 1,  2],
       [ 9, 64]])>)
```

Notar que os tipos são implícitos. Veja como o resultado da divisão é um float64 e os demais inteiros.

Outra operação útil é a de concatenação. O atributo axis indica qual dimensão será concatenada `axis=0` significa concatene as linhas, `axis=1` concatene as colunas, etc.

[12]:
```
X = tf.reshape(tf.range(12, dtype=tf.float32), (3, 4))
Y = tf.constant([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
X,Y,tf.concat([X, Y], axis=0), tf.concat([X, Y], axis=1)
```

[12]:
```
(<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
 array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]], dtype=float32)>,
 <tf.Tensor: shape=(3, 4), dtype=float32, numpy=
 array([[2., 1., 4., 3.],
        [1., 2., 3., 4.],
        [4., 3., 2., 1.]], dtype=float32)>,
 <tf.Tensor: shape=(6, 4), dtype=float32, numpy=
 array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]], dtype=float32)>,
 <tf.Tensor: shape=(3, 8), dtype=float32, numpy=
 array([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]], dtype=float32)>)
```

Uma expressão lógica envolvendo arrays é aplicada elemento a elemento. O resultado é um novo array onde um elemento vale “true” quando a condição é satisfeita e “false” quando não é.

[13]:
```
X==Y
```

[13]:
```
<tf.Tensor: shape=(3, 4), dtype=bool, numpy=
array([[False,  True, False,  True],
       [False, False, False, False],
       [False, False, False, False]])>
```

2 3. Arquivos

Frequentemente, os dados para treinar um modelo estão localizados em longos arquivos CSV (valores separados por vírgula). Nesta seção vamos seguir um exemplo do d2l: criar um pequeno arquivo de exemplo, lê-lo e manipulá-lo.

[14]:
```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('''NumRooms,RoofType,Price
NA,NA,127500
2,NA,106000
4,Slate,178100
NA,NA,140000''')
```

"os" é o módulo que contém as funções de interface com o sistema operacional. A função "makedirs" com os parâmetros acima cria um diretório chamado "data" acima do diretório atual, a menos que já exista. data_file é o ponteiro para o novo arquivo House_tiny.csv a ser criado. O write escreve o string determinado neste arquivo. As 3 aspas simples indicam que o string se estende por mais de uma linha.

Agora, importamos o módulo pandas, que têm funções de manipulação de dados, e lemos o arquivo.

[15]:
```
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms RoofType   Price
0       NaN      NaN  127500
1       2.0      NaN  106000
2       4.0    Slate  178100
3       NaN      NaN  140000
```

A função pd.read_csv devolve um "Dataframe", uma estrutura bidimensional com diferentes tipos de dados (note como a tabela tem textos e números). O "NA" que incluímos na Tabela é uma indicação de um dado faltante, algo comum em grandes bases de dados do mundo real. Esta Tabela (se tivesse milhares de registros e não 4...) poderia ser uma base de dados de preços de casas a partir do número de quartos e tipo de telhado, para um problema de regressão (estimar o preço de uma nova casa que não está na base). Neste caso, as entradas seriam os valores das duas primeiras colunas, e o alvo (saída) a última. Vamos separar os dados em entradas e saídas.

[16]:
```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
print(inputs)
print(targets)
```

```
   NumRooms RoofType
0       NaN      NaN
```

```
1       2.0      NaN
2       4.0    Slate
3       NaN      NaN
0    127500
1    106000
2    178100
3    140000
Name: Price, dtype: int64
```

A propriedade iloc é usada para acessar pontos específicos do dataframe, com as mesmas regras de indexação que vimos anteriormente.

Agora vamos tratar da questão do NA. Se o dado faltante é uma categoria, geralmente a solução é criar uma nova categoria "NA". Como todos os tipos de telhados que existem são "Slate" (Ardósia), vão passar a existir duas categorias de telhado: Slate ou NA.

[17]:
```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  RoofType_Slate  RoofType_nan
0       NaN           False          True
1       2.0           False          True
2       4.0            True         False
3       NaN           False          True
```

A função `pd.get_dummies` transforma as colunas de categoria em expressões lógicas ("True" se a categoria do nome da coluna está presente, "False" se não está). Com a opção `dummy_na=True`, é criada uma coluna para os casos em que a informação de categoria é inexistente.

Para o caso de valores numéricos faltantes, uma solução comum é atribuir a média dos valores existentes.

[23]:
```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms  RoofType_Slate  RoofType_nan
0       3.0           False          True
1       2.0           False          True
2       4.0            True         False
3       3.0           False          True
```

O método `fillna`aplicado a qualquer Dataframe substitui os valores NA com os valores dados. No caso, a média da coluna, acessada pelo método `mean`

Os dados estão separados em uma matriz de entrada e um vetor correspondende de saída, as categorias estão expressas como variáveis lógicas, e não há valores faltantes. Mas inputs e targets ainda são "dataframes". Para transformá-las em objeto tensor, usamos a função `tf.constant` e o método `to_numpy` aplicado ao dataframe.

[24]:
```
X = tf.constant(inputs.to_numpy(dtype=float))
y = tf.constant(targets.to_numpy(dtype=float))
```

```
X, y
```

[24]:
```
(<tf.Tensor: shape=(4, 3), dtype=float64, numpy=
 array([[3., 0., 1.],
        [2., 0., 1.],
        [4., 1., 0.],
        [3., 0., 1.]])>,
 <tf.Tensor: shape=(4,), dtype=float64, numpy=array([127500., 106000., 178100.,
140000.])>)
```

[ ]:
```

```