Aula 11

TensorFlow - Introdução e cálculo de gradiente

Eduardo Lobo Lustosa Cabral

1. Objetivos

Apresentar uma introdução do funcionamento do TensorFlow.

Apresentar as estruturas de dados usadas pelo TensorFlow.

Apresentar as principais funções do TensorFlow.

Apresentar como calcular derivadas de funções e gradientes de funções de múltiplas variáveis.

Apresentar a solução de um problema de ajuste de função usando o TensorFlow para calcular o gradiente da função de custo em relação aos parâmetros do modelo

2. Introdução

A utilização do TensorFlow sem o Keras não é muito simples e exige conhecimento mais profundo de programação em Python.

Existem diversas formas de utilizar o TensorFlow:

- No caso de redes neurais, a forma mais fácil é usar o Keras do TensorFlow, como visto nas aulas anteriores;
- Porém, muitos programas são feitos com o TensorFlow sem usar o Keras, assim, é importante conhecer as outras formas de usar o TensorFlow.

O TensorFlow realiza operações com tensores e é semelhante à biblioteca Numpy.

O TensorFlow possui inúmeras funções para implementar cálculos com tensores, redes neurais e muitos outros tipos de cálculos → é muito difícil conhecer tudo e para fazer cálculos mais complexos deve-se procurar ajuda na internet.

O TensorFlow realiza cálculo diferencial, sendo capaz de calcular derivadas e gradientes de funções de mútiplas variáveis. Ressalta-se que o treinamento de uma rede neural é baseado no método do gradiente descendente, que utiliza o gradiente da função de custo em relação a todos os parâmetros da rede. Dessa forma, essa capacidade do TensorFlow é essencial para o treinamento de redes neurais e o cálculo do gradiente da função de custo deve ser realizado de forma extremamente eficiente.

Importação das bilbiotecas necessárias

```
import numpy as np
import tensorflow as tf
print(tf.__version__)
```

```
2.17.0
```

3. Constantes e variáveis

No TensorFlow constantes e variáveis são usados para representar os parâmetros de uma RNA.

Os principais atributos das constantes e variáveis do TensorFlow são os seguintes:

- `value`: um valor (ou lista);
- `dtype`: tipo de elemento (int32, float32 etc);
- `shape`: dimensão do tensor;
- `name`: nome dado (opcional).

3.1 Constantes:

Constantes no TensorFlow servem para armazenar valores que não variam durante a execução do programa.

Constantes são inicializadas quando são criadas com a função `tf.constant` e seus valores nunca mudam.

Na célula a seguir é apresentado um exemplo de definir constantes e usá-las para fazer uma operação simples.

```
# Construção do gráfico
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b

# Imprime variável c
print(c)
print(format(c))
print(c.numpy())
```

```
tf.Tensor(30.0, shape=(), dtype=float32)
30.0
30.0
```

- Observe que o comando `print(c)` apresenta além do valor da variável, o seu tipo e sua dimensão.
- Se for desejado mostrar somente o valor da variável `c` deve-se usar a função `format(c)` ou `c.numpy()`.

- O método `numpy()` faz com que a constante (variável) funcione como se fosse um tensor Numpy.
- O método `format()` transforma o valor em uma string.

Existem outras formas de definir uma `tf.constant`, que são exemplificadas na célula abaixo.

```
# Cria um a constante que consiste em um vetor linha
v = tf.constant([1, 2, 3])
print('v =', v)

# Cria um tensor de constantes 2D a partir de uma lista de valores
A = tf.constant([1, 2, 3, 4, 5, 6], shape=(2,3))
print('\nA =', A)

# Cria tensor com todos elementos iguais a uma constante, por exemplo
-1
B = tf.constant(-1.0, shape=(2,3))
print('\nB =', B)
```

```
v = tf.Tensor([1 2 3], shape=(3,), dtype=int32)

A = tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)

B = tf.Tensor(
[[-1. -1. -1.]
 [-1. -1. -1.]], shape=(2, 3), dtype=float32)
```

- Observa-se que a lista com os valores não precisa ter o mesmo "shape" da constante criada com a função `tf.constant()`.

3.2 Variáveis

As variáveis do TensorFlow são estruturas de dados cujos valores podem ser modificados durante a execução do programa.

Variáveis são usadas, por exemplo, para manter e atualizar parâmetros de um modelo.

Variáveis são definidas fornecendo seus valores iniciais e tipos:

- No caso de não se definir explicitamente o tipo da variável, o TensorFlow infere a partir do formato dos números usados na definição da variável;
- Valores iniciais para as variáveis são definidos quando elas são criadas;
- Variáveis podem ser inicializadas com valores constantes ou com números aleatórios;
- Pode-se dar um nome para a variável.

Na célula a seguir é mostrado como definir e inicializar variáveis.

```
# Cria tensor W1 de uns com dimensão 2x2
W1 = tf.ones((2,2))

# Define variável W2 de dimensão 2x2 com zeros
W2 = tf.Variable(tf.zeros((2,2)), name="weights")

# Define variável R de dimensão 2x2 com números aleatórios
R = tf.Variable(tf.random.normal((2,2)), name="random_weights")

# Imprime resultados
print('\nW1=', W1)
print('\nW2=', W2)
print('\nR=', R)
```

```
W1= tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)

W2= <tf.Variable 'weights:0' shape=(2, 2) dtype=float32, numpy=
array([[0., 0.],
       [0., 0.]], dtype=float32)>

R= <tf.Variable 'random_weights:0' shape=(2, 2) dtype=float32, numpy=
array([[ 0.48741174, -0.35732424],
       [-1.3654437 , -1.4866557 ]], dtype=float32)>
```

- Observe que `W1` não é uma "variável" do TensorFlow.
- Função `tf.ones()` cria um tensor com todos elementos iguais a 1.
- Função `tf.zeros()` cria um tensor com todos elementos iguais a 0.
- Função `tf.random.normal()` gera um tensor de números aleatórios com distribuição normal com média 0 e desvio padrão igual a 1; se form desejado pode-se definir tanto a média como o desvio padrão. Os argumentos padrão dessa função são: `tf.random.normal(shape, mean=0.0, stddev=1.0, dtype=tf.dtypes.float32, seed=None, name=None).

Diferença entre variáveis e outros tipos de estruturas:

- Variáveis são mantidas após a execução de um gráfico computacional;
- Outros tipos de variáveis são automaticamente descartados após a execução do gráfico computacional.

Existem outras formas de definir uma `tf.Variable`, exemplificadas a seguir.

```
# Define e inicializa vetor de variáveis reais
v = tf.Variable([1, 2, 3], dtype=tf.float32)
```

```
# Define e inicializa matriz
C = tf.Variable([[1, 2], [3, 4], [5, 6]])

# Define matriz a partir de uma lista de valores
D = tf.Variable([1, 2, 3, 4, 5, 6], shape=tf.TensorShape(None))

# Imprime resultados
print('v =', v)
print('\nC =', C)
print('\nD =', D)
```

```
v = <tf.Variable 'Variable:0' shape=(3,) dtype=float32,
numpy=array([1., 2., 3.], dtype=float32)>

C = <tf.Variable 'Variable:0' shape=(3, 2) dtype=int32, numpy=
array([[1, 2],
       [3, 4],
       [5, 6]], dtype=int32)>

D = <tf.Variable 'Variable:0' shape=<unknown> dtype=int32,
numpy=array([1, 2, 3, 4, 5, 6], dtype=int32)>
```

- No caso de `tf.Varaible` a dimensão é definida pelo valor inicial.
- Pode especificar a dimensão (shape) de uma variável como sendo `unknown`; isso permite inicializar um tensor com determinados e valores e dimensão e depois alterar a sua dimensão do forma que for necessário.

Tanto nas constantes como nas variáveis pode-se armazenar strings e números complexos.

```
# Define tf.constante de strings
insetos = tf.constant(['formiga', 'mosca', 'aranha'])

# Define tf.Variable de strings
mamiferos = tf.Variable(['gato', 'cavalo'])

# Define tf.Variable de n[umeros complexos
z = tf.Variable([[1 + 2j], [-2 +3j]])

# Imprime resultados
print('insetos =', insetos)
print('\nmamiferos=', mamiferos)
print('\nz =', z)
```

```
insetos = tf.Tensor([b'formiga' b'mosca' b'aranha'], shape=(3,),
dtype=string)

mamiferos= <tf.Variable 'Variable:0' shape=(2,) dtype=string,
```

```
numpy=array([b'gato', b'cavalo'], dtype=object)>

z = <tf.Variable 'Variable:0' shape=(2, 1) dtype=complex128, numpy=
array([[ 1.+2.j],
       [-2.+3.j]])>
```

3.3 Operações matemáticas com constantes e variáveis

Existem comandos específicos para realizar operações matemáticas com `tf.constant` e `tf.Variable`.

- Soma: `tf.add(x, y)`
- Subtração: `tf.subtract(x, y)`
- Multiplicação: `tf.multiply(x, y)`
- Divisão: `tf.divide(x, y)`

Observa-se que essas funções realizam operações elemento por elemento.

```
# Define dois tensores
x = tf.Variable([[2., 3.], [4., 9.]])
y = tf.constant([2., 3])
print('x =', x.numpy())
print('y =', y.numpy())

# Soma
print('x + y =', tf.add(x, y))

# Subtração
print('x - y =', tf.subtract(x, y))

# Mutiplicação
print('x*y =', tf.multiply(x, y))

# Divisão
print('x/y =', tf.divide(x, y))
```

```
x = [[2. 3.]
 [4. 9.]]
y = [2. 3.]
x + y = tf.Tensor(
[[ 4.  6.]
 [ 6. 12.]], shape=(2, 2), dtype=float32)
x - y = tf.Tensor(
[[0. 0.]
 [2. 6.]], shape=(2, 2), dtype=float32)
x*y = tf.Tensor(
[[ 4.  9.]
 [ 8. 27.]], shape=(2, 2), dtype=float32)
x/y = tf.Tensor(
```

```
[[1. 1.]
 [2. 3.]], shape=(2, 2), dtype=float32)
```

- Para realizar operações elemento por elemento de dois tensores as suas dimensões devem ser compatíveis.
- Operações elemento por elemento com tensores TensorFlow seguem as mesmas regras de "broadcast" dos tensores Numpy.
- Observe que realizar cálculos com o TensorFlow para, por exemplo, criar e depois treinar uma RNA somente é possível se forem usados as suas funções.

3.4 Funções gerais

Existem inúmeras funções no TensorFlow para realizar cálculos. Alguns exemplos são mostrados nas células a seguir.

```
# Define tensor de variáveis
z = tf.Variable([1., 2., 3.])
print('z =', z)

# Calculo do quadrado elemento por elemento de um tensor
z2 = tf.square(z)
print('\nz^2 =', z2.numpy())

# Soma dos elementos de um vetor
sum_z = tf.reduce_sum(z)
print('sum_z =', sum_z.numpy())

# Soma dos elementos de uma matriz
A = tf.Variable([[1, 2], [3, 4], [5, 6]])
sum_col = tf.reduce_sum(A, axis=0)
sum_lin = tf.reduce_sum(A, axis=1)
sum_tot = tf.reduce_sum(A)
print('\nA =', A)
print('sum_col =', sum_col.numpy())
print('sum_lin =', sum_lin.numpy())
print('sum_tot =', sum_tot.numpy())

# Redimensiona tensor
a = tf.reshape(A, (1,6))
print('a =', a.numpy())

# Altera tipo da variável z de real para inteiro
z_int = tf.cast(z, tf.int32)
print('\nz_int =', z_int)
```

```
z = <tf.Variable 'Variable:0' shape=(3,) dtype=float32,
numpy=array([1., 2., 3.], dtype=float32)>
```

```
z^2 = [1. 4. 9.]
sum_z = 6.0

A = <tf.Variable 'Variable:0' shape=(3, 2) dtype=int32, numpy=
array([[1, 2],
       [3, 4],
       [5, 6]], dtype=int32)>
sum_col = [ 9 12]
sum_lin = [ 3  7 11]
sum_tot = 21
a = [[1 2 3 4 5 6]]

z_int = tf.Tensor([1 2 3], shape=(3,), dtype=int32)
```

Importante:

1. Se quisermos fazer operações com tensores do TensorFlow como se fossem tensores Numpy, basta utilizar `tensor.numpy()` nas operações.

2. As operações de adição, subtração, multiplicação e quadrado elemento por elemento de tensores do TensorFlow podem ser realizadas simplesmente por:

   `x + y`, `x - y`, `x*y` e `x**2`

3.5 Funções `assign()`, `assign_add()` e `assign_sub()`

Três funções muito úteis do TensorFlow são as seguintes:

- `assign()`: atribui um valor para uma variável
- `assign_add()`: adiciona algum valor em uma varável
- `assign_sub()`: subtrai algum valor de uma variável

No código abaixo é apresentado como se usam essas funções.

```python
# Define variável inicializad com zero
x = tf.Variable(0.0)
print('x =', x)

# Atribui valor 10 à variável x
x.assign(10.0)
print('x =', x.numpy())

# Soma 1 no novo x
x.assign_add(1.0)
print('x =', x.numpy())

# Subtrai 5 do novo x
x.assign_sub(5.0)
print('x =', x.numpy())
```

```
x = <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.0>
x = 10.0
x = 11.0
x = 6.0
```

- Essas funções são úteis para incrementar contadores e para atualizar parâmetros de uma RNA durante o processo de treinamento.

Observa-se que não é possível acessar o resultado numérico dessas funções pelo método `x.numpy()`. Porém, para isso pode-se usar o método `read_value().numpy()`, com descrito abaixo.

```
print(x.read_value().numpy())
```

```
6.0
```

Na célula a seguir é mostrado um exemplo de como realizarum cálculo simples usando o TensorFlow. Nesse programa a variável `state` é um contador que é incrementado de 1, por 3 iterações.

```
# Cria variável state
state = tf.Variable(0, name="counter")
print('state=', state.numpy())

for _ in range(3):
    # Incrementa 1 na variável state (new_value = state + 1)
    new_value = tf.add(state, tf.constant(1))

    # Atribui novo valor na variável state (state = new_value)
    state.assign(new_value)

    # Imprime resultado
    print('state=', state.numpy())
```

```
state= 0
state= 1
state= 2
state= 3
```

- Observe que o comando `sate.assign(new_value)` atribui o valor de `new_value` à variável `state`.
- Observe que poderia simplesmente utilizar `state.assign_add(1)` no lugar de definir a variável `new_value` e usar `state.assign(new_value)`.

Para que usar essa forma complicada de fazer cálculos?

- Esse tipo de programação é necessário para o TensorFlow poder criar o que se chama de "gráfico computacional".

- O gráfico computacional permite criar um modelo simbólico dos cálculos, para posteriormente, se for necessário, calcular por exemplo os gradientes das funções definidas no gráfico.
- As variáveis e constantes são os nós de um gráfico computacional, ou seja, sem elas o TensorFlow não consegue criar o gráfico.
- Para poder realizar os cálculos simbólicos para obter as equações do gradiente descendente o TensorFlow precisa que todas as operações sejam definidas utilizando as suas funções matemáticas do tipo `add()`, `multiply()` etc.

3.6 Visualização dos parâmetros de uma RNA

Um operação interessante do TensorFlow é visualizar todas os parâmetros de uma RNA.

Na célula abaixo é criada uma RNA simples de duas camadas e após isso são visualizados os seus parâmetros.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Cria RNA
rna = Sequential()
rna.add(Dense(3, activation='sigmoid', input_shape=(2,)))
rna.add(Dense(1, activation='linear'))

# Apresenta resumo da rede
rna.summary()

# Apreenta parâmetros
rna.variables
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/
dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━
┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━
┩
│ dense (Dense)                        │ (None, 3)                   │
9 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────
┤
```

```
──────────────────┤
│ dense_1 (Dense)                      │ (None, 1)                   │
4 │
└──────────────────────────────────────┴─────────────────────────────┴
──────────────────┘

 Total params: 13 (52.00 B)

 Trainable params: 13 (52.00 B)

 Non-trainable params: 0 (0.00 B)

[<KerasVariable shape=(2, 3), dtype=float32,
path=sequential/dense/kernel>,
 <KerasVariable shape=(3,), dtype=float32,
path=sequential/dense/bias>,
 <KerasVariable shape=(3, 1), dtype=float32,
path=sequential/dense_1/kernel>,
 <KerasVariable shape=(1,), dtype=float32,
path=sequential/dense_1/bias>]

rna.variables[3]

<KerasVariable shape=(1,), dtype=float32,
path=sequential/dense_1/bias>
```

4. TensorFlow versus Numpy versus Keras

TensorFlow, Keras e Numpy são muito similares → todos operam com tensores de múltiplas dimensões.

Vantagens e desvantagens:

- Numpy tem a desvantagem de não oferecer métodos para criar funções de tensores e calcular suas derivadas automaticamente;
- Numpy não tem suporte para cálculo em GPU;
- TensorFlow e Keras tem a desvantagem do código ser mais complicado.

Nas duas células a seguir são apresentados, como exemplo, alguns cálculos realizados com Numpy e com o TensorFlow. Esses cálculos mostram algumas características do TensorFlow e suas diferenças em relação ao Numpy.

```
# Cálculos com Numpy
import numpy as np

a = np.zeros((2,2))
b = np.ones((2,2))
c = np.sum(b, axis=0)
d = np.reshape(a, (1,4))
```

```
print('b=', b)
print('c=', c)
print('Dimensão de a=', a.shape)
print('d=', d)
```

```
b= [[1. 1.]
 [1. 1.]]
c= [2. 2.]
Dimensão de a= (2, 2)
d= [[0. 0. 0. 0.]]
```

```
# Cálculos com TensorFlow
a = tf.zeros((2,2))
b = tf.ones((2,2))
c = tf.reduce_sum(b, axis=0)
d = tf.reshape(a, (1, 4))
```

```
print('b=', b)
print('c=', c.numpy())
print('Dimensão de a=', a.get_shape())
print('d=', d.numpy())
```

```
b= tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)
c= [2. 2.]
Dimensão de a= (2, 2)
d= [[0. 0. 0. 0.]]
```

- No tensorFlow para para realizar a soma dos elementos de um tensor tem-se o método `tf.reduce_sum()`.
- No TensorFlow para obter as dimensões de um tensor tem-se o método `get_shape()`.

Alguns comandos do Numpy e seus equivalentes no TensorFlow;

```
Numpy                TensorFlow

a = np.zeros((2,2))    a = tf.zeros((2,2))
b = np.ones((2,2))      b = tf.ones((2,2))
np.sum(b, axis=1)        tf.reduce_sum(a, axis=1)
a.shape                 a.get_shape()
np.reshape(a, (1,4))    tf.reshape(a, (1,4))
c = b * 5 + 1          c = b * 5 + 1
np.dot(a,b)              tf.matmul(a,b)
```

4.1 Função `tf.matmul()`

A função `tf.matmul()` realiza operações entre dois tensores segundo as regras da álgebra linear. As dimensões dos eixos dos tensores tem que ser compatíveis para poder realizar essa operação. A célula seguinteapresenta alguns exemplos.

```
# Define vetores x e y, e matriz A
x = tf.Variable([[1., 2., 3.]])
y = tf.Variable([[-1.], [-2.], [-3.]])
A = tf.Variable(tf.ones((3,3)), dtype=tf.float32)
B = tf.Variable(3*tf.ones((3,2)), dtype=tf.float32)
print('x =', x.numpy())
print('y =', y.numpy())
print('A =', A.numpy())
print('B =', B.numpy())

# Produto escalar entre dois vetores
z = tf.matmul(x, y)
print('\nProduto escalar entre x e y =', z.numpy())

# Produto externo entre dois vetores
w = tf.matmul(y, x)
print('\nProduto externo entre x e y:\n', w.numpy())

# Multiplicação entre matriz  e vetor
C = tf.matmul(A, y)
print('\nProduto da matriz A pelo vetor y:\n', C.numpy())

# Multiplicação entre matrizes
D = tf.matmul(A, B)
print('\nProduto das matrizes A e B:\n', D.numpy())
```

```
x = [[1. 2. 3.]]
y = [[-1.]
 [-2.]
 [-3.]]
A = [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
B = [[3. 3.]
 [3. 3.]
 [3. 3.]]

Produto escalar entre x e y = [[-14.]]

Produto externo entre x e y:
 [[-1. -2. -3.]
 [-2. -4. -6.]
 [-3. -6. -9.]]
```

```
Produto da matriz A pelo vetor y:
 [[-6.]
 [-6.]
 [-6.]]

Produto das matrizes A e B:
 [[9. 9.]
 [9. 9.]
 [9. 9.]]
```

5. Cálculo de derivada de função

O TensorFlow possui a função `GradientTape()` (https://www.tensorflow.org/api_docs/python/tf/GradientTape), que realiza diferenciação automática de funções.

Essa função é capaz de calcular derivada de qualquer ordem de qualquer tipo de função.

Lembre que o treinamento das redes neurais é baseado no método do Gradiente Descendente, que consite no cálculo das derivadas parciais da função de custo em relação aos parâmetros da rede.

5.1 Derivada de função de uma variável

A utilização dessa função é muito simples. Seja a seguinte função:

$$y = 3x^2 - 1$$

Deseja-se obter a derivada de y em relação a x, ou seja:

$$\frac{dy}{dx} = 6x$$

Na célula abaixo é apresentado como calcular essa derivada.

```
# Define alguns valores para variável x
x = tf.constant([-3., -2., -1., 0., 1., 2., 3.])

# Define função que se deseja calcular a derivada
with tf.GradientTape() as grad:
    grad.watch(x)
    y = 3.0*x**2 - 1.0

# Calcula a derivada de y em relação a x
dydx = grad.gradient(y, x)

print('Derivada de y em relação a x =', dydx.numpy())
```

```
Derivada de y em relação a x = [-18. -12.  -6.   0.   6.  12.  18.]
```

- Observe que são retornados valores numéricos para a derivada.

As operações cujas derivadas deseja-se calcular, devem ser executadas dentro do gerenciador de contexto (comando `with`) e as variáveis devem ser "observadas" pelo método `watch()`.

No caso de uma RNA, as suas variáveis treináveis, marcadas por `trainable=True`, são observadas automaticamente e não precisam ser "observadas" com o método `watch()`.

Outras variáveis, como no exemplo acima, são observadas invocando o método `watch()` dentro do gerenciador de contexto.

5.2 Derivada de ordem superior

Vários contextos de `GradientTapes()` podem ser inseridos um dentro do outro para calcular derivadas de ordem superior.

Seja a seguinte função:

$$y = x^3 + 2x$$

Primeira derivada de y em relação a x é dada por:

$$\frac{dy}{dx} = 3x^2 + 2$$

A segunda derivada de y em relação a x é dada por:

$$\frac{d^2y}{dx^2} = 6x$$

Na célula abaixo é apresentado como calcular essa segunda derivada.

```
# Define alguns valores para variável x
x = tf.constant([-3., -2., -1., 0., 1., 2., 3.])

# Contexto (bloco) da 2a derivada
with tf.GradientTape() as grad2:
    grad2.watch(x)

    # Bloco da 1a derivada
    with tf.GradientTape() as grad1:
        grad1.watch(x)
        y = x**3 + 2*x
    # Calcula primeira derivada
    dydx = grad1.gradient(y, x)

# Calcula segunda
d2ydx2 = grad2.gradient(dydx, x)

# Mostra resultados
print('Primeira derivada =', dydx.numpy())
print('Segunda derivada =', d2ydx2.numpy())
```

```
Primeira derivada = [29. 14.  5.  2.  5. 14. 29.]
Segunda derivada = [-18. -12.  -6.   0.   6.  12.  18.]
```

- Observe que a primeira derivada fica dentro do bloco da segunda derivada.

5.3 Gradiente de função de múltiplas variáveis

A função `GradientTape()` também pode calcular o gradiente de uma função de várias variáveis. Esse tipo de problema é encontrado no treinamento de uma rede neural.

As RNAS possuem inúmeros parâmetros e no seu treinamento é calculado o gradiente da função de custo em relação a todos os parâmetros treináveis da RNA.

Seja a seguinte função de 4 variáveis:

$$y = x_1 + 2x_2 + 3x_3 + 4x_4$$

$$z = y^2$$

ou seja,

$$z(X) = (x_1 + 2x_2 + 3x_3 + 4x_4)^2$$

onde,

$$X = \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix}$$

Queremos calcular o gradiente da função $z(X)$, ou seja, as derivadas parciais de z em relação a x_1, x_2, x_3 e x_4.

Usando a regra da cadeia da derivada, tem-se para a derivada parcial de z em relação a x_1:

$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x_1}$$

onde $\frac{\partial z}{\partial y} = 2y$ e $\frac{\partial y}{\partial x_1} = 1$.

Portanto,

$$\frac{\partial z}{\partial x_1} = 2y$$

Analogamente para x_2, x_3 e x_4, tem-se:

$$\frac{\partial z}{\partial x_2} = 4y$$

$$\frac{\partial z}{\partial x_3} = 6y$$

$$\frac{\partial z}{\partial x_4} = 8y$$

O gradiente de z em relação à matriz X é também uma matriz, dada por:

$$\frac{\partial z}{\partial X} = \begin{pmatrix} \frac{\partial z}{\partial x_1} & \frac{\partial z}{\partial x_2} \\ \frac{\partial z}{\partial x_3} & \frac{\partial z}{\partial x_4} \end{pmatrix} = \begin{pmatrix} 2y & 4y \\ 6y & 8y \end{pmatrix}$$

O código da célula abaixo calcula o gradiente dessa função em relação à X.

```
# Define matriz X
X = tf.ones((2,2))

# DEfine contexto para o cálculo do gradiente
with tf.GradientTape() as grad:
    # Marca variável que se deseja calcular o gradiente
    grad.watch(X)

    # Define função z
    y = X[0,0] + 2*X[0,1] + 3*X[1,0] + 4*X[1,1]
    z = tf.square(y)

# Calcula o gradiente
dzdX = grad.gradient(z, X)

print('Gradiente de z em relação a X:\n', dzdX.numpy())
```

```
Gradiente de z em relação a X:
 [[20. 40.]
 [60. 80.]]
```

6. Regressão linear usando TensorFlow

Como exemplo de uso do TensorFlow para desenvolver um modelo de aprendizado de máquina, é mostrado como resolver um problema de ajuste de função usando regressão linear.

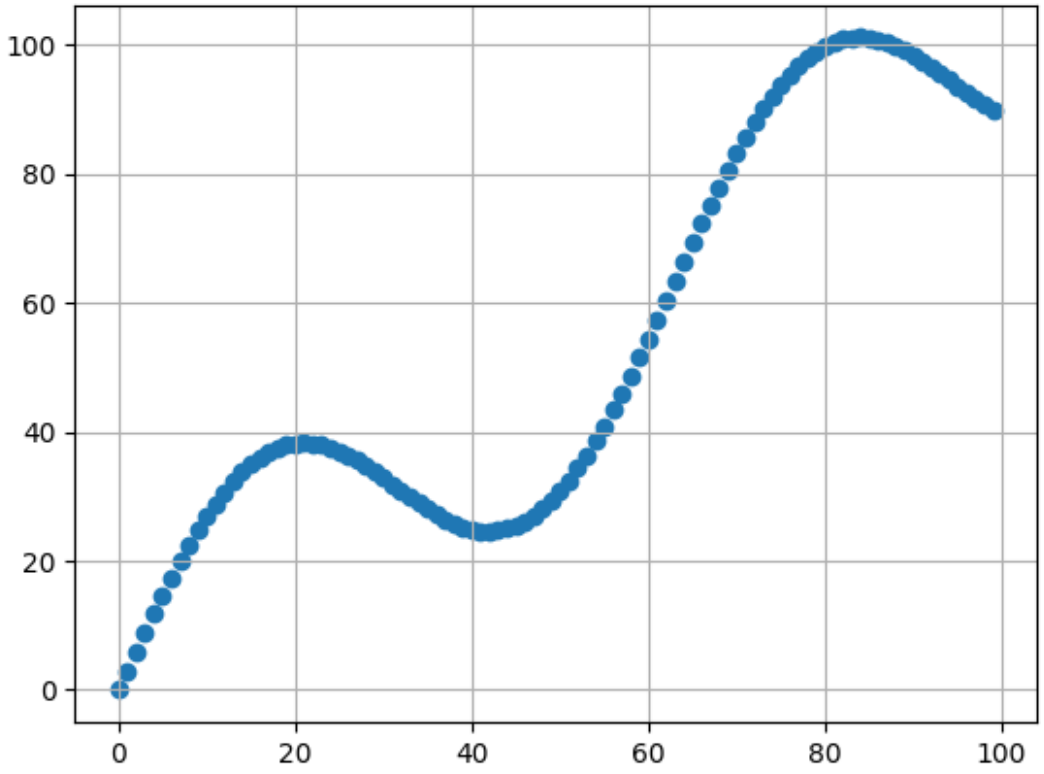
6.1 Dados

Na célula a seguir são gerados os dados do problema.

```
import numpy as np
import matplotlib.pyplot as plt

# Define dados do problema
```

```
X_data = np.arange(100, step=1)
X_data = np.float32(X_data)
y_data = X_data + 20*np.sin(X_data/10)

# Dimensão dos dados de entrada e de saída
print(X_data.shape, y_data.shape)

# Gráfico dos dados
plt.scatter(X_data, y_data)
plt.grid()
plt.show()
```

```
(100,) (100,)
```



- A função `arange(start, stop, step)` do Numpy retorna um vetor com valor inicial igual a `start` e valor final igual a `stop` menos `step`, com valores espaçados de `step`.
- Observe que os dados de entrada e de saída são vetores, mas não é definido se são vetores linha ou coluna.

O TensorFlow é muito sensível à dimensão dos dados, assim, é sempre bom definir essas dimensões sem incerteza. Na célula a seguir são ajustadas as dimensões dos dados de entrada e de saída para não ter erro de dimensão nos cálculos.

```
# Define dimensão dos dados
n_samples = X_data.shape[0]

# Ajuste da dimensão dos dados
X_data = np.reshape(X_data, (n_samples,1))
y_data = np.reshape(y_data, (n_samples,1))

print(X_data.shape, y_data.shape)
```

```
(100, 1) (100, 1)
```

- Observe que a dimensão original dos dados de entrada e de saída era (100,) e após o seu "acerto" fica sendo (100, 1).

6.2 Incialização de parâmetros do modelo

O modelo que será usado para ajustar os dados é uma reta cuja equação é a seguinte:

$$y = W\,x + b$$

onde W é a inclinação da reta e b é o ponto de cruzamento da reta no eixo das ordendas. Dessa forma W e b são os parâmetros do modelo que serão calculados no treinamemto.

Na célula a seguir são definidos e inicializados os parâmetros W e b.

```
# Define e inicializa parâmetros da regressão a serem calculados
b = tf.Variable(tf.zeros((1)), name="bias", trainable=True)
W = tf.Variable(tf.random.uniform((1,1), seed=1), name="weight",
trainable=True)

# Imprime valores iniciais de W e b
print('W inicial=', W)
print('b inicial=', b)
```

```
W inicial= <tf.Variable 'weight:0' shape=(1, 1) dtype=float32,
numpy=array([[0.8514762]], dtype=float32)>
b inicial= <tf.Variable 'bias:0' shape=(1,) dtype=float32,
numpy=array([0.], dtype=float32)>
```

- Observe que os parâmetros de ajuste da reta (`W` e `b`) são do tipo `tf.Variable`.
- O viés `b` é inicializado com zero e o peso `W` com um número aleatório com distribuição uniforme.

6.3 Ajuste do modelo

Para ajustar o modelo aos dados vamos criar um loop de treinamento usando a função `tf.GradientTape()` para calcular o gradiente da função de custo em relação aos parâmetros do modelo e usar o método do Gradiente Descendente para atualizar os parâmetros em cada iteração (época).

O código da célula abaixo implementa o loop de treinamento.

```
# Taxa de aprendizado
lr = 0.001

# Número de épocas
num_epocas = 1000

# Define otimizador Adam
optimizer = tf.keras.optimizers.RMSprop()

# Seleção do otimizador
# Se flag_optim = 0 => usa Gradiente Descentende codificado de forma
simples
# Se flag_optim = 1 => usa otimizador do keras
flag_optim = 1

# Loop de treinamento
for i in range(num_epocas):

    # Encapsula equações do modelo e função de custo com o
GradientTape
    with tf.GradientTape(persistent=True) as grad:
        # Equação do modelo de regressão linear
        y_prev = W*X_data + b

        # Função de custo erro quadrático médio
        custo = tf.reduce_mean(tf.square(y_data - y_prev))

    # Calculo do gradiente da função de custo em relação aos
parâmetros
    grad_W = grad.gradient(custo, W)
    grad_b = grad.gradient(custo, b)

    # Atualiza parâmetros com GD simples
    if flag_optim == 0:
        W.assign_sub(lr*grad_W)
        b.assign_sub(lr*grad_b)

    # Atualiza parâmetros com otimizador do Keras
    if flag_optim ==1:
        optimizer.apply_gradients(zip([grad_W, grad_b], [W, b]))

    # Imprime resultado da função de custo
    if i%50 == 0:
      print('Época: ', i, '-', 'custo =', custo.numpy())

# Imprime resultado final
print('\nCusto =', custo.numpy())
print('W final =', W[0][0].numpy())
print('b final =', b[0].numpy())
```

```
Época:  0 - custo = 310.92944
Época:  50 - custo = 246.67564
Época:  100 - custo = 210.43176
Época:  150 - custo = 189.46275
Época:  200 - custo = 182.11867
Época:  250 - custo = 181.69629
Época:  300 - custo = 181.57574
Época:  350 - custo = 181.45569
Época:  400 - custo = 181.33682
Época:  450 - custo = 181.21916
Época:  500 - custo = 181.10281
Época:  550 - custo = 180.9877
Época:  600 - custo = 180.87383
Época:  650 - custo = 180.76125
Época:  700 - custo = 180.64992
Época:  750 - custo = 180.53986
Época:  800 - custo = 180.43108
Época:  850 - custo = 180.32358
Época:  900 - custo = 180.2173
Época:  950 - custo = 180.1123

Custo = 180.01065
W final = 1.0350403
b final = 0.9866402
```

- A função `GradientTape()` é chamada com o argumento `persistent=True`. A razão disso é que quando se usa o objeto criado pela função `GradientTape()`, que no caso foi denominado por `grad`, com o método `gradient()` os recursos alocados para o seu cálculo são apagados. Assim, se for necessário usar o objeto `grad` mais do que uma vez deve-se indicar isso para a função `GradientTape()` usando `persistent=True`.

- No loop de treinamento o método `gradient()` é usado 2 vezes, portanto, se não usar `persistent=True`, quando for chamar a segunda vez, o objeto `grad` não vai existir mais.

- `persistent=True` permite usar o objeto `grad` várias vezes.

- A função de custo utilizada é o erro quadrático médio que é calculada utilizando as funções `tf.reduce_mean()` e `tf.square()` do TensorFlow.

- O método `assign_sub()` é utilizado para atualizar os parâmetros do modelo na direção oposta ao gradiente, de acordo com o método do Gradiente Descendente.

- A atualização dos parâmetros com o otimizador do Keras é realizada com o método `optimizer.apply_gradients()`. Esse método recebe o gradiente da função de custo e os parâmetros do modelo. A função `zip()` é usada ao para passar as variáveis para esse método para alinhar os gradientes com os parâmetros e eliminar eixos desnecessários.

Uso de otimizador do Keras

Para realizar o treinamento de um modelo com a função `GradientTape()` pode-se usar qualquer otimizador fornecido pelo Keras para atualizar os parâmetros do modelo, tais como, momento, RMSprop, Adam etc.

O ajuste do modelo implementado no código acima está preparado para utilizar tanto o Gradiente Descendente programado de forma simples com o método `assign_sub()`, como também um otimizador do Keras.

Para selecionar entre o GD simples e o otimizador do Keras, basta alterar a variável `flag_optim`:

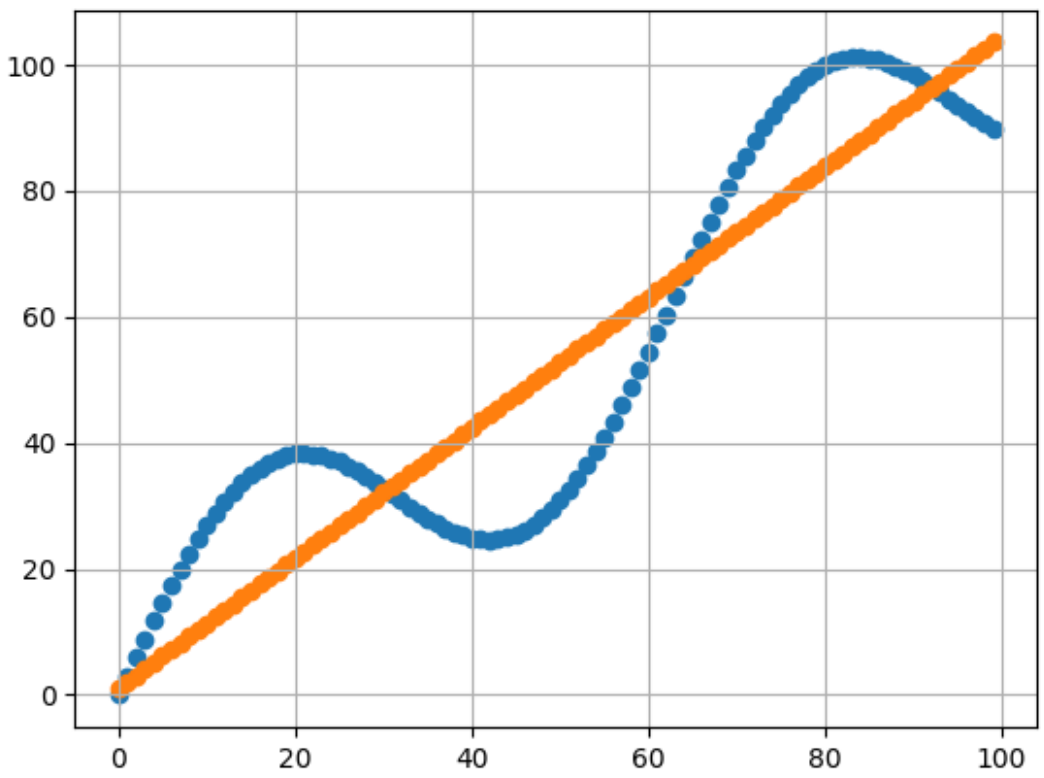
- Se `flag_optim = 0`, GD simples é usado
- Se `flag_optim = 1`, otimizador do Keras é usado

6.4 Verificação do modelo

Na célula a seguir é calculada a saída prevista usando o resultado da regressão linear, calculado o erro absoluto médio e feito um gráfico das duas curvas para visualização.

```
#Calcula resultado do ajuste
y_prev = X_data*W + b

# Calcula erro absoluto médio
erro = np.sum(np.abs(y_prev - y_data)/n_samples)

# Imprime erro
print('Erro médio quadrático =', erro)

# Realiza gráfico dos resultados
plt.scatter(X_data, y_data)
plt.scatter(X_data, y_prev)
plt.grid()
plt.show()
```

```
Erro médio quadrático = 11.849287
```


100
80
60
40
20
0
0
20
40
60
80
100