

## Automatic differentiation

<https://www.tensorflow.org/guide/autodiff>

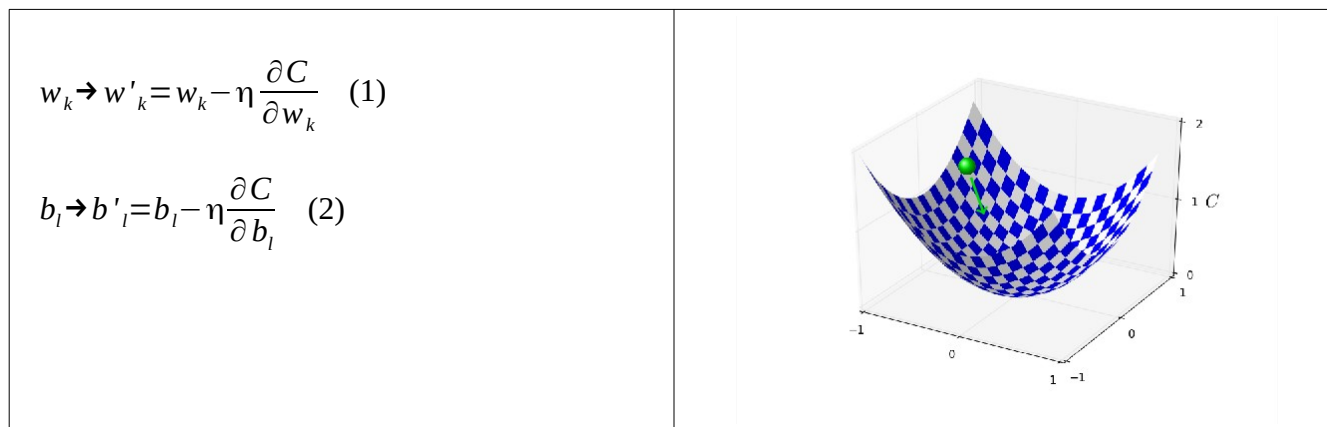
<https://insights.willogy.io/tensorflow-part-3-automatic-differentiation/>

<https://medium.com/analytics-vidhya/tf-gradienttape-explained-for-keras-users-cc3f06276f22>

[Programas no diretório ~/deep/keras/autodiff]

### I. O problema

Quando estudamos redes neurais, vimos que é necessário calcular as derivadas parciais da função custo em relação a cada peso e viés, para que possamos modificá-los para diminuir função custo, isto é, executar back-propagation:



Como é possível calcular as derivadas parciais eficientemente? Como as bibliotecas como TensorFlow e PyTorch calculam as derivadas parciais? As redes podem ser muito complexas e profundas, com desvios e execuções condicionais. Também pode haver operações complexas como convoluções, camadas recorrentes e camadas de atenção. Parece que TensorFlow faz alguma magia para calcular as derivadas parciais... Acredito que é importante saber como as derivadas parciais são calculadas, pois é fundamental para o funcionamento das redes neurais. Para isso, precisamos analisar o que acontece nas camadas de baixo nível de Keras/TensorFlow.

### II. Regra da cadeia

<https://www.khanacademy.org/math/ap-calculus-ab/ab-differentiation-2-new/ab-3-1a/a/chain-rule-review>

[https://pt.wikipedia.org/wiki/Regra\\_da\\_cadeia](https://pt.wikipedia.org/wiki/Regra_da_cadeia)

1) A regra da cadeia é uma fórmula para calcular a derivada da função composta.

$$(f \circ g)'(x) = f'(g(x))g'(x) \quad \text{ou} \quad \frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx} \quad (3)$$

## 2) Exemplo de regra da cadeia:

[https://en.wikipedia.org/wiki/Chain\\_rule](https://en.wikipedia.org/wiki/Chain_rule)

Vamos calcular a derivada da função:

$$f(x) = e^{\sin(x^2)} \quad (4)$$

Para isso, consideramos que a função  $f(x)$  é composta como  $f(x) = f(u(v(x)))$ :

$\begin{aligned} v(x) &= x^2 \rightarrow dv/dx = 2x \\ u(v) &= \sin(v) \rightarrow du/dv = \cos(v) \\ f(u) &= \exp(u) \rightarrow df/du = \exp(u) \end{aligned}$	(5)
--	-----

Aplicando a regra da cadeia:

$$\frac{df}{dx} = \frac{df}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}, \quad (6)$$

Obtemos a expressão algébrica para a derivada de  $f$ , em função de variáveis intermediárias  $u$  e  $v$ .

$$\frac{df}{dx} = e^u \cdot \cos(v) \cdot 2x \quad (7)$$

Se substituirmos as variáveis  $u$  e  $v$  pelas expressões correspondentes, obtemos a expressão algébrica da derivada parcial:

$$\frac{df}{dx} = e^{\sin(x^2)} \cdot \cos(x^2) \cdot 2x \quad (8)$$

## 3) Diferenciação automática.

À medida que a função  $f$  torna-se mais complexa, a expressão algébrica da derivada (8) fica cada vez mais longa e rapidamente torna-se impraticável escrevê-la ou calculá-la. Isto é especialmente verdade em deep learning, onde a função custo final é composta por muitas funções intermediárias. Além disso, muitas vezes não é possível escrever a derivada como uma expressão algébrica simples, como no caso de execução condicional. Diferenciação automática consegue superar essas dificuldades.

## II. GradientTape

Antes de prosseguirmos, vamos ver que TensorFlow consegue calcular facilmente as derivadas parciais da função  $f(x) = e^{\sin(x^2)}$  acima, usando uma API chamada GradientTape. Digamos que queiramos calcular, para  $x=2$ , as derivadas parciais  $df/du$ ,  $df/dv$  e  $df/dx$ . O programa abaixo faz isso.

<pre>#!/deep/keras/autodiff/autodiff2.py import numpy as np import matplotlib.pyplot as plt import tensorflow as tf  x = tf.Variable(2.0)  with tf.GradientTape(persistent=True) as tape:     v = tf.pow(x,2) # ou v=x**2     u = tf.sin(v)     f = tf.exp(u)  dfdu = tape.gradient(f, u); print("dfdu:", dfdu.numpy()) dudv = tape.gradient(u, v); print("dudv:", dudv.numpy()) dvdx = tape.gradient(v, x); print("dvdx:", dvdx.numpy()) dfdv = tape.gradient(f, v); print("dfdv:", dfdv.numpy()) dfdx = tape.gradient(f, x); print("dfdx:", dfdx.numpy()) dudx = tape.gradient(u, x); print("dudx:", dudx.numpy()) del tape</pre>	<pre>dfdu: 0.4691642 dudv: -0.65364367 dvdx: 4.0 dfdv: -0.3066662 dfdx: -1.2266648 dudx: -2.6145747</pre>
---	---

Programa: Autodiff2.py

Note que:

- 1) TensorFlow nunca calcula as expressões algébricas completas das derivadas parciais (equação 8). TensorFlow calcula apenas as derivadas parciais no ponto desejado ( $x=2$ ).
- 2) O programa acima utiliza as funções próprias do TensorFlow (`tf.pow`, `tf.sin`, `tf.exp`) para calcular a função. Se usasse uma única função de outras bibliotecas (como de `numpy` ou da biblioteca padrão de Python) seria impossível calcular as derivadas parciais.

Nota 1: A propriedade `persistent=True` faz com que a fita não seja apagada quando se calcula uma derivada parcial. Caso contrário, a fita seria apagada quando calculasse uma derivada parcial.

Nota 2: Quando não precisar mais do `tape`, apague-o para economizar memória: “`del tape`”.

### III. Diferenciação automática

Como podemos calcular, para  $x=2$  (por exemplo), as derivadas  $df/du$ ,  $df/dv$  e  $df/dx$ ? Há três possibilidades.

**Método 1)** A primeira é achar uma expressão algébrica para cada uma das derivadas. O problema desta abordagem é que as expressões ficam muito longas quando a função  $f$  for composta por muitas funções intermediárias. Quando há execução condicional, pode não ser possível escrever uma expressão para a derivada. Se quiser calcular  $(df/dx)(x=2)$  por este método (usando Octave), substituímos  $x$  por 2 na equação (8), obtendo:

```
>> x=2
>> dfdx=exp(sin(x^2))*cos(x^2)*2*x
dfdx = -1.2267
```

**Método 2)** A segunda é calcular a aproximação numérica da derivada, calculando:

$$\frac{df(x)}{dx} \cong \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

O problema desta abordagem é a imprecisão numérica. Além disso, precisamos “chutar” um valor adequado para  $\epsilon$ . Se quiser calcular  $(df/dx)(x=2)$  por este método:

```
>> x=2
>> epsilon=1e-3
>> x1=x-epsilon
>> x2=x+epsilon
>> f1=exp(sin(x1^2))
>> f2=exp(sin(x2^2))
>> dfdx=(f2-f1)/(2*epsilon)
dfdx = -1.2267
```

**Método 3)** A diferenciação automática (autodiff) é a terceira opção. É usada pelo TensorFlow e PyTorch. A autodiff não calcula a expressão algébrica inteira (equação 8) das derivadas nem usa diferenciação numérica calculando  $f(x\pm\epsilon)$ . Em vez disso, para cada função componente da  $f$ , calcula-se a sua derivada (equação 7).

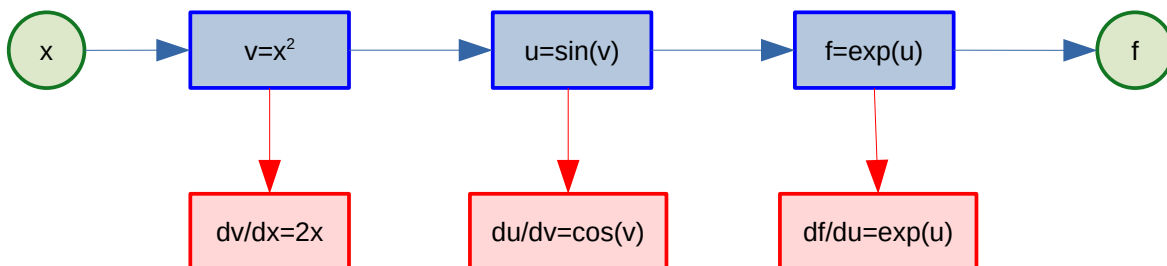


Figura: Autodiff de uma função composta.

Depois, usando as expressões das derivadas das funções constituintes, calcula-se o valor numérico da derivada apenas no ponto  $x$  desejado (no nosso exemplo,  $x=2$ ).

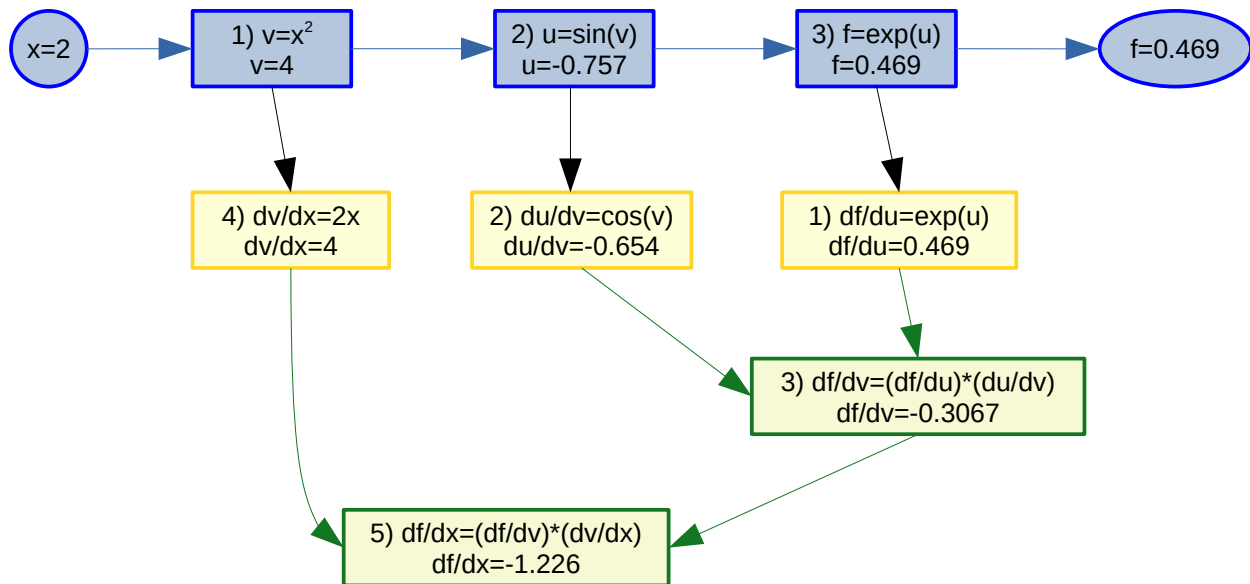


Figura: Autodiff com feed-forward e back-propagation.

O processo de calcular derivada parcial consiste de dois passos:

- Feed-forward onde se calcula  $f(2)$  passo a passo (quadrados azuis na figura acima, na ordem enumerada).
- Back-propagation, onde se calculam as derivadas parciais  $df/du$ ,  $df/dv$  e  $df/dx$  usando os valores calculados no feed-forward e multiplicando as derivadas parciais conforme regra da cadeia (os quadrados amarelos e verdes, na ordem enumerada). Repare que as derivadas são calculadas de frente para trás, daí o nome “back-propagation”.

Os valores calculados coincidem com os resultados obtidos pelo programa Autodiff2.py, método 1 e método 2. Com o que vimos, é possível intuir o que deve ser gravado no GradientTape do TensorFlow para poder calcular as derivadas: a sequência de todas as operações efetuadas envolvendo o parâmetro (peso ou viés) a respeito do qual gostaríamos de calcular a derivada parcial da função final  $f$ .

#### IV. Autodiff em perceptron com função custo

Como um exemplo mais próximo de rede neural, vamos calcular as derivadas parciais dos parâmetros  $w_1$ ,  $w_2$  e  $b$  num perceptron com função de custo MSE usando autodiff.

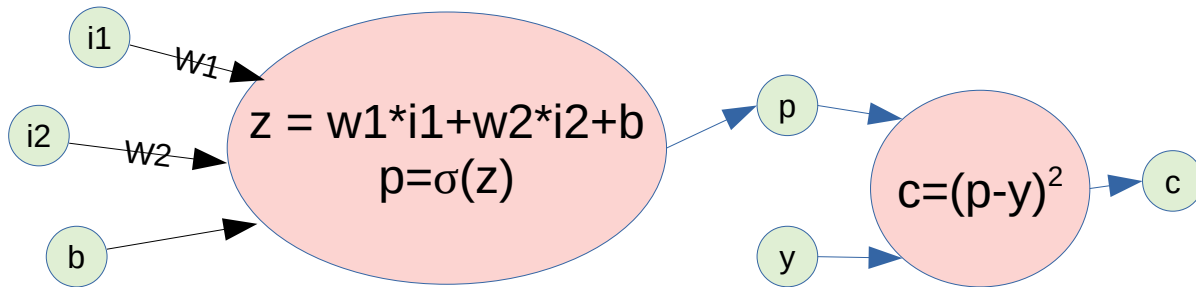


Figura: Um perceptron com função custo MSE.

Vamos supor  $i_1=0.5$ ,  $i_2=-0.3$ ,  $y=0.4$ ,  $w_1=-0.2$ ,  $w_2=0.2$ ,  $b=0.1$ . Para treinar a rede, precisamos calcular as derivadas parciais de  $c$  em relação a  $w_1$ ,  $w_2$  e  $b$ , isto é,  $dc/dw_1$ ,  $dc/dw_2$  e  $dc/db$ . Os valores de  $i_1$ ,  $i_2$  e  $y$  podem ser considerados constantes para o cálculo das derivadas. Vamos fazer primeiro o feed-forward:

$$z = w_1 i_1 + w_2 i_2 + b = -0.06$$

$$p = \sigma(z) = 0.48500$$

$$c = (p - y)^2 = 0.0072258$$

Vamos fazer back-propagation:

$$c = (p - y)^2 = p^2 - 2yp + y^2 \rightarrow dc/dp = 2p - 2y \rightarrow dc/dp(p=0.485) = 0.17$$

$$p = \sigma(z) \rightarrow dp/dz = \sigma(z)(1 - \sigma(z)) \rightarrow dp/dz(z=-0.06) = 0.24978$$

Nota: A derivada de sigmoide  $\sigma(z)$  é  $\sigma(z)(1 - \sigma(z))$ .

Para calcular as 3 derivadas parciais:

$$z = w_1 i_1 + w_2 i_2 + b \rightarrow$$

$$dz/dw_1 = i_1 = 0.5$$

$$dz/dw_2 = i_2 = -0.3$$

$$dz/db = 1$$

Portanto,

$$dc/dw_1 = (dc/dp) \times (dp/dz) \times (dz/dw_1) = 0.17001 \times 0.24978 \times 0.5 = 0.021233$$

$$dc/dw_2 = (dc/dp) \times (dp/dz) \times (dz/dw_2) = 0.17001 \times 0.24978 \times -0.3 = -0.012740$$

$$dc/db = (dc/dp) \times (dp/dz) \times (dz/db) = 0.17001 \times 0.24978 \times 1 = 0.042465$$

Vamos verificar se os resultados acima calculados manualmente estão corretos calculando as mesmas derivadas com GradientTape do TensorFlow:

```
#!/deep/keras/autodiff/perceptron1.py
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

i1 = tf.constant(0.5)
i2 = tf.constant(-0.3)
y = tf.constant(0.4)

w1 = tf.Variable(-0.2)
w2 = tf.Variable(0.2)
b = tf.Variable(0.1)

with tf.GradientTape(persistent=True) as tape:
    z = w1*i1+w2*i2+b
    p = tf.math.sigmoid(z)
    c = (p-y)**2

dcdw1 = tape.gradient(c, w1); print("dcdw1:", dcdw1.numpy())
dcdw2 = tape.gradient(c, w2); print("dcdw2:", dcdw2.numpy())
dcdb = tape.gradient(c, b); print("dcdb: ", dcdb.numpy())
```

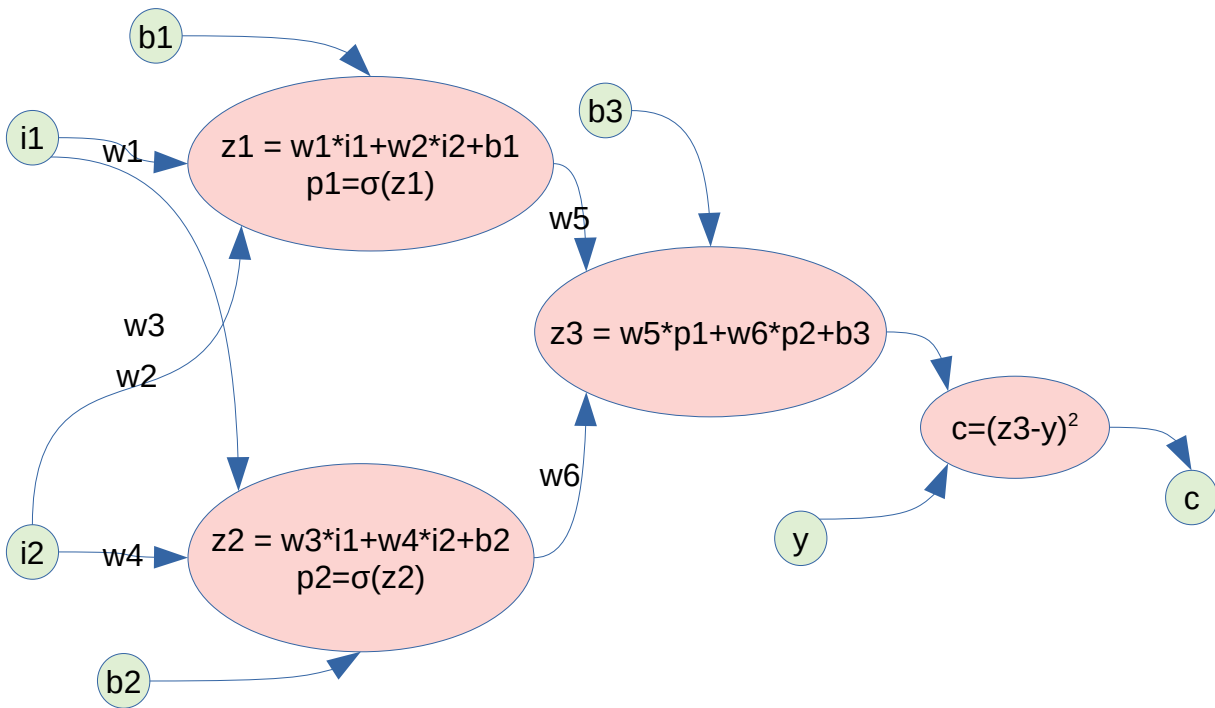
Programa: Perceptron1.py

Saída:

```
dcdw1: 0.021232
dcdw2: -0.0127392
dcdb: 0.042464
```

Os cálculos feitos manualmente coincidem com os do TensorFlow.

[PSI3472-2023 Lição de casa #2 das aulas 1 e 2 (vale 5 pontos)] Considere a seguinte rede neural:



Escreva um programa TensorFlow que calcula as derivadas parciais  $dc/dw_1, dc/dw_2, dc/dw_3, dc/dw_4, dc/dw_5, dc/dw_6, dc/db_1, dc/db_2$  e  $dc/db_3$  quando:

$w_1 = -0.2, w_2 = 0.5, w_3 = 0.9, w_4 = -0.6, w_5 = 0.2, w_6 = -0.4, b_1 = 0.4, b_2 = -0.2, b_3 = -0.5$

$i_1 = 0.6, i_2 = -0.3, y = 1$

```
Solução privada em ~/deep/keras/autodiff/regressao1.py
dcdw1: -0.09824643
dcdw2: 0.049123216
dcdw3: 0.184564
dcdw4: -0.092282
dcdw5: -1.751102
dcdw6: -2.0625236
dcdb1: -0.16374405
dcdb2: 0.30760664
dcdb3: -3.2887366
```



## V. Autodiff com convolução, relu e MAE

Agora, vamos ver como autodiff funciona numa rede convolucional com função de ativação relu.

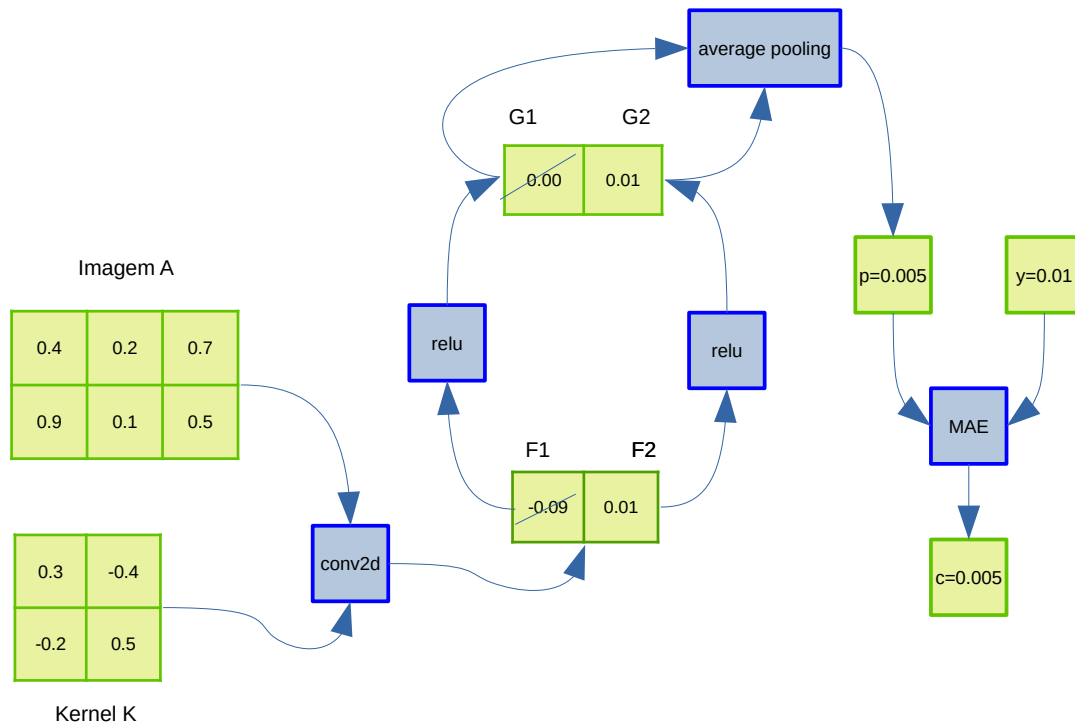


Figura: Autodiff numa rede convolucional com ativação relu.

Suponha que queremos treinar a rede acima, escolhendo a convolução  $K$  que minimiza o custo  $c$ . Precisamos calcular  $dc/dK$ .

Vamos fazer feed-forward:

$$F = \text{conv2d}(A) = [-0.09, 0.01]$$

$$G = \text{relu}(F) = [0.00, 0.01]$$

$$p = \text{avePool}(G) = 0.005$$

$$c = |p - y| = |0.005 - 0.01| = 0.005$$

Nota: Se quiser fazer convolução em Octave ou Matlab, deve fazer antes a rotação do Kernel  $K$  por 180 graus.

$$F = \text{conv2d}(A, \text{rot90}(K, 2)) = [-0.09, 0.01]$$

Vamos fazer back-propagation.

$c = |p - y| \rightarrow dc/dp = -1$  se  $p < y$ ;  $+1$  se  $p > y$ ; indefinido se  $p = y$ .  $\rightarrow dc/dp = -1$ , pois  $p < y$ .

Aqui, o cálculo da derivada se divide em dois caminhos:

$p = \text{avePool}(G) = (G_1 + G_2) / 2 \rightarrow dp/dG_1 = 0.5$ ;  $dp/dG_2 = 0.5$ .

$G_1 = \text{relu}(F_1) \rightarrow dG_1/dF_1 = 0$  se  $F_1 < 0$ ;  $+1$  se  $F_1 > 0$ ; indefinido se  $F_1 = 0$ .  $\rightarrow dG_1/dF_1 = 0$  (pois  $F_1 < 0$ ).

$G_2 = \text{relu}(F_2) \rightarrow dG_2/dF_2 = 0$  se  $F_2 < 0$ ;  $+1$  se  $F_2 > 0$ ; indefinido se  $F_2 = 0$ .  $\rightarrow dG_2/dF_2 = 1$  (pois  $F_2 > 0$ ).

$F_1 = A_{11}K_{11} + A_{12}K_{12} + A_{21}K_{21} + A_{22}K_{22} = 0.4K_{11} + 0.2K_{12} + 0.9K_{21} + 0.1K_{22} \rightarrow dF_1/dK = [0.4, 0.2; 0.9, 0.1]$

$F_2 = A_{12}K_{11} + A_{13}K_{12} + A_{22}K_{21} + A_{23}K_{22} = 0.2K_{11} + 0.7K_{12} + 0.1K_{21} + 0.5K_{22} \rightarrow dF_2/dK = [0.2, 0.7; 0.1, 0.5]$

Vamos aplicar a regra da cadeia:

$dc/dK = (dc/dp) \times (dp/dG) \times (dG/dF) \times (dF/dK) =$

$(dc/dp) \times (dp/dG_1) \times (dG_1/dF_1) \times (dF_1/dK) + (dc/dp) \times (dp/dG_2) \times (dG_2/dF_2) \times (dF_2/dK) =$

$(-1) \times 0.5 \times 0 \times [0.4, 0.2; 0.9, 0.1] + (-1) \times 0.5 \times 1 \times [0.2, 0.7; 0.1, 0.5] =$

$(-0.5) \times [0.2, 0.7; 0.1, 0.5] = [-0.1, -0.35; -0.05, -0.25]$

Agora, vamos verificar as nossas contas manuais estão corretas (ou não), comparando  $dc/dK$  acima com o obtido pelo TensorFlow.

```
#~/deep/keras/autodiff/conv1.py
import numpy as np
import tensorflow as tf

A_in=np.array([[0.4, 0.2, 0.7],[0.9, 0.1, 0.5]], dtype=np.float32)
A_in=np.reshape(A_in, (1, 2, 3, 1))
A=tf.constant(A_in, dtype=tf.float32)

K_in=np.array([[0.3, -0.4],[-0.2, 0.5]], dtype=np.float32)
K_in=np.reshape(K_in, (2, 2, 1, 1))
K=tf.Variable(K_in, dtype=tf.float32)

y=tf.constant(0.01, dtype=tf.float32)

with tf.GradientTape(persistent=True) as tape:
    F=tf.nn.conv2d(A, K, strides=(1, 1, 1, 1), padding='VALID')
    F=tf.reshape(F, (1,1,2,1)); print("F",F.numpy().reshape(2,))
    G=tf.nn.relu(F); print("G",G.numpy().reshape(2,))
    p=tf.nn.avg_pool2d(G, (1,2), (1,1), "VALID"); p=tf.reshape(p, (1,))
    print("p",p.numpy().reshape(1,))
    c=tf.abs(p-y); print("c",c.numpy().reshape(1,))

dcdp = tape.gradient(c, p); print("dcdp:",dcdp.numpy().reshape(1,))
dpdG = tape.gradient(p, G); print("dpdG:",dpdG.numpy().reshape(2,))
dGdF = tape.gradient(G, F); print("dGdF:",dGdF.numpy().reshape(2,))
dFdK = tape.gradient(F, K); print("dFdK:",dFdK.numpy().reshape(2,2))
dcdK = tape.gradient(c, K); print("dcdK:",dcdK.numpy().reshape(2,2))
dcdG = tape.gradient(c, G); print("dcdG:",dcdG.numpy().reshape(2,))
dcdF = tape.gradient(c, F); print("dcdF:",dcdF.numpy().reshape(2,))
```

Programa: Conv1.py

Saída:

```
F [-0.09      0.01000001]
G [0.         0.01000001]
p [0.005]
c [0.005]
dcdp: [-1.]
dpgG: [0.5 0.5]
dGdF: [0. 1.]
dFdK:
[[0.6 0.9]
 [1.  0.6]]
dcdK:
[[-0.1  -0.35]
 [-0.05 -0.25]]
dcdG: [-0.5 -0.5]
dcdF: [-0.  -0.5]
```

Podemos verificar que  $dc/dK$  calculado manualmente coincide com  $dc/dK$  calculado pelo TensorFlow.

## Camada personalizada (custom layer)

Programas em ~/deep/keras/densa/fromScratch.

1) Copio abaixo o programa “regression.py” (da apostila densakeras-ead) com pequenas alterações que não fazem diferença no resultado final:

```
# from1.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import tensorflow.keras.activations as activations
from tensorflow.keras import optimizers
import numpy as np

model = keras.Sequential()
model.add(layers.Input(shape=(2,)))
model.add(layers.Dense(2))
model.add(layers.Activation(activations.sigmoid))
model.add(layers.Dense(2))
sgd=optimizers.SGD(learning_rate=1)
model.compile(optimizer=sgd, loss='mse')

AX = np.matrix('0.9 0.1; 0.1 0.9', dtype='float32')
AY = np.matrix('0.1 0.9; 0.9 0.1', dtype='float32')

model.fit(AX, AY, epochs=120, batch_size=1, verbose=0)

QX = np.matrix('0.9 0.1; 0.1 0.9; 0.8 0.0; 0.2 0.9', dtype='float32')
print("QX="); print(QX)
QP=model.predict(QX, verbose=0)
print("QP="); print(QP)
```

Programa from1.py.

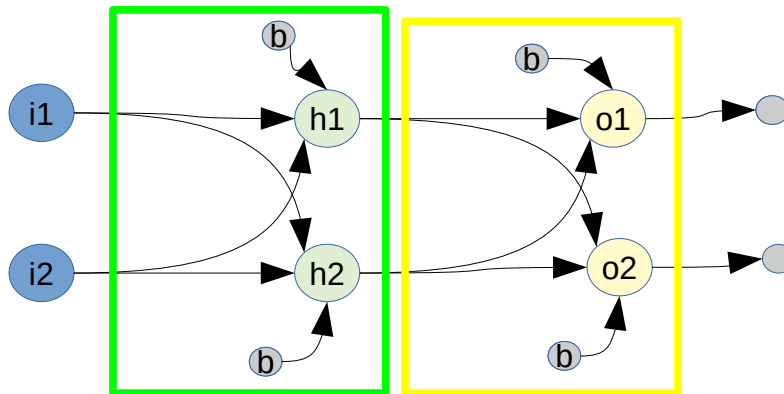


Figura: Estrutura da rede neural do programa from1.py

Saída:

QX=

```
[[0.9 0.1]
 [0.1 0.9]
 [0.8 0. ]
 [0.2 0.9]]
```

QP=

```
[[0.09999999 0.9      ]
 [0.9        0.10000002]
 [0.13876095 0.9329134]
 [0.82762444 0.1424768 ]]
```

2) Agora que sabemos como TensorFlow calcula as derivadas parciais, estamos prontos para implementar camadas personalizadas. Algumas vezes, pode ser necessário criar camadas diferentes das que estão pré-implementadas em Keras/TensorFlow.

Para criar uma camada personalizada, basta criar uma classe derivada da classe Layer de Keras. Não é necessário trabalhar explicitamente com GradientTape – classe Layer vai cuidar disso. Porém, é preciso usar apenas as operações do TensorFlow. Caso contrário, a operação não será gravada no GradientTape e não será possível calcular as derivadas parciais pela autodiff.

Para aprendermos implementar novas camadas em Keras/TensorFlow, vamos substituir as duas camadas Dense por camadas personalizadas MyDense.

```
# from2.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import tensorflow.keras.activations as activations
from tensorflow.keras import optimizers
import numpy as np

#https://www.tensorflow.org/tutorials/customization/custom_layers
#http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf
class MyDense(tf.keras.layers.Layer):
    #Funciona para qualquer numero de entradas e saidas
    def __init__(self, output_dim):
        super(MyDense, self).__init__()
        self.num_outputs = output_dim

    def build(self, input_shape):
        self.W = self.add_weight("W",
            [input_shape[1], self.num_outputs], initializer="glorot_uniform")
        self.b = self.add_weight("b", [1, self.num_outputs], initializer="zeros")

    def call(self, inputs):
        z = tf.matmul(inputs, self.W) + self.b
        return z

model = keras.Sequential()
model.add(layers.Input(shape=(2,)))
model.add(MyDense(2))
model.add(layers.Activation(activations.sigmoid))
model.add(MyDense(2))
sgd=optimizers.SGD(learning_rate=1)
model.compile(optimizer=sgd, loss='mse')

AX = np.matrix('0.9 0.1; 0.1 0.9', dtype='float32')
AY = np.matrix('0.1 0.9; 0.9 0.1', dtype='float32')

model.fit(AX, AY, epochs=120, batch_size=1, verbose=0)

QX = np.matrix('0.9 0.1; 0.1 0.9; 0.8 0.0; 0.2 0.9', dtype='float32')
print("QX="); print(QX)
QP=model.predict(QX, verbose=0)
print("QP="); print(QP)
```

Programa from2.py.

Saída:

QX=

```
[[0.9 0.1]
 [0.1 0.9]
 [0.8 0. ]
 [0.2 0.9]]
```

QP=

```
[[0.099999999 0.9          ]
 [0.900000004 0.100000002]
 [0.13584077  0.8724962   ]
 [0.8098951   0.18530202]]
```

Podemos ver que tanto programa from1.py como from2.py efetuam corretamente a regressão. As duas saídas não são exatamente iguais devido à inicialização aleatória de pesos e vieses.

Referências:

<https://www.guru99.com/tensor-tensorflow.html>

[https://www.tensorflow.org/tutorials/customization/custom\\_layers](https://www.tensorflow.org/tutorials/customization/custom_layers)

[http://introtodeeplearning.com/slides/6S191\\_MIT\\_DeepLearning\\_L1.pdf](http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf)

*Exercício:* Pense em alguma alteração da camada MyDense que possa ser útil para alguma aplicação.

3) Para que fique mais claro o que está acontecendo na camada MyDense, vamos usar variáveis comuns de TensorFlow (em vez de add\_weight) e calcular manualmente a multiplicação matricial. Vamos fixar MyDense para ter número de entradas 2 e saídas também 2.

Além disso, vamos “chutar” manualmente os valores iniciais dos pesos imitando a inicialização “glorot\_uniform” (em vez de inicializar automaticamente). A iniciação “glorot\_uniform” escolhe amostras da distribuição uniforme no intervalo [-limit, +limit] onde

$$limit = \sqrt{\frac{6}{fanin + fanout}} = \sqrt{\frac{6}{2+2}} \simeq 1,225 \quad .$$

```
# from3.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import tensorflow.keras.activations as activations
from tensorflow.keras import optimizers
import numpy as np

#https://www.tensorflow.org/tutorials/customization/custom_layers
#http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf
class MyDense(tf.keras.layers.Layer):
    #So funciona para 2 entradas e 2 saidas
    def __init__(self):
        super(MyDense, self).__init__()
        self.num_outputs = 2

    def build(self, input_shape):
        self.W = tf.Variable( [ [0.3, -0.8], [-0.4, 0.2] ], dtype=tf.float32, name="W" )
        self.b = tf.Variable( [0,0], dtype=tf.float32, name="b");

    def call(self, inputs):
        # A dimensao 0 (indicada por ":") e' para permitir rodar batches.
        z0 = inputs[:,0]*self.W[0,0]+inputs[:,1]*self.W[1,0]+self.b[0];
        z1 = inputs[:,0]*self.W[0,1]+inputs[:,1]*self.W[1,1]+self.b[1];
        z = tf.stack([z0,z1], axis=1, name="z")
        return z

model = keras.Sequential()
model.add(layers.Input(shape=(2,)))
model.add(MyDense())
model.add(layers.Activation(activations.sigmoid))
model.add(MyDense())
sgd=optimizers.SGD(learning_rate=1)
model.compile(optimizer=sgd,loss='mse')

AX = np.matrix('0.9 0.1; 0.1 0.9',dtype='float32')
AY = np.matrix('0.1 0.9; 0.9 0.1',dtype='float32')

#batch_size deve ser 1 ou 2
model.fit(AX, AY, epochs=120, batch_size=1, shuffle=False, verbose=0)

QX = np.matrix('0.9 0.1; 0.1 0.9; 0.8 0.0; 0.2 0.9',dtype='float32')
print("QX="); print(QX)
QP=model.predict(QX,verbose=0)
print("QP="); print(QP)
```

Podemos ver que o programa continua funcionando.

Saída:

QX=

```
[[0.9 0.1]
 [0.1 0.9]
 [0.8 0. ]
 [0.2 0.9]]
```

QP=

```
[[0.1          0.9          ]
 [0.900000004  0.100000002]
 [0.11033662   0.89271045]
 [0.8298143    0.16628951]]
```





Agora, podemos debugar o que acontece dentro da camada MyDense.

As últimas saídas do treino:

```
inputs= tf.Tensor([[0.9 0.1]], shape=(1, 2), dtype=float32)
z= tf.Tensor([[ -1.9235604 -1.9927275]], shape=(1, 2), dtype=float32)
inputs= tf.Tensor([[0.12746507 0.11996861]], shape=(1, 2), dtype=float32)
z= tf.Tensor([[0.1 0.9]], shape=(1, 2), dtype=float32)

inputs= tf.Tensor([[0.1 0.9]], shape=(1, 2), dtype=float32)
z= tf.Tensor([[ -1.2987914 0.6264709]], shape=(1, 2), dtype=float32)
inputs= tf.Tensor([[0.21436849 0.6516889 ]], shape=(1, 2), dtype=float32)
z= tf.Tensor([[0.9 0.10000002]], shape=(1, 2), dtype=float32)
```

A impressão acima mostra que a rede converteu [0.9 0.1] em [0.1 0.9] e vice-versa. Também é possível observar as ativações entre as duas camadas.

Saídas durante o teste:

```
inputs= tf.Tensor(
[[0.9 0.1]
 [0.1 0.9]
 [0.8 0. ]
 [0.2 0.9]], shape=(4, 2), dtype=float32)
z= tf.Tensor(
[[ -1.9235604 -1.9927275]
 [ -1.2987914 0.6264709]
 [ -1.8094821 -1.9071858]
 [ -1.3948787 0.42      ]], shape=(4, 2), dtype=float32)
inputs= tf.Tensor(
[[0.12746507 0.11996861]
 [0.21436849 0.6516889 ]
 [0.14070073 0.12929733]
 [0.19863003 0.60348326]], shape=(4, 2), dtype=float32)
z= tf.Tensor(
[[0.1 0.9 ]
 [0.9 0.10000002]
 [0.11852115 0.8831827 ]
 [0.82446176 0.17439479]], shape=(4, 2), dtype=float32)
```

[PSI3472-2023. Aula 2. Fim.]