[PSI3472-2023. Aula 2. Início.]

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
https://www.deeplearningbook.com.br/algoritmo-backpropagation-parte1-grafos-computacionais-e-chain-rule/
[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:

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \quad (1)$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \quad (2)$$

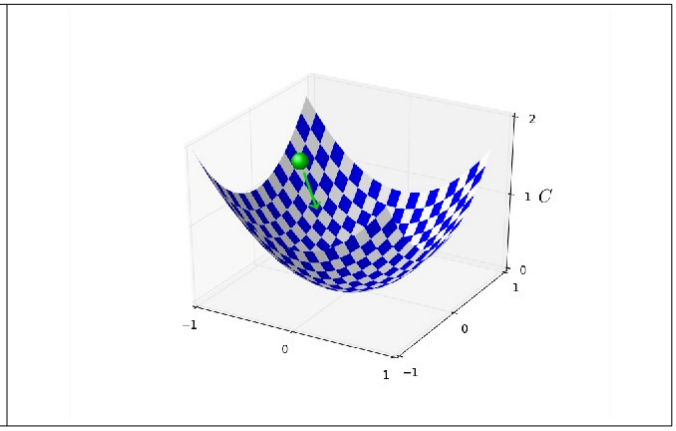

Figura 1:

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 mágica para calcular as derivadas parciais... É importante saber como as derivadas parciais são calculadas, pois é fundamental para entender 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

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

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

2) Exemplo de regra da cadeia:
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))*:

$v(x)=x^2 \rightarrow dv/dx = 2x$ $u(v)=sin(v) \rightarrow du/dv = cos(v)$ $f(u)=exp(u) \rightarrow df/du = exp(u)$	(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:

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

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 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

[https://www.tensorflow.org/guide/autodiff?hl=pt-br]

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

```
#~/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
```

```
dfdu: 0.4691642
dudv: -0.65364367
dvdx: 4.0
dfdv: -0.3066662
dfdx: -1.2266648
dudx: -2.6145747
```

Programa 1: Autodiff2.py

Note que:

1. TensorFlow nunca calcula as expressões algébricas completas das derivadas (equação 8). TensorFlow calcula apenas as derivadas 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 funções de outras bibliotecas (como de *numpy* ou da biblioteca padrão de Python) seria impossível calcular as derivadas.

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 intermediárias *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. Quando há execução condicional, pode não ser possível escrever uma expressão para a derivada. Se quiser calcular *(df/dx)(x=2)* por este método, 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:

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

O problema desta abordagem é a imprecisão numérica. Além disso, precisamos "chutar" um valor adequado para ε. 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 das derivadas (equação 8) nem usa diferenciação numérica *f(x±ε)*. Em vez disso, para cada função componente da *f*, calcula-se a sua derivada (equação 7). Na figura 2, os retângulos azuis calculam *f*(x) passo a passo. Os retângulos vermelhos são as derivadas de cada um desses passos.

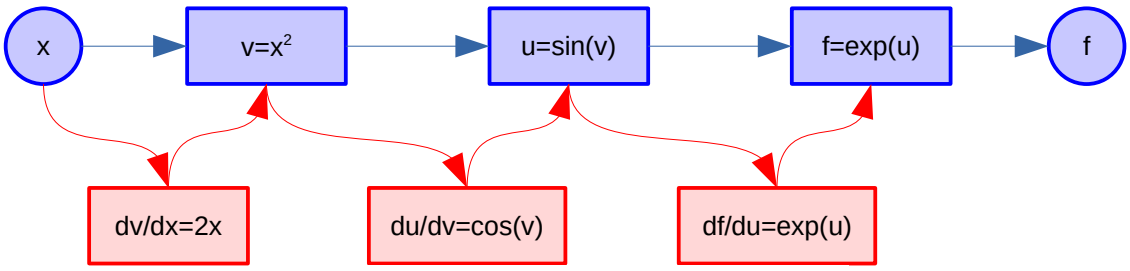

Figura 2: Autodiff de uma função composta.

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 3).

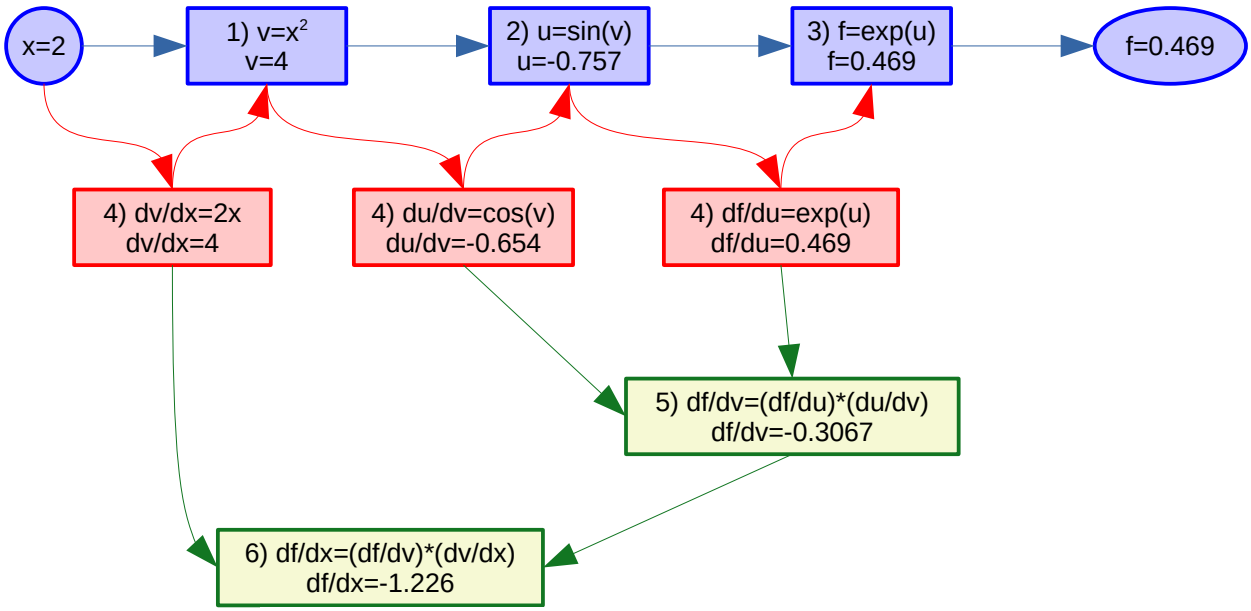

Figura 3: Autodiff com feed-forward e back-propagation.

O processo de calcular derivada parcial consiste de dois passos:

- Feed-forward onde se calcula $f(2)$ passo a passo (os quadrados azuis na figura 3, na ordem enumerada).
- Back-propagation, onde se calculam as derivadas parciais *df/du*, *du/dv* e *dv/dx* (retângulos vermelhos) e *df/dv* e *df/dx* (retângulos verdes) usando os valores calculados no feed-forward e multiplicando as derivadas parciais conforme a regra da cadeia.

Os valores calculados coincidem com os resultados obtidos pelo programa 1 (autodiff2.py) e pelos métodos 1 e 2 acima.

IV. Autodiff em perceptron com função custo

Como um exemplo mais próximo da rede neural, vamos calcular as derivadas parciais do custo c em relação aos parâmetros (w_1, w_2, b) num neurônio com um único exemplo de treino (x_1, x_2, y).

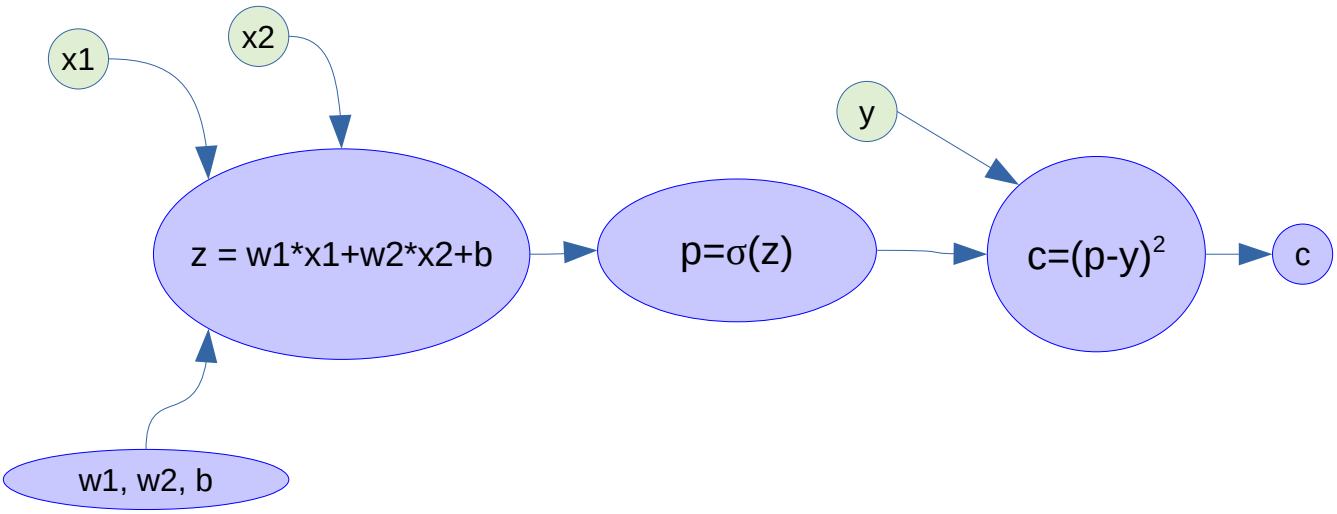

Figura 4: Um neurônio com função custo MSE.

Vamos supor os valores do exemplo de treino (x_1=0.5, x_2=-0.3, y=0.4) e parâmetros (w_1=-0.2, w_2=0.2, b=0.1). Para treinar a rede, precisamos calcular as derivadas parciais $\partial c/\partial w_1$, $\partial c/\partial w_2$ e $\partial c/\partial b$. Os valores do exemplo de treino x_1, x_2 e y podem ser considerados constantes para o cálculo das derivadas parciais.

Vamos fazer primeiro o feed-forward (elipses vermelhos da figura 5):

$z = w_1x_1+w_2x_2+b = -0.06$
$p = \sigma(z) = 0.48500$
$c = (p-y)^2 = 0.0072258$

Depois, podemos calcular e armazenar as derivadas de cada passo (retângulos vermelhos da figura 5):

$z=w_1x_1+w_2x_2+b \rightarrow$
  $\partial z/\partial w_1 = x_1 = 0.5$
  $\partial z/\partial w_2 = x_2 = -0.3$
  $\partial z/\partial b = 1$
$p = \sigma(z) \rightarrow dp/dz = \sigma(z)(1-\sigma(z)) \rightarrow dp/dz(z=-0.06) = 0.24978$
  Nota: A derivada de sigmoide $\sigma(z)$ é $\sigma(z)(1-\sigma(z))$.
$c = (p-y)^2 = p^2 - 2yp + y^2 \rightarrow dc/dp = 2p - 2y \rightarrow dc/dp(p=0.485) = 0.17$

Por fim, faz backpropagation (retângulos verdes da figura 5):

$dc/dz = (dc/dp) \times (dp/dz) = 0.17001 \times 0.24978 = 0.042465$
$\partial c/\partial w_1 = (dc/dz) \times (\partial z/\partial w_1) = 0.042465 \times 0.5 = 0.021233$
$\partial c/\partial w_2 = (dc/dz) \times (\partial z/\partial w_2) = 0.042465 \times -0.3 = -0.012740$
$\partial c/\partial b = (dc/dz) \times (\partial z/\partial b) = 0.042465 \times 1 = 0.042465$

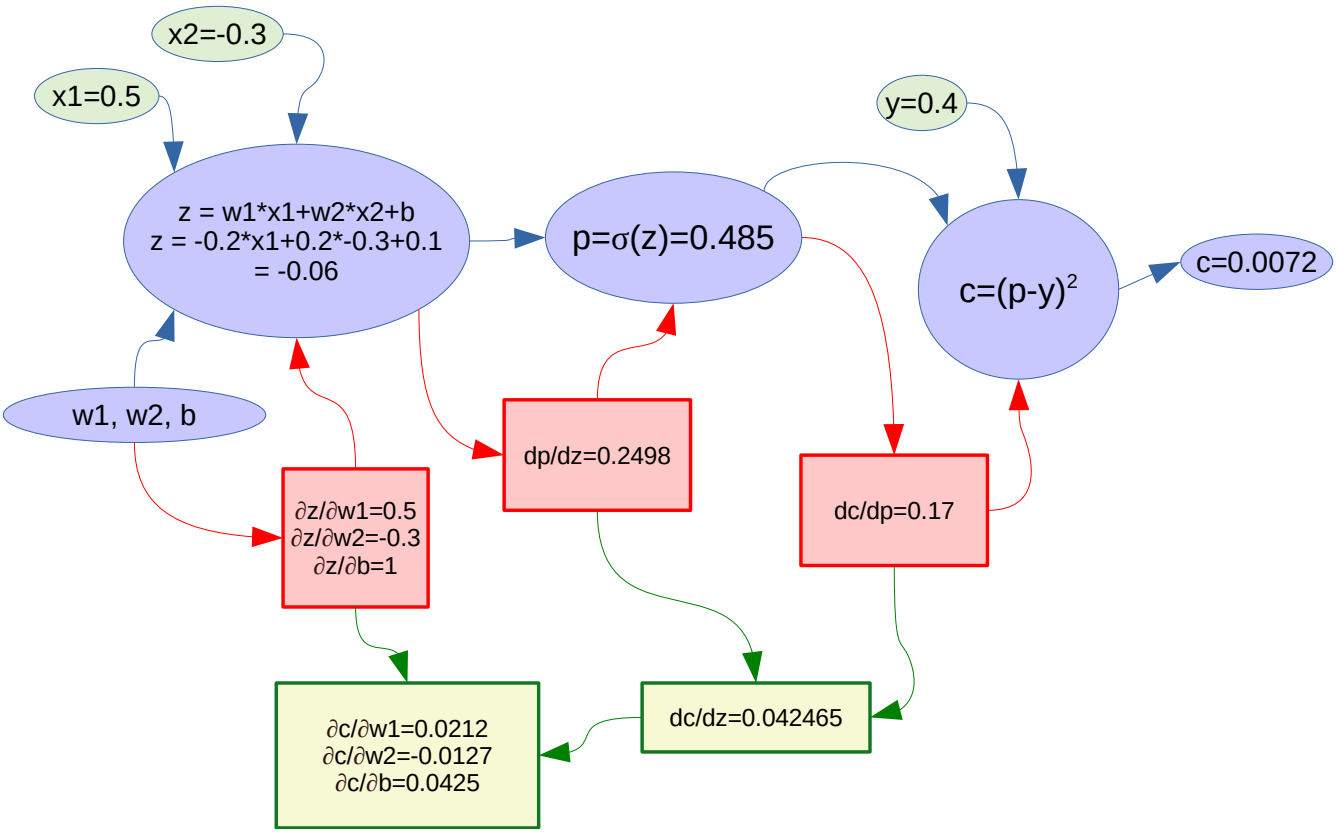

Figura 5: Feedforward e backpropagation do neurônio da figura 4 com um único exemplo de treino.

Podemos verificar que os resultados acima estão corretos calculando as mesmas derivadas com GradientTape do TensorFlow (programa 2).

```python
#~/deep/keras/autodiff/perceptron1.py - escalar
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

x1 = tf.constant(0.5)
x2 = 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*x1+w2*x2+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())
```

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

(a) perceptron1.py: usa escalares.

```python
#~/deep/keras/autodiff/perceptron1b.py - tensor
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

x = tf.constant((0.5, -0.3), name='x')
y = tf.constant(0.4)

w = tf.Variable((-0.2, 0.2), name='w')
b = tf.Variable(0.1)

with tf.GradientTape(persistent=True) as tape:
  z = tf.tensordot(w,x,1)+b
  p = tf.math.sigmoid(z)
  c = (p-y)**2

dcdw = tape.gradient(c, w); print("dcdw:",dcdw.numpy())
dcdb = tape.gradient(c, b); print("dcdb:",dcdb.numpy())
```

```
Saída
dcdw: [ 0.021232  -0.0127392]
dcdb: 0.042464
```

(b) perceptron1b.py: usa tensores.

Programa 2: Programas que calculam as derivadas parciais da figura 5 usando GradientTape, com variáveis escalares ou tensores.

Por que percorrer o grafo de frente para trás e não no sentido contrário? É que, percorrendo de frente para trás, estaremos calculando dc/dv para cada uma das variáveis v do grafo, que é o que precisamos obter para calcular as derivadas parciais do custo. Se percorréssemos o grafo de trás para frente, poderíamos calcular $\partial v/\partial b$ ou $\partial v/\partial w_1$ ou $\partial v/\partial w_2$ para cada variável v do grafo, que não é o que queremos.

IV. Autodiff em neurônio usando lote com 2 exemplos de treino

Considere a rede da figura 4 agora alimentado com um lote contendo duas amostras de treno:

Amostra 1: x_{11}= 0.5, x_{12}= -0.3, y_1= 0.4.

Amostra 2: x_{21}= 0.2, x_{22}= -0.4, y_2= 0.6.

Os parâmetros iniciais da rede continuam sendo w_1=-0.2, w_2=0.2, b=0.1. A função custo c é a média dos custos c_1 e c_2 das duas amostras, isto é $c = (c_1+c_2)/2$.

Para calcular as derivadas parciais $\partial c/\partial b$, $\partial c/\partial w_1$ e $\partial c/\partial w_2$, devemos fazer as contas mostradas na figura 5 duas vezes, uma vez para cada amostra de treino, obtendo ($\partial c_1/\partial b$, $\partial c_1/\partial w_1$, $\partial c_1/\partial w_2$) e ($\partial c_2/\partial b$, $\partial c_2/\partial w_1$, $\partial c_2/\partial w_2$). Depois, deve tirar as médias:

$\partial c/\partial b = (\partial c_1/\partial b + \partial c_2/\partial b)/2$,

$\partial c/\partial w_1 = (\partial c_1/\partial w_1 + \partial c_2/\partial w_1)/2$,

$\partial c/\partial w_2 = (\partial c_1/\partial w_2 + \partial c_2/\partial w_2)/2$.

Outra maneira de enxergar o mesmo problema é representar o problema como na figura 6. Para calcular as derivadas parciais $\partial c/\partial b$, $\partial c/\partial w_1$ e $\partial c/\partial w_2$, devemos percorrer os dois caminhos:

$\partial c/\partial w_1 = (\partial c/\partial z_1)*(\partial z_1/\partial w_1) + (\partial c/\partial z_2)*(\partial z_2/\partial w_1)$

$\partial c/\partial w_2 = (\partial c/\partial z_1)*(\partial z_1/\partial w_2) + (\partial c/\partial z_2)*(\partial z2/\partial w_2)$

$\partial c/\partial b = (\partial c/\partial z_1)*(\partial z_1/\partial b) + (\partial c/\partial z_2)*(\partial z_2/\partial b)$

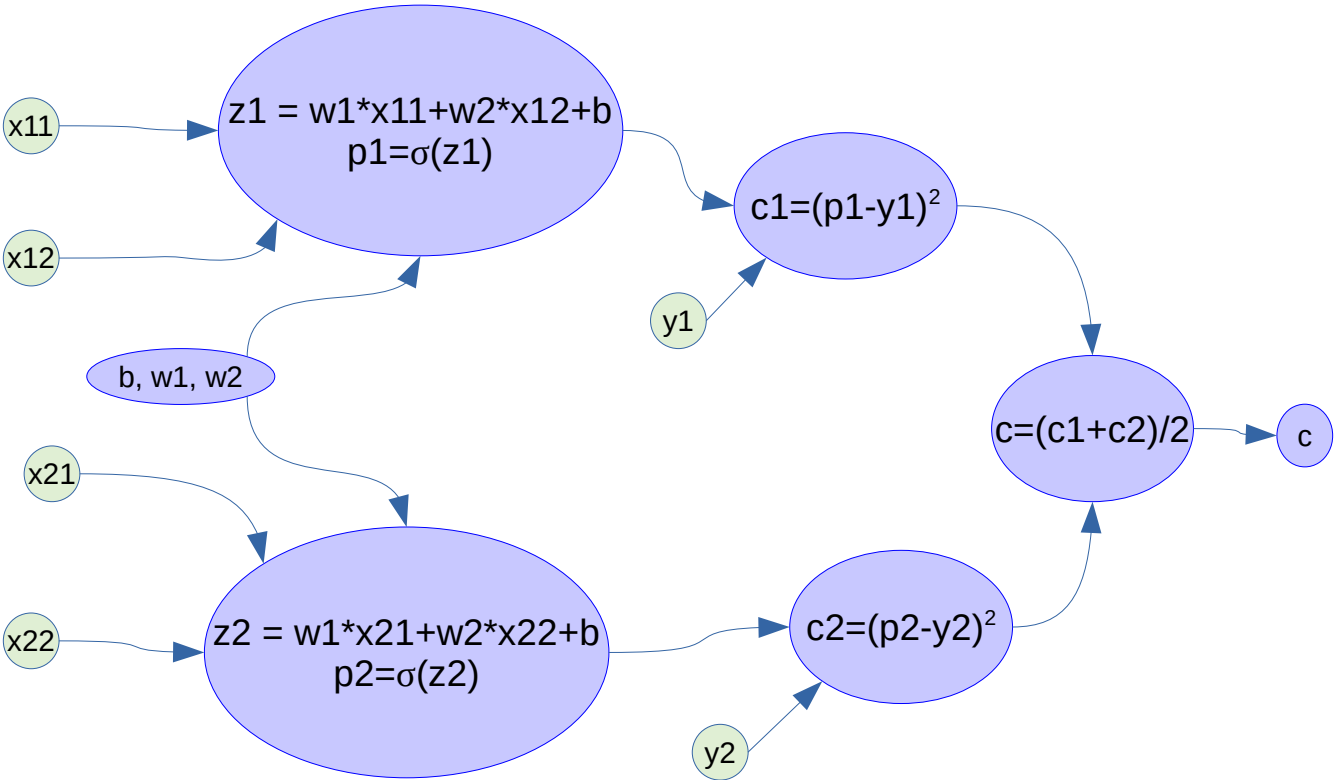

Figura 6: Feedforward e backpropagation do neurônio da figura 4 usando lote com dois exemplos de treino.

O programa 4 calcula as derivadas parciais dos parâmetros para os exemplos de treino dados usando GradientTape do TensorFlow. O programa obteve as três derivadas parciais necessárias para fazer a descida da gradiente (em amarelo).

```
#~/deep/keras/autodiff/perceptron2.py
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

x11 = tf.constant(0.5)
x12 = tf.constant(-0.3)
y1  = tf.constant(0.4)

x21 = tf.constant(0.2)
x22 = tf.constant(-0.4)
y2  = tf.constant(0.6)

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

with tf.GradientTape(persistent=True) as tape:
  z1 = w1*x11+w2*x12+b
  p1 = tf.math.sigmoid(z1)
  c1 = (p1-y1)**2

  z2 = w1*x21+w2*x22+b
  p2 = tf.math.sigmoid(z2)
  c2 = (p2-y2)**2

  c=(c1+c2)/2

dcdc1 = tape.gradient(c, c1); print("dcdc1:",dcdc1.numpy())       dcdc1: 0.5
dcdc2 = tape.gradient(c, c2); print("dcdc2:",dcdc2.numpy())       dcdc2: 0.5

dcdp1 = tape.gradient(c, p1); print("dcdp1:",dcdp1.numpy())       dcdp1: 0.08500445
dcdp2 = tape.gradient(c, p2); print("dcdp2:",dcdp2.numpy())       dcdp2: -0.10499986

dp1dz1 = tape.gradient(p1, z1); print("dp1dz1:",dp1dz1.numpy())   dp1dz1: 0.24977514
dp2dz2 = tape.gradient(p2, z2); print("dp2dz2:",dp2dz2.numpy())   dp2dz2: 0.24997501

dz1dw1 = tape.gradient(z1, w1); print("dz1dw1:",dz1dw1.numpy())   dz1dw1: 0.5
dz1dw2 = tape.gradient(z1, w2); print("dz1dw2:",dz1dw2.numpy())   dz1dw2: -0.3
dz1db = tape.gradient(z1, b); print("dz1db:",dz1db.numpy())      dz1db: 1.0

dz2dw1 = tape.gradient(z2, w1); print("dz2dw1:",dz2dw1.numpy())   dz2dw1: 0.2
dz2dw2 = tape.gradient(z2, w2); print("dz2dw2:",dz2dw2.numpy())   dz2dw2: -0.4
dz2db = tape.gradient(z2, b); print("dz2db:",dz2db.numpy())      dz2db: 1.0

dcdz1 = tape.gradient(c, z1); print("dcdz1:",dcdz1.numpy())      dcdz1: 0.021232
dcdz2 = tape.gradient(c, z2); print("dcdz2:",dcdz2.numpy())      dcdz2: -0.026247343

dcdw1 = tape.gradient(c, w1); print("dcdw1:",dcdw1.numpy())      dcdw1: 0.005366531
dcdw2 = tape.gradient(c, w2); print("dcdw2:",dcdw2.numpy())      dcdw2: 0.004129337
dcdb  = tape.gradient(c, b);  print("dcdb: ",dcdb.numpy())       dcdb:  -0.0050153434
```

Programa 4: Perceptron1b.py

Nota: A solução está em ~/deep/keras/autodiff/perceptron2.py e perceptron2.m (Octave)

Agora, vamos verificar como as contas foram feitas. Feed-forward fica:

```
sigmoid = @(z) 1 ./ (1 + exp(-z));

x11=0.5
x12=-0.3
y1=0.4
x21=0.2
x22=-0.4
y2=0.6
w1=-0.2
w2=0.2
b=0.1

z1 = w1*x11 + w2*x12 + b
p1=sigmoid(z1)
c1 = (p1-y1)^2

z2 = w1*x21 + w2*x22 + b
p2=sigmoid(z2)
c2 = (p2-y2)^2

c=(c1+c2)/2
```

```
x11 =  0.50000
x12 = -0.30000
y1 =  0.40000
x21 =  0.20000
x22 = -0.40000
y2 =  0.60000
w1 = -0.20000
w2 =  0.20000
b =  0.10000

z1 = -0.060000
p1 =  0.48500
c1 =  0.0072258

z2 = -0.020000
p2 =  0.49500
c2 =  0.011025

c =  0.0091254
```

Programa 5: Perceptron2.m em Octave.

Vamos calcular e armazenar as derivadas parciais:

$z_1 = x_{11}w_1+x_{12}w_2+b \rightarrow$
$dz_1/dw_1 = x_{11} = 0.5$
$dz_1/dw_2 = x_{12} = -0.3$
$dz_1/db = 1$
$p1 = \sigma(z1) \rightarrow dp1/dz1 = \sigma(z1)(1-\sigma(z1)) \rightarrow dp1/dz1 = 0.24978$

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

$c1 = (p1-y1)^2 = (p1-0.4)^2 = p1^2 - 0.8\ p1 + 0.16 \rightarrow dc1/dp1 = 2*p1-0.8$
$\rightarrow dc/dp1 = (dc/dc1) * (dc1/dp1) = p1-0.4$
$\rightarrow dc/dp1 = 0.485-0.4 = 0.085$

$z_2 = x_{21}w_1+x_{22}w_2+b \rightarrow$
$dz_2/dw_1 = x_{21} = 0.2$
$dz_2/dw_2 = x_{22} = -0.4$
$dz_2/db = 1$
$p2 = \sigma(z2) \rightarrow dp2/dz2 = \sigma(z2)(1-\sigma(z2)) \rightarrow dp2/dz2 = 0.24998$
$c2 = (p2-y2)^2 = (p2-0.6)^2 = p2^2 - 1.2\ p2 + 0.36 \rightarrow dc2/dp2 = 2*p2-1.2$
$\rightarrow dc/dp2 = (dc/dc2) * (dc2/dp2) = p2-0.6$
$\rightarrow dc/dp2 = 0.495-0.6 = -0.105$

$c = c1/2+c2/2 \rightarrow dc/dc1 = dc/dc2 = 0.5$

Vamos fazer backpropagation. Para calcular as derivadas parciais dc/dw1, dc/dw2 e dc/db, é necessário somar as derivadas parciais das duas amostras de treino:

```
dc/dz1 = (dc/dp1)*(dp1/dz1) = 0.085 * 0.24978 = 0.021231
dc/dz2 = (dc/dp2)*(dp2/dz2) = -0.105 * 0.24998 = -0.026248

dc/dw1 = (dc/dz1)*(dz1/dw1) + (dc/dz2)*(dz2/dw1) =
         0.021231 * 0.5 - 0.026248 * 0.2 = 0.010616 – 0.0052496 = 0.0053664
dc/dw2 = (dc/dz1)*(dz1/dw2) + (dc/dz2)*(dz2/dw2) =
         0.021231 * (-0.3) + (-0.026248) * (-0.4) = 0.0041299
dc/db = (dc/dz1)*(dz1/db) + (dc/dz2)*(dz2/db) =
         0.021231 * 1 + (-0.026248) * 1 = -0.0050170
```

Este processo está resumido em azul no programa 6.

`sigmoid = @(z) 1 ./ (1 + exp(-z));`	Feed forward	GradientTape
x11=0.5	x11 = 0.50000	
x12=-0.3	x12 = -0.30000	
y1=0.4	y1 = 0.40000	
x21=0.2	x21 = 0.20000	
x22=-0.4	x22 = -0.40000	
y2=0.6	y2 = 0.60000	
w1=-0.2	w1 = -0.20000	
w2=0.2	w2 = 0.20000	
b=0.1	b = 0.10000	
z1 = w1*x11 + w2*x12 + b	z1 = -0.060000	dz1/dw1 = x11 = 0.5 dz1/dw2 = x12 = -0.3 dz1/db = 1
p1=σ(z1)	p1 = 0.48500	dp1/dz1 = σ(z1)(1-σ(z1)) = 0.24978
c1 = (p1-y1)^2	c1 = 0.0072258	dc1/dp1 = 2*p1-0.8 = 0.170
z2 = w1*x21 + w2*x22 + b	z2 = -0.020000	dz2/dw1 = x21 = 0.2 dz2/dw2 = x22 = -0.4 dz2/db = 1
p2=σ(z2)	p2 = 0.49500	dp2/dz2 = σ(z2)(1-σ(z2)) = 0.24998
c2 = (p2-y2)^2	c2 = 0.011025	dc2/dp2=2*p2-1.2=−0.210
c=(c1+c2)/2	c = 0.0091254	dc/dc1 = dc/dc2 = 0.5

```
Backpropagation

dc/dw1 = (dc/dc1)*(dc1/dp1)*(dp1/dz1)*(dz1/dw1) + (dc/dc2)*(dc2/dp2)*(dp2/dz2)*(dz2/dw1) =
  0.5*0.170*0.24978*0.5 + 0.5*-0.210*0.24998*0.2 = 0.0053661

dc/dw2 = (dc/dc1)*(dc1/dp1)*(dp1/dz1)*(dz1/dw2) + (dc/dc2)*(dc2/dp2)*(dp2/dz2)*(dz2/dw2) =
  0.5*0.170*0.24978*-0.3 + 0.5*-0.210*0.24998*-0.4 = 0.0041298

dc/db = (dc/dc1)*(dc1/dp1)*(dp1/dz1)*(dz1/db) + (dc/dc2)*(dc2/dp2)*(dp2/dz2)*(dz2/db) =
  0.5*0.170*0.24978*1 + 0.5*-0.210*0.24998*1 = -0.0050166
```

Programa 6: Perceptron2.m os resultados que devem ser gravados em GradientTape.

Exercício: Considere a seguinte rede neural:

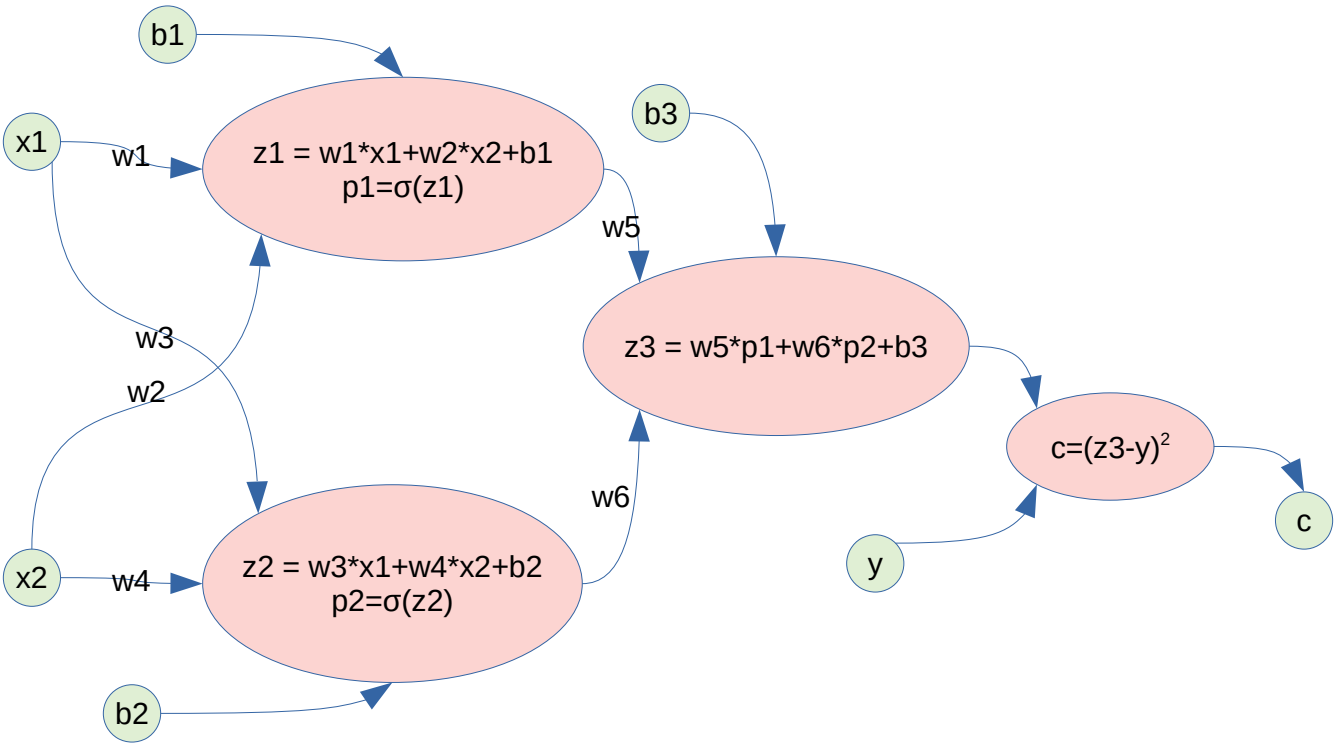

Figura 7

Escreva um programa TensorFlow que calcula as derivadas parciais $\partial c/\partial w_1$, $\partial c/\partial w_2$, $\partial c/\partial w_3$, $\partial c/\partial w_4$, $\partial c/\partial w_5$, $\partial c/\partial w_6$, $\partial c/\partial b_1$, $\partial c/\partial b_2$ e $\partial c/\partial b_3$ 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
x_1=0.6, x_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.

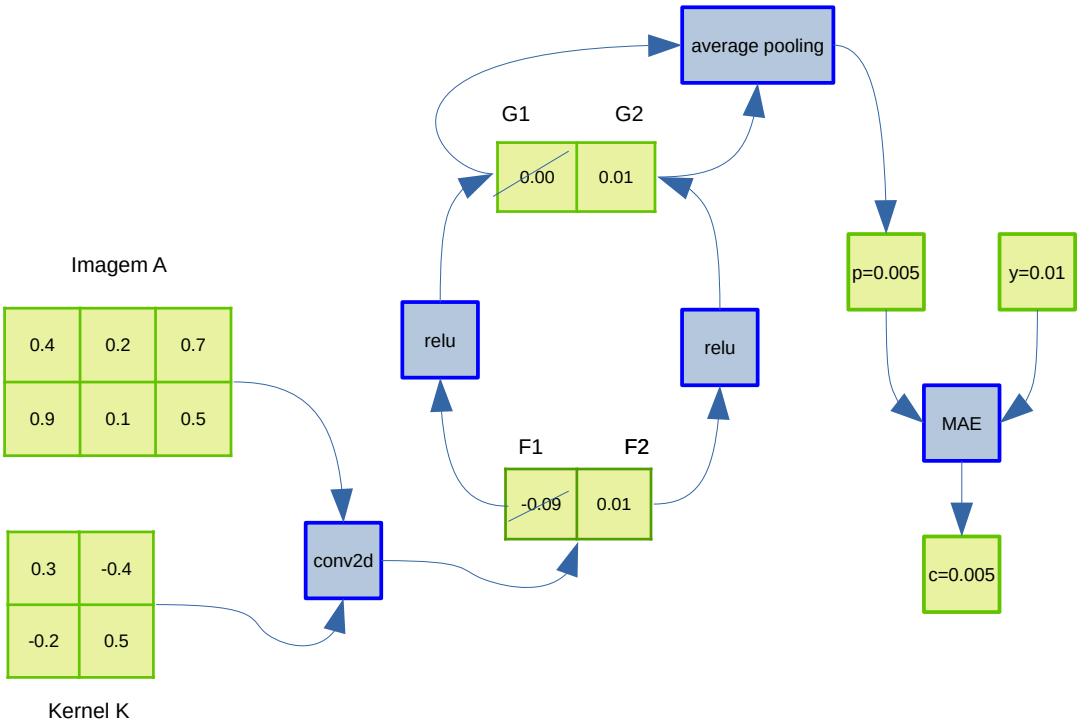

Figura 8: Autodiff numa rede convolucional com ativação relu.

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 = conv2d(A) = [-0.09, 0.01]
G = relu(F) = [0.00, 0.01]
p = 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 = conv2d(A, 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:

$p = \text{avePool}(G) = (G1+G2)/2 \rightarrow dp/dG_1 = 0.5$; $dp/dG2 = 0.5$.

$G_1 = \text{relu}(F_1) \rightarrow dG_1/dF_1 = 0$ 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:

$dc/dK = (dc/dp)\times(dp/dG)\times(dG/dF)\times(dF/dK) =$
$(dc/dp)\times(dp/dG_1)\times(dG_1/dF_1)\times(dF_1/dK) + (dc/dp)\times(dp/dG_2)\times(dG_2/dF_2)\times(dF_2/dK) =$
$(-1)\times0.5\times0\times[0.4, 0.2; 0.9, 0.1] + (-1)\times0.5\times1\times[0.2, 0.7; 0.1, 0.5] =$
$(-0.5)\times[0.2, 0.7; 0.1, 0.5] =$ [-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.]
dpdG: [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.

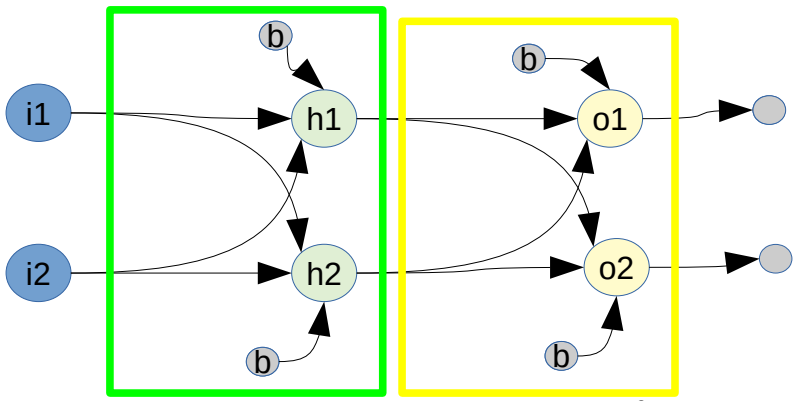

Figura: Estrutura da rede neural do programa from1.py

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=
[[0.09999999 0.9       ]
 [0.90000004 0.10000002]
 [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
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[1,0]+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       ]
 [0.90000004 0.10000002]
 [0.11033662 0.89271045]
 [0.8298143  0.16628951]]
```

4) Para debugar o que acontece dentro do custom layer, é necessário compilar usando opção *run_eagerly=True*. Se não colocar esse comando, o *print* dentro do custom layer não é impresso. TensorFlow funciona desta forma para garantir a velocidade computacional. O programa rodará muito mais devagar com a opção *run_eagerly=True*.
https://keras.io/examples/keras_recipes/debugging_tips/

```
# from4.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; import sys

#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):
    if input_shape[1]!=2: sys.exit("Erro: Dimensao de entrada deve ser 2")
    self.W00 = tf.Variable( 0.3, name="W00"); self.W01 = tf.Variable(-0.8, name="W01");
    self.W10 = tf.Variable(-0.4, name="W10"); self.W11 = tf.Variable( 0.2, name="W11");
    self.b0  = tf.Variable( 0.0, name="b0");  self.b1  = tf.Variable( 0.0, name="b1");

  def call(self, inputs):
    # A dimensao 0 (indicada por ":") e' para permitir rodar batches.
    print("inputs=",inputs)
    z0 = inputs[:,0]*self.W00+inputs[:,1]*self.W10+self.b0;
    z1 = inputs[:,0]*self.W01+inputs[:,1]*self.W11+self.b1;
    z = tf.stack([z0,z1], axis=1, name="z")
    print("  z=",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", run_eagerly=True)

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
print("<<<<<<<<<< Treino <<<<<<<<<<<<<<<<")
model.fit(AX, AY, epochs=120, batch_size=1, shuffle=False, verbose=0)

print("<<<<<<<<<< Teste <<<<<<<<<<<<<<<<")
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)
```

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.]