[PSI3472 aula 7, Início]

Diferenciação automática

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 um dos parâmetros (pesos e vieses), para que possamos modificá-los para diminuir função custo executando descida do gradiente, como nas equações da figura 1.

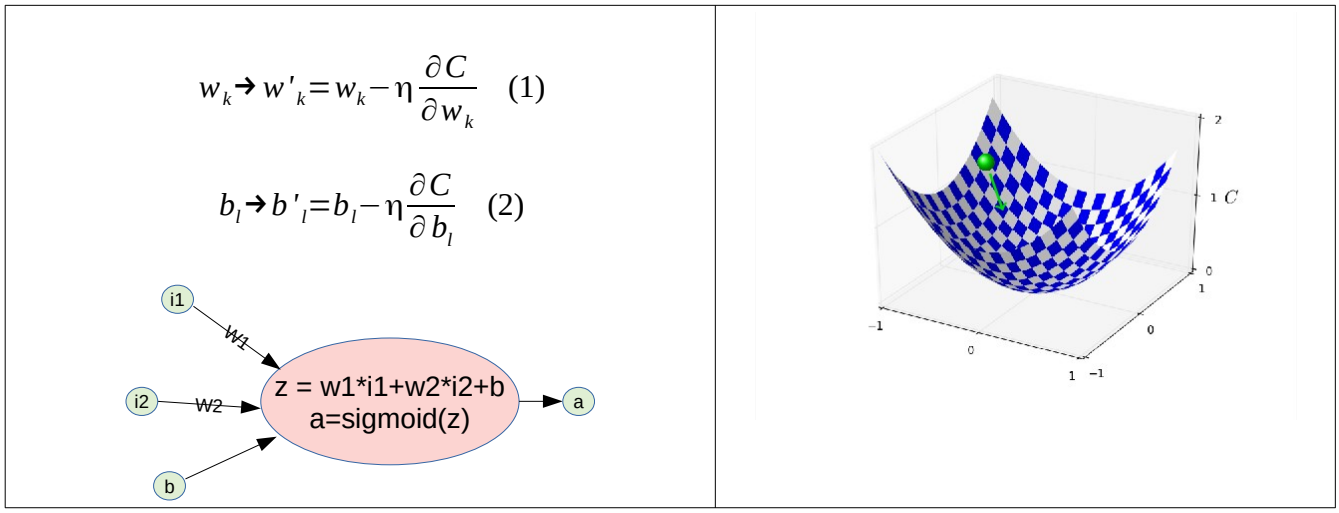


Figura 1: Descida do gradiente

Porém, até agora não vimos como calcular as derivadas parciais. 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 podem haver operações complexas como convoluções, camadas recorrentes e camadas de atenção. Parece que TensorFlow faz alguma mágica... É importante saber como as derivadas parciais são calculadas, pois é fundamental para entender o funcionamento das redes neurais.

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 $f(g(x))$.

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

2) Exemplo de regra da cadeia:
https://en.wikipedia.org/wiki/Chain_rule

Vamos calcular a derivada da função:

$$y = e^{\sin(x^2)} \qquad (4)$$

Esta função pode ser decomposta como a composição de 3 funções tal que $y = f(g(h(x)))$:

$y = f(u) = exp(u)$ $u = g(v) = sin(v)$ $v = h(x) = x^2$	(5)

Calculando as derivadas:

$dy/du = exp(u)$ $du/dv = cos(v)$ $dv/dx = 2x$	(6)

Aplicando a regra da cadeia, obtemos a expressão algébrica para a derivada de y, em função de variáveis intermediárias u e v.

$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx} = e^u \cdot \cos(v) \cdot 2x$	(7)

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

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

À medida que a função y torna-se mais complexa, a expressão algébrica da derivada (8) fica cada vez mais longa e complexa. 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, como no caso de execução condicional.

II. GradientTape

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

Antes de prosseguirmos, vamos verificar que TensorFlow consegue calcular facilmente as derivadas *dy/du*, *dy/dv* e *dy/dx* da função $y = e^{\sin(x^2)}$, usando uma API chamada *GradientTape* num ponto *x* dado. Digamos que queiramos calcular as derivadas para *x*=2. O programa 1 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)
  y = tf.exp(u)

dydu = tape.gradient(y, u); print("dydu:",dydu.numpy())
dudv = tape.gradient(u, v); print("dudv:",dudv.numpy())
dvdx = tape.gradient(v, x); print("dvdx:",dvdx.numpy())
dydv = tape.gradient(y, v); print("dydv:",dydv.numpy())
dydx = tape.gradient(y, x); print("dydx:",dydx.numpy())
dudx = tape.gradient(u, x); print("dudx:",dudx.numpy())
del tape
```

```
dydu: 0.4691642
dudv: -0.65364367
dvdx: 4.0
dydv: -0.3066662
dydx: -1.2266648
dudx: -2.6145747
```

Programa 1: Autodiff2.py (~/deep/keras/autodiff/autodiff2.py)

Parece mágica! Note que:

1. TensorFlow não calcula as expressões algébricas completas das derivadas, como a equação 8. TensorFlow calcula as derivadas numéricas apenas no ponto desejado (*x=2*).
2. O programa 1 utiliza as funções próprias do TensorFlow (*tf.pow*, *tf.sin*, *tf.exp*) para calcular a função *y*. Isto parece sugerir que as funções numéricas do TensorFlow fazem algo a mais do que simplesmente calcular x^y, *seno* e e^x.

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 *dy/du*, *dy/dv* e *dy/dx*? Há três possibilidades.

Método 1) A primeira é achar uma expressão algébrica completa para cada uma das derivadas. Se quiser calcular *(dy/dx)(x=2)* por este método, substituimos x por 2 na equação (8), obtendo:

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

O problema desta abordagem é que as expressões podem ficar muito complexas (pense como ficaria a derivada numa camada convolucional em relação aos elementos do kernel ou de uma execução condicional ou dentro de um loop).

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

$$\frac{dy(x)}{dx} \cong \frac{y(x+\varepsilon)-y(x-\varepsilon)}{2\,\varepsilon}$$

Se quiser calcular *(dy/dx)(x=2)* por este método:

```
>> x=2
>> epsilon=1e-3
>> x1=x-epsilon
>> x2=x+epsilon
>> y1=exp(sin(x1^2))
>> y2=exp(sin(x2^2))
>> dydx=(y2-y1)/(2*epsilon)
dydx = -1.2267
```

O problema desta abordagem é a imprecisão numérica, principalmente quando a expressão é longa. Além disso, precisamos "chutar" um valor adequado para ε.

Método 3) A diferenciação automática é a terceira opção. Para cada função componente da $y(x)$, calcula-se a sua derivada. Na figura 2, os retângulos azuis calculam $y(x)$ passo a passo. Os retângulos vermelhos são as derivadas de cada um desses passos. $y = e^{\sin(x^2)}$

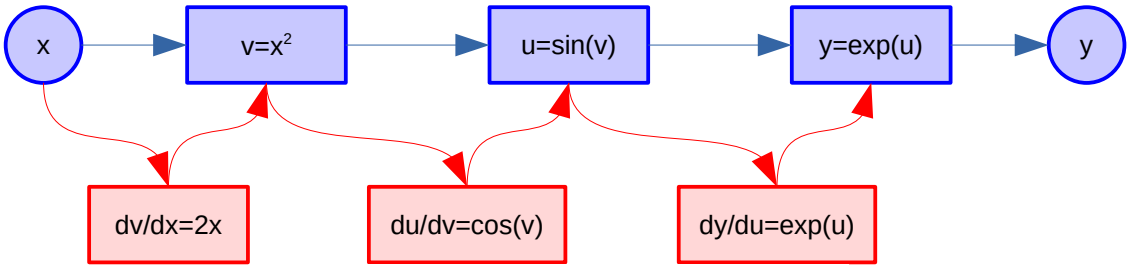


Figura 2: Autodiff de uma função composta.

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). Desta forma, não é necessário calcular a derivada da função completa *dy/dx*, mas apenas as derivadas das funções constituintes. Ou, melhor ainda, apenas os valores numéricos das derivadas constituintes no ponto x considerado.

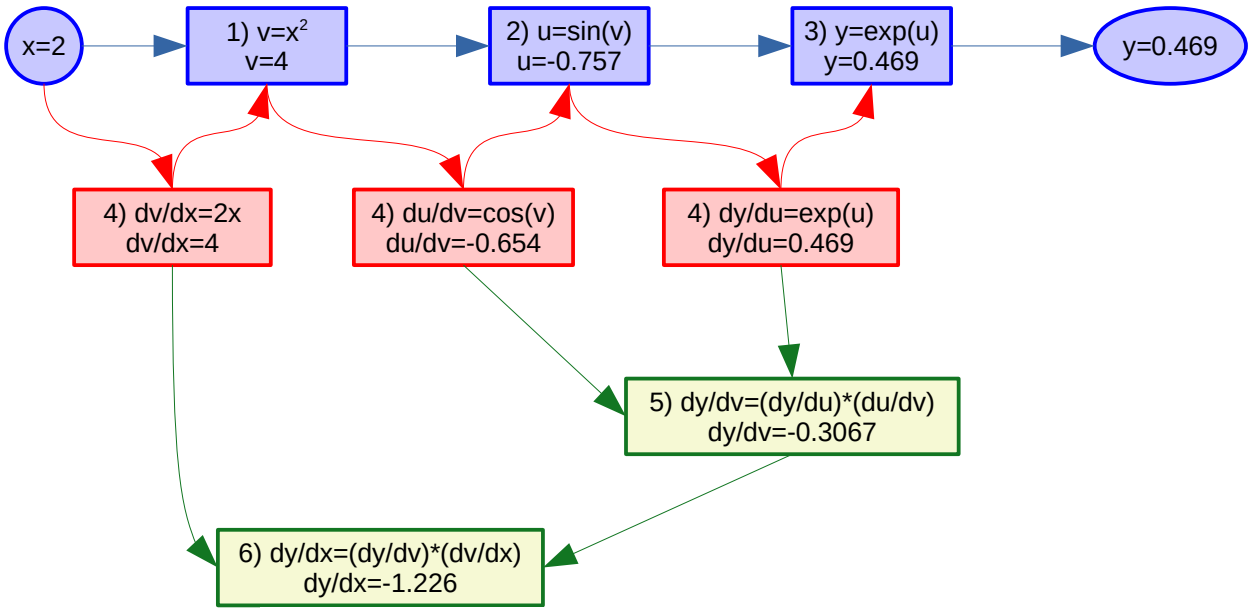


Figura 3: Autodiff com feed-forward e back-propagation.

O processo de calcular derivada parcial consiste de dois passos:

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

Os valores calculados na figura 3 coincidem com os resultados obtidos pelo programa 1 (autodiff2.py) e pelos métodos 1 e 2. Assim, é possível calcular as derivadas parciais se gravar a sequência de operações efetuadas durante feed-forward.

Nota: Aqui, vemos que nem é necessário calcular as expressões completas das derivadas das funções constituintes *dy/du*, *du/dv* e *dv/dx*, mas apenas os valores numéricos das derivadas no ponto considerado:

$$\left.\frac{dy}{du}\right|_{u=-0.757}, \quad \left.\frac{du}{dv}\right|_{v=4}, \quad \left.\frac{dv}{dx}\right|_{x=2}$$

Assim, para a função *relu*(*x*), por exemplo, é possível especificar a derivada para os três casos:

$$relu'(x)=\begin{cases}1, se\ x>0\\0, se\ x<0\\indefinido, se\ x=0\end{cases}$$

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) de um neurônio com um único exemplo de treino (x_1, x_2, y) – figura 4.

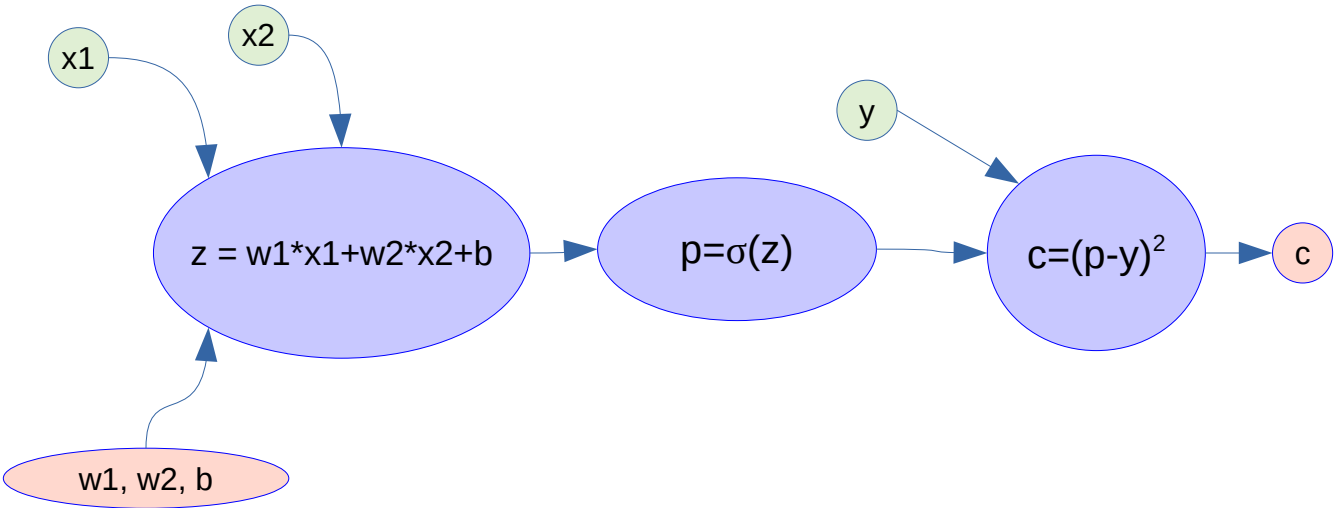


Figura 4: Um neurônio com função custo *MSE*. Ativação sigmoide é representada como σ.

Vamos considerar o exemplo de treino (x_1=0.5, x_2=-0.3, y=0.4) e os parâmetros iniciais (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 $\partial c/\partial b$.

Vamos fazer primeiro o feed-forward (elipses azuis 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 numéricas 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-se backpropagation (retângulos verdes da figura 5):

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

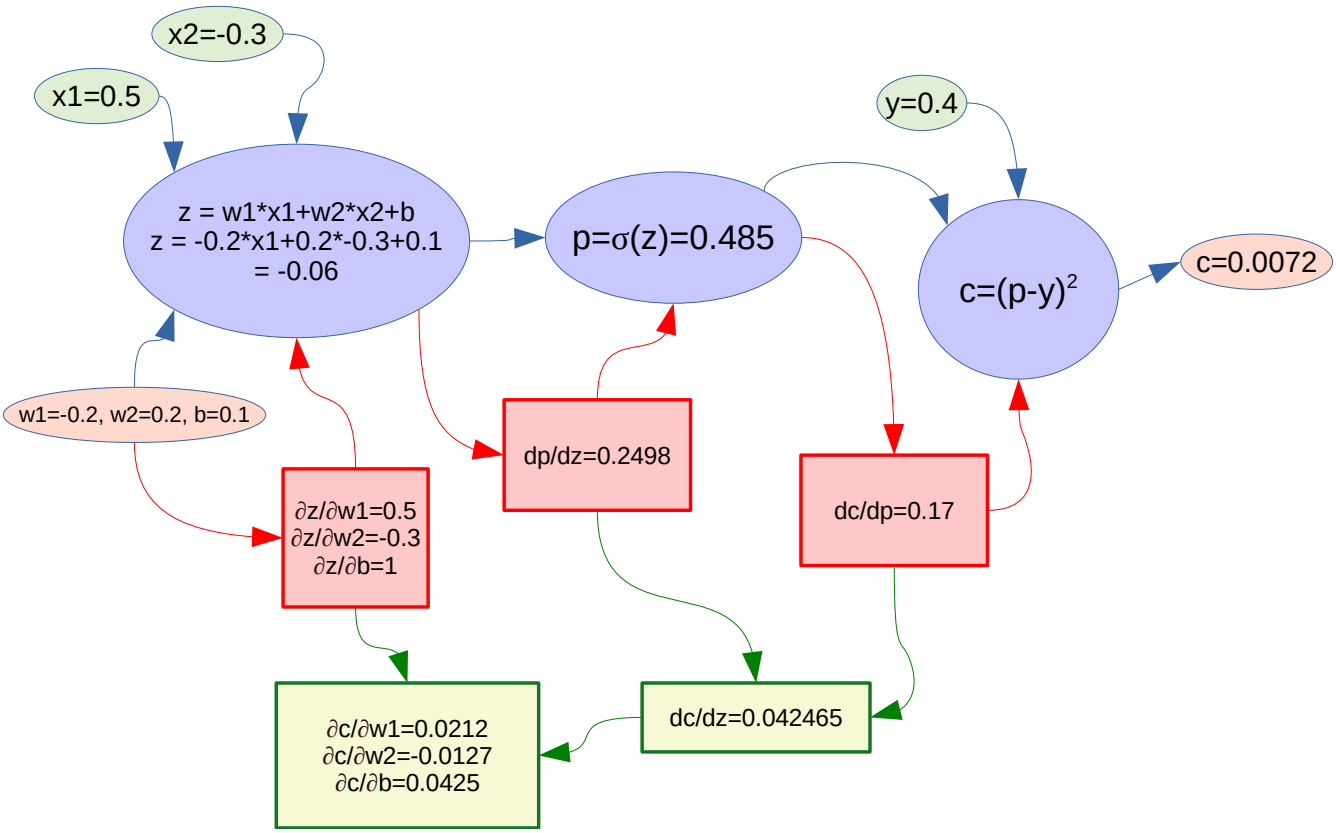


Figura 5: Feedforward e backpropagation do neurônio da figura 4 com um único exemplo de treino.

Podemos comparar os resultados acima com GradientTape do TensorFlow no programa 2. É possível usar variáveis escalares (programa 2a) ou tensores (programa 2b). Todos dão os mesmos resultados.

Nota: Existem variáveis *tf.Variable* e *tf.constant*. Só é possível calcular as derivadas em relação às *tf.Variable*.

```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: variáveis 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: tensores.

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

[PSI3472 aula 7/8, lição de casa #1/2. Vale 5.] Considere a rede neural da figura *F* e os valores numéricos x=0.5, y=0.4, w=-0.2 e b=0.7. Escreva um programa Python que calcula os valores numéricos de $\partial c/\partial w$ e $\partial c/\partial b$ sem usar TensorFlow ou qualquer outra biblioteca que faça diferenciação automática.

Nota: O exercício está simples demais.

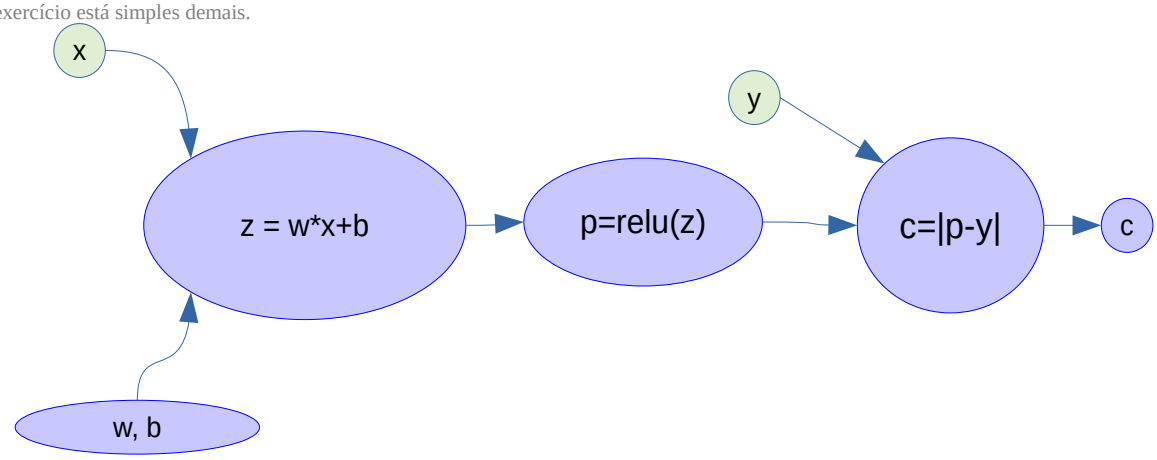


Figura *F*: Calcule $\partial c/\partial w$ e $\partial c/\partial b$.

[PSI3472 aula 7/8, lição de casa #2/2. Vale 5.] Resolva o exercício anterior escrevendo um programa Tensorflow e GradientTape.

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

Considere a mesma rede da figura 4, agora alimentada com um lote com 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.

Vamos supor que 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, devemos calcular 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 enxergarmos 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:

$\partial c/\partial w_1 = (\partial c/\partial z_1)\times(\partial z_1/\partial w_1)/2 + (\partial c/\partial z_2)\times(\partial z_2/\partial w_1)/2$
$\partial c/\partial w_2 = (\partial c/\partial z_1)\times(\partial z_1/\partial w_2)/2 + (\partial c/\partial z_2)\times(\partial z_2/\partial w_2)/2$
$\partial c/\partial b = (\partial c/\partial z_1)\times(\partial z_1/\partial b)/2 + (\partial c/\partial z_2)\times(\partial z_2/\partial b)/2$

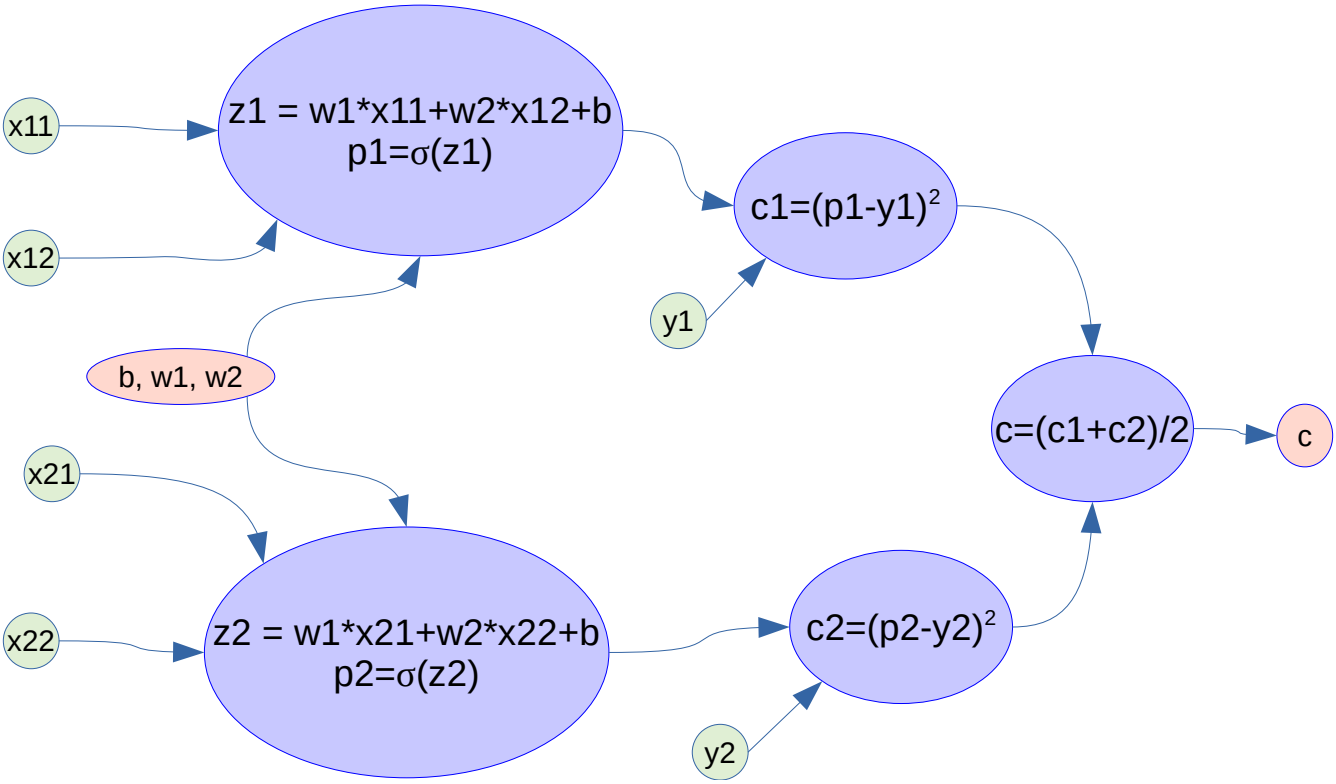


Figura 6: Feedforward e backpropagation do neurônio da figura 4 usando lote com dois exemplos de treino.

O programa 4 calcula as derivadas parciais dos parâmetros para o lote com dois exemplos 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())
dcdc2 = tape.gradient(c, c2); print("dcdc2:",dcdc2.numpy())

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

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

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

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

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

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())
```

```
dcdc1: 0.5
dcdc2: 0.5

dcdp1: 0.08500445
dcdp2: -0.10499986

dp1dz1: 0.24977514
dp2dz2: 0.24997501

dz1dw1: 0.5
dz1dw2: -0.3
dz1db: 1.0

dz2dw1: 0.2
dz2dw2: -0.4
dz2db: 1.0

dcdz1: 0.021232
dcdz2: -0.026247343

dcdw1: 0.005366531
dcdw2: 0.004129337
dcdb:  -0.0050153434
```

Programa 4: Perceptron1b.py

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

Exercício: Considere a rede neural da figura 7:

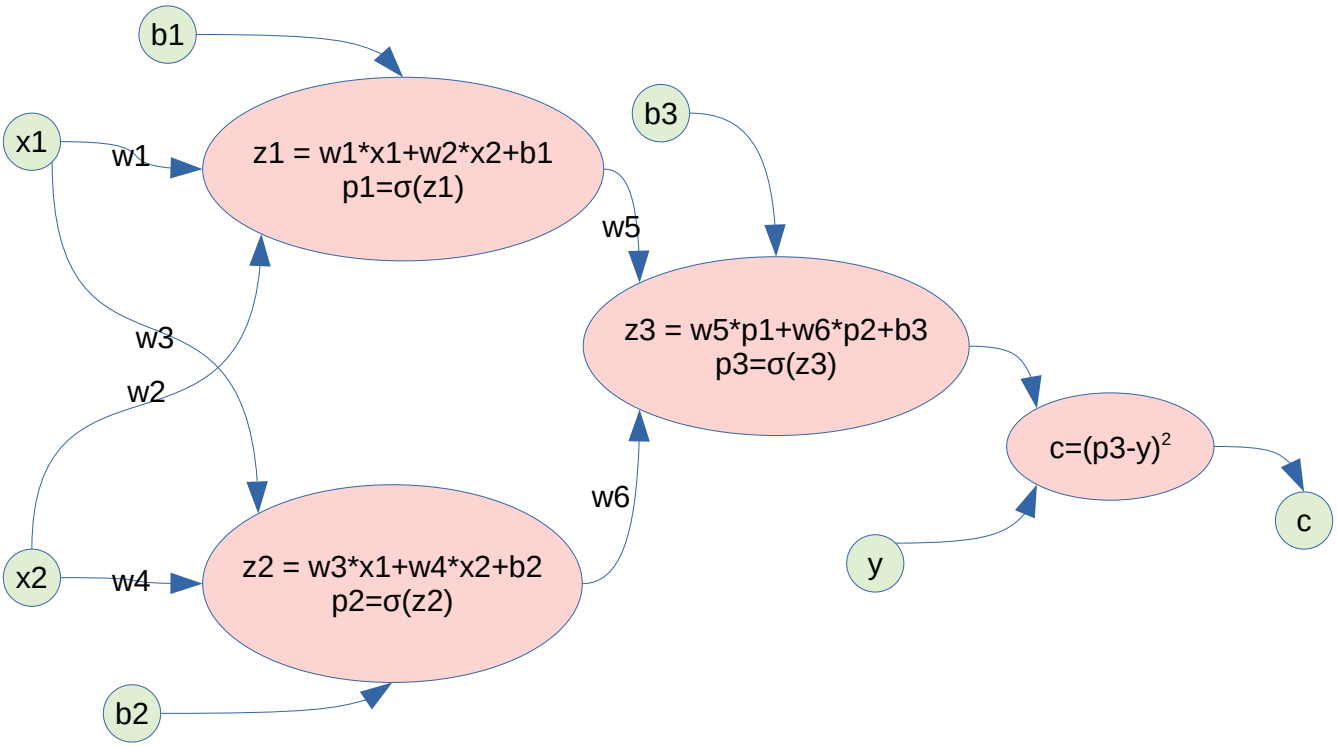


Figura 7

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: -0.008844384
dcdw2: 0.004422192
dcdw3: 0.016614905
dcdw4: -0.008307452
dcdw5: -0.15763849
dcdw6: -0.18567343
dcdb1: -0.014740639
dcdb2: 0.027691506
dcdb3: -0.29606012
```

V. Autodiff em rede neural convolucional com relu e MAE

Agora, vamos ver o funcionamento de autodiff numa pequena rede convolucional com função de ativação relu (figura 8). A rede recebe uma imagem *A* e aplica uma convolução 2D usando kernel *K* no modo "valid", gerando atributos F_1 e F_2. Esses atributos passarão pela ativação *relu*, gerando atributos G_1 e G_2. Um *average-pooling* irá calcular a média entre G_1 e G_2, obtendo a saída final da rede *p*. Como a saída desejada é *y*, obtemos o custo final *c* = | *p*-*y* | usando MAE como função de perda.

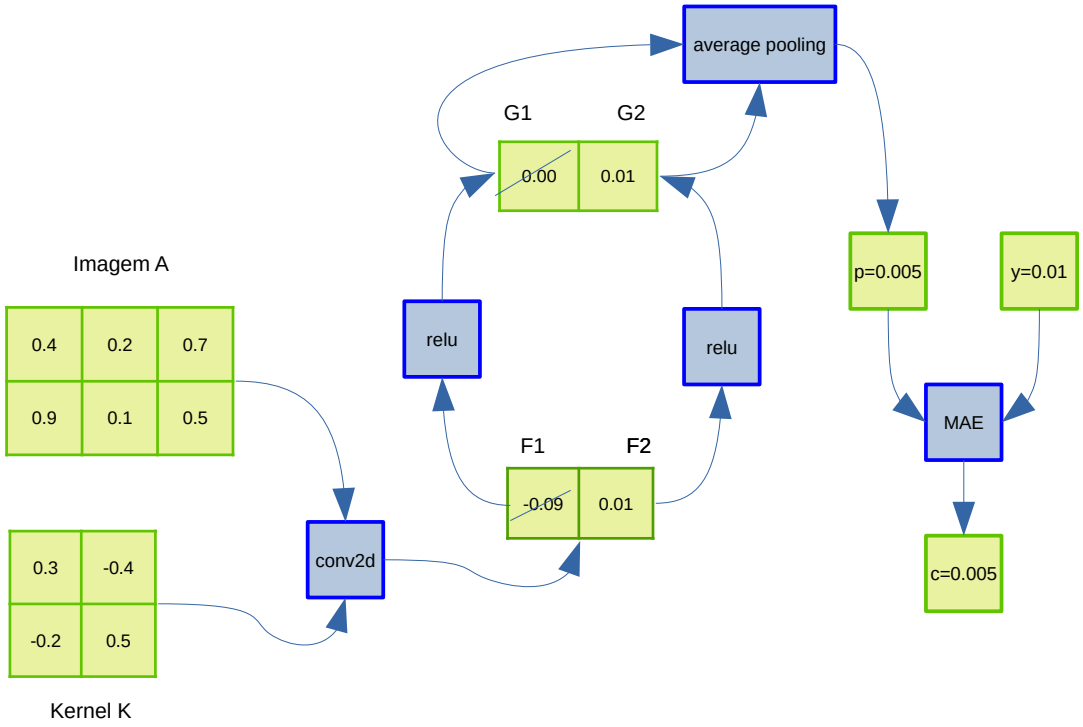


Figura 8: Autodiff numa rede convolucional com ativação relu.

Suponha que queremos treinar a rede convolucional acima, para obter a convolução *K* que minimiza o custo *c*. Para isso, precisamos calcular *dc*/*dK*, isto é, como devemos alterar o filtro *K* para que a rede convolucional execute a tarefa desejada. Vamos fazer feed-forward (os valores de *A* e *K* estão na figura 8):

F = conv2d(*A*, *K*) = [-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 quisermos fazer convolução em Octave ou Matlab obtendo o mesmo resultado que Tensorflow, devemos rotacionar antes o Kernel K por 180 graus.
F = conv2d(A, rot90(K,2)) = [-0.09, 0.01]

Vamos fazer back-propagation.

No cálculo de MAE $c = |p-y|$:
$dc/dp =$ -1 se $p<y$;
+1 se $p>y$;
indefinido se $p=y$.
dc/dp=-1, pois $p<y$.

No cálculo de average pooling:
p = avePool(G) = $(G_1+G_2)/2 \rightarrow dp/dG_1 = 0.5$; $dp/dG_2 = 0.5$.

A derivada da função y=relu(x) depende do valor de x:
$dy/dx =$ 0 se $x<0$;
+1 se $x>0$;
indefinido se $x=0$.
Assim no cálculo de relu:
G_1 = relu(F_1) → $dG_1/dF_1 = 0$ (pois $F_1<0$).
G_2 = relu(F_2) → $dG_2/dF_2 = 1$ (pois $F_2>0$).

Na convolução, vamos calcular as derivadas de F_1 e F_2 em relação à K:
$F_1 = A_{11}K_{11}+A_{12}K_{12}+A_{21}K_{21}+A_{22}K_{22} = 0.4K_{11}+0.2K_{12}+0.9K_{21}+0.1K_{22} \rightarrow dF_1/dK$ = [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 se as nossas contas manuais acima estão corretas calculando dc/dK pelo GradientTape do 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 5: 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 ver que *dc*/*dK* calculado manualmente coincide com *dc*/*dK* calculado pelo TensorFlow.

Exercício: A figura abaixo mostra os diferentes atributos aprendidos pela CNN para reconhecer faces. Vamos tentar fazer algo parecido com um modelo *M* que reconhece os dígitos de MNIST. Gere uma imagem *I*, 28×28, com pixels aleatórios. Depois, escolha um dos 10 neurônios de saída (por exemplo, a do dígito "0" que corresponde a one-hot-encoding AY="100000000"). Faça uma espécie de "descida de gradiente modificado" para alterar a imagem *I* (em vez dos pesos de *M*) de forma que a predição *M*(*I*') da imagem atualizada *I*' esteja mais próxima do one-hot-encoding "100000000" que *M*(*I*). Repetindo "descida de gradiente modificado" muitas vezes, devemos obter uma imagem do dígito "0" típico (high-level feature). É possível fazer isto também com atributos de nível baixo ou médio.

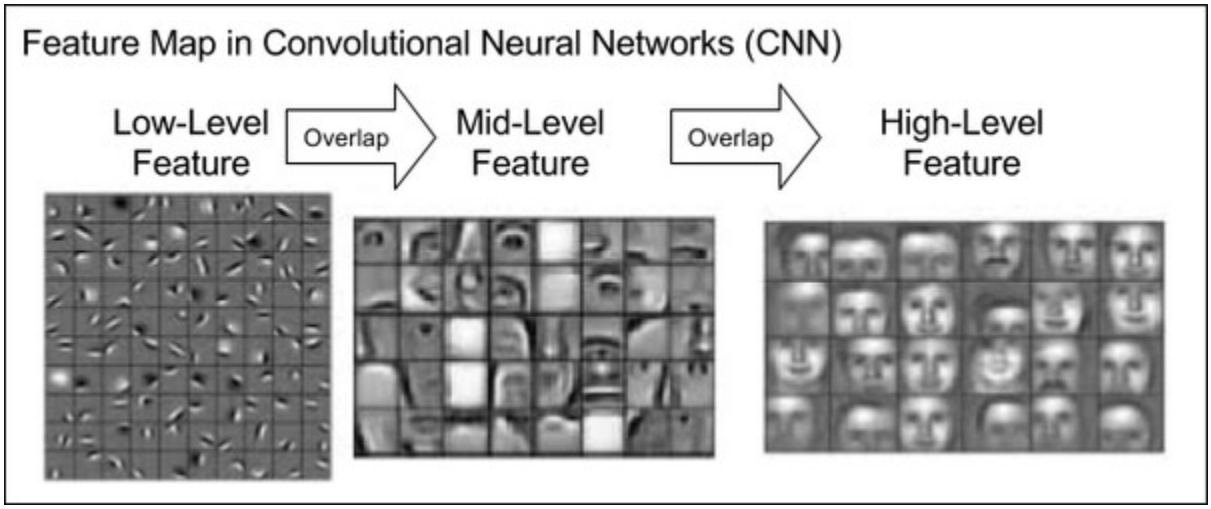


Ilustração que mostra a hierarquia de atributos que permitem uma CNN reconhece a face humana.
https://www.quora.com/How-does-a-convolutional-neural-network-recognize-an-occluded-face

Camada personalizada (custom layer)

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

1) Algumas vezes, pode ser necessário criar camadas diferentes das que estão pré-implementadas em Keras/TensorFlow. Agora que sabemos como TensorFlow calcula as derivadas parciais, estamos prontos para implementar camadas personalizadas: já sabemos que não precisamos nos preocupar em calcular as derivadas parciais, pois *GradientTape* irá cuidar disso.

Keras possui três métodos (APIs) para construir um modelo:

1. Sequencial – o método mais simples para criar modelos.
2. Funcional – o método mais flexível que permite criar modelos mais complexos
3. Criar subclasse (subclassing) – o método mais poderoso mas que requer mais codificação.

Já estudamos os métodos (1) e (2). Para criar uma camada personalizada, precisamos usar o método 3 (subclassing), criando uma classe derivada da classe *Layer* de Keras. Não é necessário trabalhar explicitamente com *GradientTape*, pois 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.

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

```
# ~/deep/keras/densa/fromScratch/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 6: Programa simples que faz regressão.

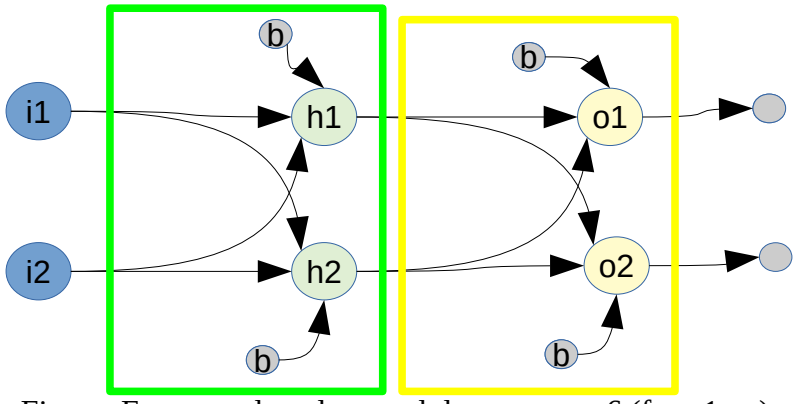


Figura: Estrutura da rede neural do programa 6 (from1.py).

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 ]]
```

Vamos reescrever o programa 6, substituindo a camada “Dense” pela camada customizada “MyDense”.

3) Substituindo camada “Dense” pela camada customizada, obtemos o programa 7. Para que fique claro o que está acontecendo dentro de MyDense, vamos calcular a multiplicação matricial fazendo cálculos variável por variável (em vez de chamar a operação que calcula multiplicação matricial). Vamos fixar o número de entradas e saídas da camada “MyDense” em 2.

A camada Dense inicializa os pesos aleatoriamente com “glorot_uniform”, escolhendo amostras da distribuição uniforme no intervalo [-limit, +limit] onde

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

onde *fanin* e *fanout* são número de entradas e saídas da camada. Para simplificar, vamos “chutar” manualmente os pesos iniciais de MyDense (dentro do intervalo [-1.225, +1.225]) em vez de inicializá-los aleatoriamente. Vieses são inicializados com zeros.

Como pode ver pela saída, o programa continua funcionando corretamente.

Nota: Para poder salvar um modelo que use camada customizada MyDense (model.save(“nome”)), é necessário criar mais métodos.
Veja https://keras.io/guides/serialization_and_saving/

```
# ~/deep/keras/densa/fromScratch/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)
```

Programa 7: Programa que faz regressão usando camada personalizada MyDense.

Saída:

```
Epoch 1/120   2/2 [==============================] - 1s 4ms/step - loss: 1.1747
Epoch 30/120  2/2 [==============================] - 0s 2ms/step - loss: 1.0373e-10
Epoch 60/120  2/2 [==============================] - 0s 3ms/step - loss: 1.2490e-16
Epoch 90/120  2/2 [==============================] - 0s 2ms/step - loss: 1.2490e-16
Epoch 120/120 2/2 [==============================] - 0s 3ms/step - loss: 1.2490e-16
QX=
[[0.9 0.1]
 [0.1 0.9]
 [0.8 0. ]
 [0.2 0.9]]
1/1 [==============================] - 0s 90ms/step
QP=
[[0.1        0.9       ]
 [0.9        0.10000002]
 [0.11852115 0.8831827 ]
 [0.82446176 0.17439479]]
```

Usando multiplicação matricial de Keras, é possível construir camada densa com número arbitrário de entradas e saídas. Programa 8 ilustra isso.

```
# ~/deep/keras/densa/fromScratch/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 8: Camada MyDense com número arbitrário de entradas e saídas.

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]]
```

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.

Exercício: O blog abaixo traz exemplo de ativação antirectifier.
https://keras.io/examples/keras_recipes/antirectifier/
Substitua sigmoide pelo antirectifier no programa 6, execute e verifique o resultado.

4) Para debugar o que acontece dentro do custom layer, é necessário compilar o modelo usando opção *run_eagerly=True*.

Nota: 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/

```
# ~/deep/keras/densa/fromScratch/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.

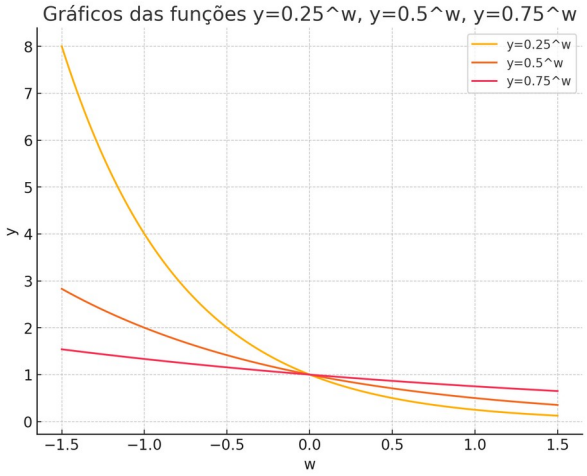
Saídas durante o teste:

```
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)
```

5) Vamos tentar fazer algo diferente de simplesmente re-implementar a camada densa. Vamos criar uma nova camada chamada *Exponenc*, trocando multiplicações pelas exponenciações dentro da camada *Dense*:

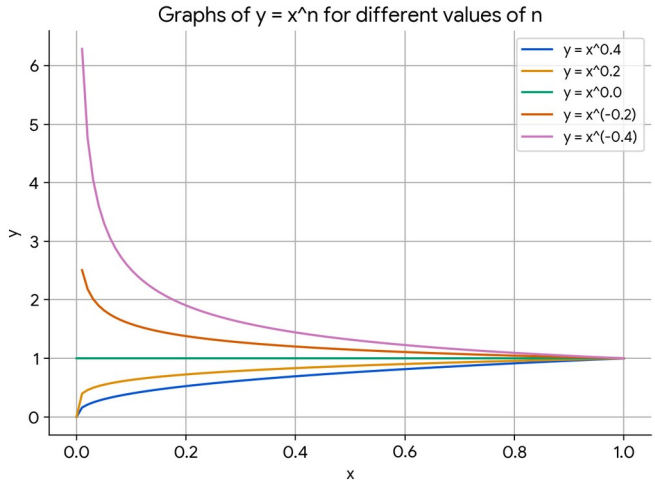
$$z = x_1 \hat{} w_1 + x_2 \hat{} w_2 + b$$

Vamos verificar o comportamento dessa camada. Vamos supor que as entradas dos neurônios estarão no intervalo de 0 a 1 (saída de sigmoide) e que os pesos serão inicializadas pelo "glorot_uniform". O gráfico abaixo mostra $y = x \hat{} w$ para x = 0.25, 0.5 e 0.75. Podemos observar a saída tende para infinito para x próximo de zero e w negativo, pois não existe exponenciação de zero por um número negativo.



Para evitar instabilidade numérica, vamos somar um ε a x: $z = (x_1+\varepsilon) \hat{} w_1 + (x_2+\varepsilon) \hat{} w_2 + b$

Outro gráfico:



O programa abaixo faz regressão usando camada *Exponenc*. Podemos verificar pela saída que a saída da rede está correta, apesar de ter convergido mais lentamente que o programa 7 (from3.py).

```python
# ~/deep/keras/densa/fromScratch/exponenc2.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 Exponenc(tf.keras.layers.Layer):
  #So funciona 2 entradas e 2 saidas
  def __init__(self):
    super(Exponenc, 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.
    # print("inputs=",inputs)
    x=inputs+1e-6
    z0 = x[:,0]**self.W[0,0]+x[:,1]**self.W[1,0]+self.b[0];
    z1 = x[:,0]**self.W[0,1]+x[:,1]**self.W[1,1]+self.b[1];
    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(Exponenc())
model.add(layers.Activation(activations.sigmoid))
model.add(Exponenc())
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
model.fit(AX, AY, epochs=120, batch_size=1, shuffle=False, verbose=1)

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=1)
print("QP="); print(QP)
```

Saída:

```
Epoch 1/120   2/2 [==============================] - 1s 5ms/step - loss: 1.7045
Epoch 30/120  2/2 [==============================] - 0s 2ms/step - loss: 1.1864e-04
Epoch 60/120  2/2 [==============================] - 0s 2ms/step - loss: 5.7809e-10
Epoch 90/120  2/2 [==============================] - 0s 3ms/step - loss: 2.6007e-14
Epoch 120/120 2/2 [==============================] - 0s 2ms/step - loss: 2.4980e-16
QX=
[[0.9 0.1]
 [0.1 0.9]
 [0.8 0. ]
 [0.2 0.9]]
1/1 [==============================] - 0s 97ms/step
QP=
[[0.10000002 0.9       ]
 [0.9        0.10000002]
 [0.06504607 0.9557694 ]
 [0.8474424  0.1339904 ]]
```

[PSI3472 aula 7/8, lição de casa extra 1. Vale +10.] O artigo
https://arxiv.org/pdf/1909.01532
define camada convolucional morfológica como:

> Morphological dilation and erosion are approximated using counter-harmonic mean in [10]. For a grayscale image $f(x)$ and a kernel $\omega(x)$, the core is to define a PConv layer as:
>
> $$PConv(f;\omega,P)(x) = \frac{(f^{P+1}*\omega)(x)}{(f^{P}*\omega)(x)} = (f *_P \omega)(x) \quad (1)$$
>
> where "$*$" denotes the convolution, and P is a scalar controlling the choice of operation ($P < 0$ is pseudo-erosion, $P > 0$ is pseudo-dilation, and $P = 0$ is standard convolution). Since

Implemente essa camada e use-a para classificar MNIST. Qual foi a taxa de erro obtida? O tempo de processamento piorou muito?
Nota: Não pode usar alguma biblioteca pronta de convolução morfológica.

[PSI3472 aula 7/8, lição de casa extra 2. Vale +10.] O artigo
https://arxiv.org/abs/1412.6071
propõe fractional max-pooling. Implemente-a e aplique-a para classificar MNIST, procurando minimizar a taxa de erro. Qual foi a taxa de erro obtida? O tempo de processamento piorou muito?
Nota: Tensorflow possui uma camada pronta max-pooling fracionária. Não pode usar essa camada pronta.

[PSI3472. Aula 7. Fim.]