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.

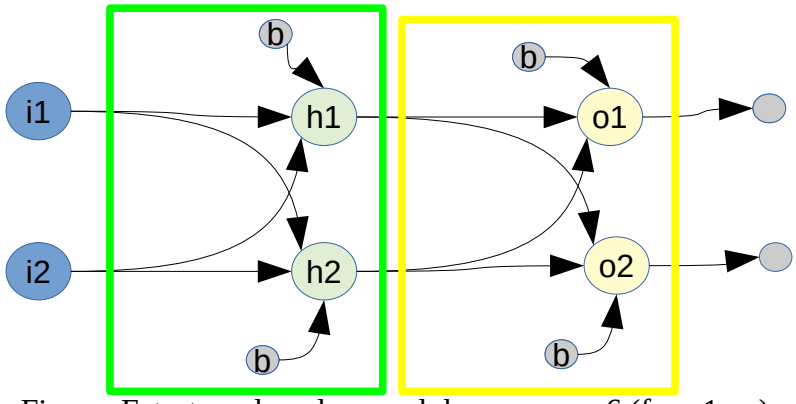


Figura: Estrutura da rede neural do programa 6 (from1.py).

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

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

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.

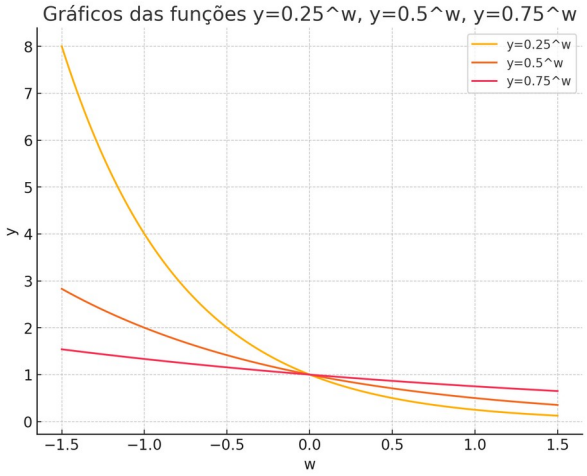
Saídas durante o teste:

```
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*:

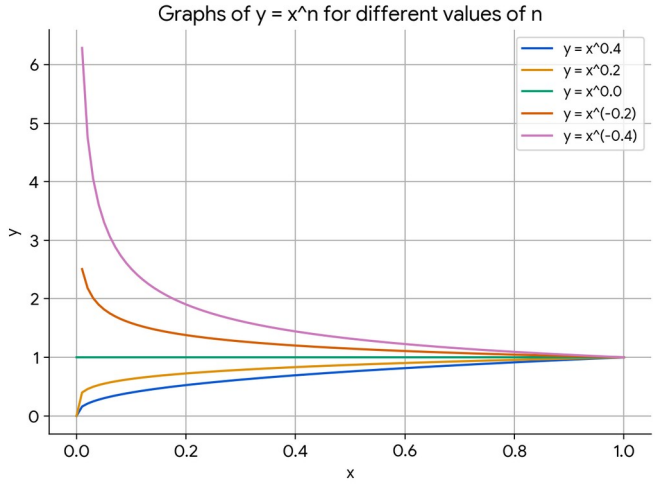
$$z = x_1 \hat{} w_1 + x_2 \hat{} w_2 + b$$

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 ]]
```

Exercício: 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.

Exercício extra: 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.

[PSI3471 aula 13 parte 2. Fim]