[PSI3471 aula 11 parte 2. Início]
PSI3471: Vocês já devem ter visto a maioria dos tópicos desta apostila com prof. Magno. Assim, vou passar rapidamente por esta apostila.

Rede neural densa em Keras sobre TensorFlow ou PyTorch

1. Keras, Tensorflow, PyTorch e Colab

A partir desta aula, vamos trabalhar com aprendizado profundo, usando redes neurais convolucionais. Trabalharemos em linguagem Python, usando bibliotecas Keras sobre TensorFlow ou PyTorch, juntamente com OpenCV. Continuaremos estudando estes tópicos na disciplina PSI3472.

Nota: Alguns programas foram traduzidos para PyTorch e estão nos anexos das versões antigas desta apostila.

Keras é uma biblioteca de código aberto escrita em Python para construção e treinamento de modelos de aprendizado profundo, projetada para ser "amigável e de alto nível", lançada em 2015. Permite escrever os programas de forma simples e rápida, porém é difícil escrever um programa que "foge do padrão".

Originalmente, Keras foi desenvolvida para ser usada em cima do TensorFlow, mas com o tempo, ganhou suporte para outras *backends*, como o PyTorch, através de projetos como keras-pytorch. Neste curso, vamos usar Keras com *backend* TensorFlow.

TensorFlow é uma biblioteca de código aberto para aprendizado de máquina, lançado em 2015. Foi desenvolvida pela equipe Google Brain e é usada tanto para a pesquisa quanto produção na Google. Escrever aplicações de aprendizado de máquina diretamente em TensorFlow costuma ser uma tarefa árdua.

Nota: O que é um tensor (de TensorFlow)? Em aprendizado de máquina, tensor é a generalização dos conceitos de vetor e matriz. É um arranjo multi-dimensional ou uma matriz de mais de duas dimensões. Em matemática, tensor possui outra definição. Porém, para nós, um tensor é simplesmente uma matriz multi-dimensional.

A forma específica de instalar depende do hardware do seu computador e da opção de instalação escolhida (se vai usar ou não GPU; do modelo do GPU; se vai instalar em ambiente virtual ou não; se vai usar Anaconda ou não, etc). Instale como você achar melhor.
Nota: Deixei no site abaixo as instruções de instalação, mas devem estar desatualizadas:
http://www.lps.usp.br/hae/software/instalacao_tensorflow.pdf

Keras em TensorFlow/PyTorch roda consideravelmente mais rápido em GPU do que CPU (algo como 3-10 vezes mais rápido). Por outro lado, dá muito mais trabalho instalá-los para usar GPU do que CPU.

Nota: A dificuldade para instalar TensorFlow para GPU é causado pela necessidade de instalar software de versões compatíveis: sistema Linux, Python, driver de nvidia, CUDA toolkit, cuDNN, Tensorflow, PyTorch. TensorFlow para GPU não funciona se instalar as últimas versões de todos os softwares.

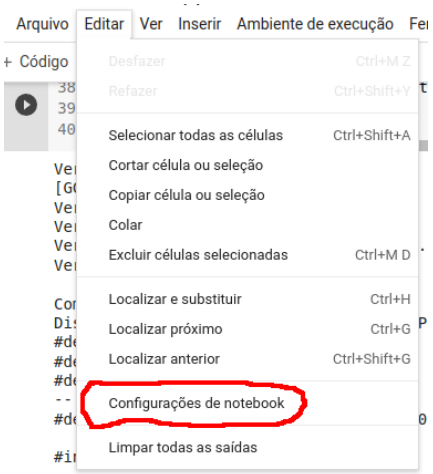
[PSI3471 aula 11. Pular a partir daqui. Devem ter usado Colab com prof. Magno]

1.1 Google Colab

a) Uma outra forma de usar Tensorflow/Keras com GPU, sem precisar instalá-las no computador local, é através do Google Colab. Você pode entrar no Colab via browser (Chrome, Firefox, Brave, etc.) com sua conta Google, dar copy-paste dos programas desta apostila e tudo executa corretamente (talvez com algumas pequenas adaptações). Inclusive, dá para usar GPU/TPU bom só editando as configurações em "notebook settings". Você praticamente não precisa instalar nada no Google Colab – quase todas as bibliotecas estão instaladas de forma compatíveis e funcionando.

GPU (Graphics Processing Unit): Originalmente projetada para renderização gráfica (como em jogos), a GPU é uma unidade de processamento paralelo com milhares de núcleos pequenos e eficientes. GPUs são amplamente usadas para acelerar operações de matriz e vetor, que são fundamentais em cálculos de redes neurais. Pode ser usada tanto por TensorFlow como PyTorch.

TPU (Tensor Processing Unit): Desenvolvida pelo Google, a TPU é uma unidade de processamento especializada para operações de tensores (matrizes multidimensionais), com foco em IA e deep learning. TPUs são otimizadas para treinamento e inferência de modelos de aprendizado profundo, especialmente em grandes escalas. Possui integração nativa com TensorFlow, mas é mais difícil de usar em PyTorch.

Para usar GPU ou TPU dentro de Google Colab, você deve editar "editar → configurações de notebook".	

Há alguns outros serviços semelhantes a Google Colab, entre eles:

- Kaggle: https://www.kaggle.com/ - Disponibilizava TPUs com mais memória que Colab (2022).
- Paperspace gradient: https://www.paperspace.com/gradient – Pagando, consegue usar GPUs melhores que Colab (2022).
- AWS

Após instalar Keras/TensorFlow no seu computador local, rode o programa abaixo usando o comando:

```
Linux$ python3 versao3-local.py
Windows> python versao3-local.py
```

```python
#versao3-local.py - para computador local
#Imprime versao de TensorFlow/Keras.
#Tambem imprime o modelo do GPU e se TensorFlow consegue usa-lo.
#Imprime versao da Cuda, CUDNN, etc.
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import tensorflow as tf
import tensorflow.keras as keras
import sys, cv2

print("Versao python3:",sys.version)
print("Versao de tensorflow:",tf.__version__)
print("Versao de OpenCV:",cv2.__version__)
print()

gpu=tf.test.gpu_device_name()
if gpu=="":
  print("Computador sem GPU.")
else:
  print("Computador com GPU:",tf.test.gpu_device_name())
  from tensorflow.python.client import device_lib
  devices=device_lib.list_local_devices()
  print("Dispositivos:",[x.physical_device_desc for x in devices if x.physical_device_desc!=""])
  os.system("cat /usr/include/cudnn_version.h | grep '#define CUDNN_MAJOR' -A 2")
  os.system("nvcc --version | grep release")
  os.system('nvidia-smi')
print()

os.system('lsb_release -a | grep "Description"') #imprime qual é o sistema operacional
os.system('cat /proc/cpuinfo | grep -E "model name"') #especificações de CPU
os.system('cat /proc/meminfo  | grep "Mem"') #especificações de RAM
print()

import torch;
print("Versao pytorch: ",torch.__version__);
print("GPU disponivel em pytorch: ",torch.cuda.is_available());
```

Programa versao3-local.py: Imprime versões das bibliotecas no computador local.

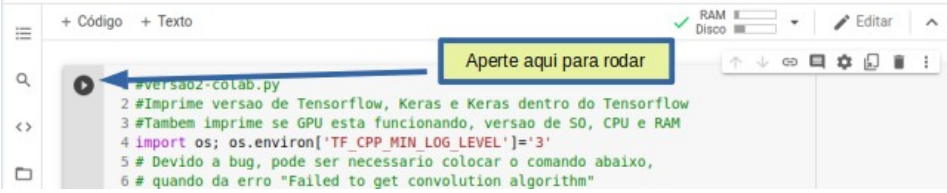
A saída obtida executando esse programa no meu computador está abaixo. Destaquei em cores as informações importantes: Python 3.8.10, TensorFlow 2.13.1, GPU RTX 2060 com 6GB, CUDNN 8.6.0, Cuda 12.4.

```
Versao python3: 3.8.10 (default, Nov 22 2023, 10:22:35) [GCC 9.4.0]
Versao de tensorflow: 2.13.1
Versao de OpenCV: 4.6.0

Computador com GPU: /device:GPU:0
Dispositivos: ['device: 0, name: NVIDIA GeForce RTX 2060, pci bus id: 0000:01:00.0, compute capability: 7.5']
#define CUDNN_MAJOR 8
#define CUDNN_MINOR 6
#define CUDNN_PATCHLEVEL 0
Cuda compilation tools, release 12.4, V12.4.131
Thu Apr 18 19:41:45 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2060        Off |   00000000:01:00.0  On |                  N/A |
| N/A   45C    P2             29W /   80W |     661MiB /   6144MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
Description:Linux Mint 20.2
model name  : Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
(...)
model name  : Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
MemTotal:       16309356 kB
MemFree:          256564 kB
MemAvailable:   10245740 kB
Versao pytorch:  1.12.0+cu102
GPU disponivel em pytorch:  True
```

No Google Colab, selecione uma máquina virtual com GPU (Editar →Configurações do Notebook →Acelerador de hardware GPU) e execute o programa abaixo:

```
#versao3-colab.py - para Colab
#Imprime versao de TensorFlow/Keras.
#Tambem imprime o modelo do GPU e se TensorFlow consegue usa-lo.
#Imprime versao da Cuda, CUDNN, etc.
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import tensorflow as tf
import tensorflow.keras as keras
import sys, cv2

print("Versao python3:",sys.version)
print("Versao de tensorflow:",tf.__version__)
print("Versao de OpenCV:",cv2.__version__)
print()

gpu=tf.test.gpu_device_name()
if gpu=="":
  print("Computador sem GPU.")
else:
  print("Computador com GPU:",tf.test.gpu_device_name())
  from tensorflow.python.client import device_lib
  devices=device_lib.list_local_devices()
  print("Dispositivos:",[x.physical_device_desc for x in devices if x.physical_device_desc!=""])
  !cat /usr/include/cudnn_version.h | grep '#define CUDNN_MAJOR' -A 2
  !nvcc --version | grep release
  !nvidia-smi
print()

!lsb_release -a | grep "Description" #imprime qual é o sistema operacional
!cat /proc/cpuinfo | grep -E "model name" #especificações de CPU
!cat /proc/meminfo  | grep "Mem" #especificações de RAM
print()

import torch;
print("Versao pytorch: ",torch.__version__);
print("GPU disponivel em pytorch: ",torch.cuda.is_available());
```

Programa versao3-colab.py: Imprime versões das bibliotecas no Google Colab.
https://colab.research.google.com/drive/1__PInkwdMB-BwCQhLd0tmt61SC6e0bdb?usp=sharing



Rodando esse programa no Colab com GPU, a saída indica: Python 3.10.12, TensorFlow 2.15.0, GPU Tesla T4 com 16GB, CUDNN 8.9.6, Cuda 12.2.

```
Versao python3: 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]
Versao de tensorflow: 2.15.0
Versao de OpenCV: 4.8.0
Computador com GPU: /device:GPU:0
Dispositivos: ['device: 0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5']
#define CUDNN_MAJOR 8
#define CUDNN_MINOR 9
#define CUDNN_PATCHLEVEL 6
Cuda compilation tools, release 12.2, V12.2.140
Thu Apr 18 23:16:10 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   46C    P0              27W /  70W |    103MiB / 15360MiB |      1%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
Description:Ubuntu 22.04.3 LTS
model name  : Intel(R) Xeon(R) CPU @ 2.30GHz
model name  : Intel(R) Xeon(R) CPU @ 2.30GHz
MemTotal:       13290480 kB
MemFree:         8330164 kB
MemAvailable:   11942096 kB
Versao pytorch:  2.2.1+cu121
GPU disponivel em pytorch:  True
```

[PSI3471 aula 11. Pular até aqui]

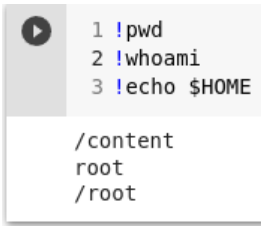
Há duas bibliotecas Keras distintas:

A) Uma biblioteca Keras independente do TensorFlow ("import keras"); e
B) Uma outra biblioteca Keras que vem incluída dentro do TensorFlow ("import tensorflow.keras as keras").

Use sempre Keras incluída dentro do Tensorflow (opção B), pois algumas funções só existem nessa versão.

[PSI3471 aula 11. Pular a partir daqui]

b) Associado à sua conta Google, você tem acesso a Google Drive e Colab. Os "notebooks" (programas Python) que você cria em Colab são salvos normalmente no diretório "Colab Notebooks" do seu Drive e não desaparecem quando você fecha a sessão Colab. Porém, todos os arquivos que você deixar no Colab (exceto os "notebooks") desaparecem quando você fecha a seção. Se você instalou alguma biblioteca dentro do Colab, esta biblioteca também não estará mais disponível quando você iniciar uma nova seção. Explicarei adiante no item (f) como contornar isto.

c) Quando você entra no computador virtual do Google Colab, o seu diretório atual é "/content", você é o usuário "root" e o seu diretório $HOME é /root.	

d) Os comandos de bash (terminal de Linux) podem ser executados dentro do Colab em "subshell" prefixando o comando com "!". Por exemplo, escrevendo "!ls" no Colab, irá obter o conteúdo do diretório atual. Se escrever "!pwd", obterá o seu diretório default.

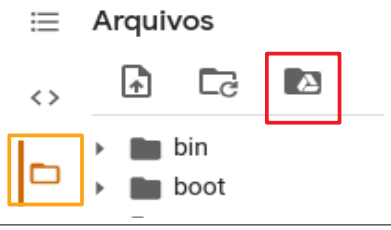
```
!ls  #lista conteúdo do diretório atual (inicialmente "sample_data")
!pwd #imprime qual é o diretório atual (normalmente "/content")
```

Nota: É também possível usar o comando *os.system("...")* de Python em vez de "!". O problema é que a saída no terminal deste comando fica invisível.

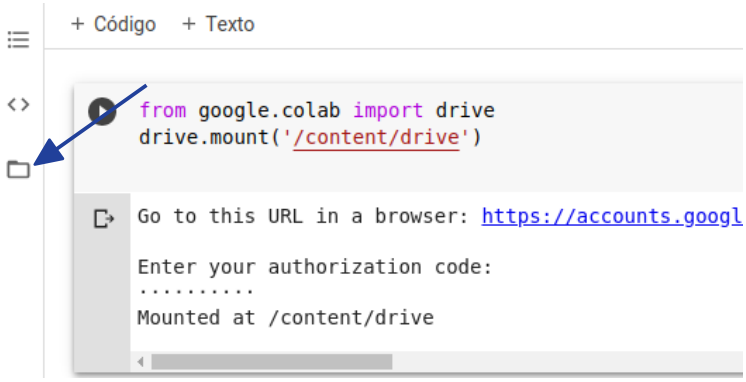
e) Para mudar o diretório atual "!cd" não funciona, pois esse comando é executado em subshell e a mudança de diretório atual é desfeita quando termina subshell. Para mudar o diretório permanentemente, deve usar "%cd" onde "%" prefixa "magic commands":

```
%cd diretorio [muda para diretorio]
%reset -f [apaga todas as variáveis e funções de Python]
```

f) Como já disse no item b, Colab não armazena permanentemente os arquivos dentro do seu computador virtual. Os arquivos são todos excluídos quando a seção Colab é fechada. Uma forma de armazenar arquivos permanentemente é você montar o seu Google Drive como um subdiretório do sistema de arquivos de Colab e deixar lá os seus arquivos. Isto pode ser feito rodando o código Python (na figura abaixo, direita). Se você clicar no botão laranja e depois no botão vermelho (na figura abaixo, esquerda), Colab cria essa célula.

	`from google.colab import drive` `drive.mount('/content/drive')`

Depois que o diretório estiver montado, você pode visualizar a sua estrutura de diretórios clicando em:



O seu Google Drive ficará montado no diretório:
`"/content/drive/MyDrive"`

Nota: Apesar do seu Google Drive parecer um diretório local Colab (`/content/drive/MyDrive/`) o seu Google Drive pode estar fisicamente bem distante do computador Colab. Assim, a transferência de arquivos entre Google Drive e Colab pode ser lenta, principalmente a transferência de vários arquivos pequenos. É muito mais rápido transferir um arquivo compactado (.zip) grande entre Google Drive e Colab do que transferir vários arquivos pequenos. Outra forma de acelerar a transferência de arquivos é trabalhar no HD local (/content) e transferir os arquivos para Google Drive no final da execução.

Nota: "cv2.imshow" do OpenCV não funciona dentro do Colab. No seu lugar, deve usar "plt.imshow" do pyplot.

```
from matplotlib import pyplot as plt
a=plt.imread("lenna.jpg")
plt.imshow(a); plt.axis("off"); plt.show()
```

Outra opção é usar cv2_imshow:

```
from google.colab.patches import cv2_imshow
cv2_imshow(img)
```

Outras dicas sobre Google Colab estão em Anexo B.

1.3 NumPy e Tensores

Para poder trabalhar com aprendizado de máquina em Python, é necessário manipular tensores (matrizes multidimensionais). Listo abaixo os comandos que considero os mais essenciais, para quem está iniciando.

1) *NumPy.* Provavelmente, todas as bibliotecas de aprendizado de máquina em Python (OpenCV, Keras, Tensorflow, PyTorch, etc) ou trabalham diretamente com tensores de NumPy ou possuem funções para importar/exportar tensores de/para NumPy. Assim, NumPy é uma espécie de "biblioteca universal" em Python para trabalhar com tensores.

2) *Descobrir o tipo de variável.* Como em Python o tipo de variável é dinâmico (pode ficar mudando de tipo ao longo do programa), é importante saber descobrir o tipo de variável num certo ponto do programa. Para isso, use "type(v)". O tipo de um tensor NumPy é "numpy.ndarray".

3) *Tamanho do tensor.* O número de dimensões do tensor pode ser determinado com "len(v.shape)". As dimensões de um tensor NumPy podem ser determinadas com "v.shape".

4) *Tipo do tensor.* O tipo de elemento de um tensor NumPy pode ser determinado com "v.dtype"

O exemplo abaixo cria um tensor NumPy 3×4 tipo "unsigned int 8" ou "uchar". Depois, imprime o tipo de variável (class 'numpy.ndarray'), o número de dimensões (2), o tamanho (3,4) e o tipo de elemento (uint8).

Exemplo:

Programa	Saída

`import numpy as np`
`v=np.empty((3,4),dtype=np.uint8)`
`print(type(v))`
`print(len(v.shape))`
`print(v.shape)`
`print(v.dtype)`
	`<class 'numpy.ndarray'>`
	`2`
	`(3, 4)`
	`uint8`

As outras funções de NumPy ou de outras bibliotecas se encontram nos respectivos manuais e tutoriais.

[PSI3471 aula 11. Pular até aqui.]

2. Regressão em Keras

Após instalar Python3/TensorFlow/Keras (ou usando Colab), vamos criar uma rede neural simples, rasa e com camadas densas (também chamado de *fully-connected* ou *inner-product*). Vamos criar uma rede neural com 2 entradas e 2 saídas. Vamos usar somente dois exemplos de treinamento: {[ax1; ay1]; [ax2; ay2]} = { [(0.9, 0.1); (0.1, 0.9)]; [(0.1, 0.9); (0.9, 0.1)] }. Isto é, quando a entrada for ax1=(0.9, 0.1) queremos obter a saída ay1=(0.1, 0.9). Quando a entrada for ax2=(0.1, 0.9) queremos obter a saída ay2=(0.9, 0.1).

A figura 1 mostra a estrutura da rede que possui duas entradas (i1 e i2, bolinhas azuis), duas saídas (bolinhas cinzentas) e quatro neurônios (h1, h2, o1 e o2) organizados em duas camadas densas (verde e amarelo).

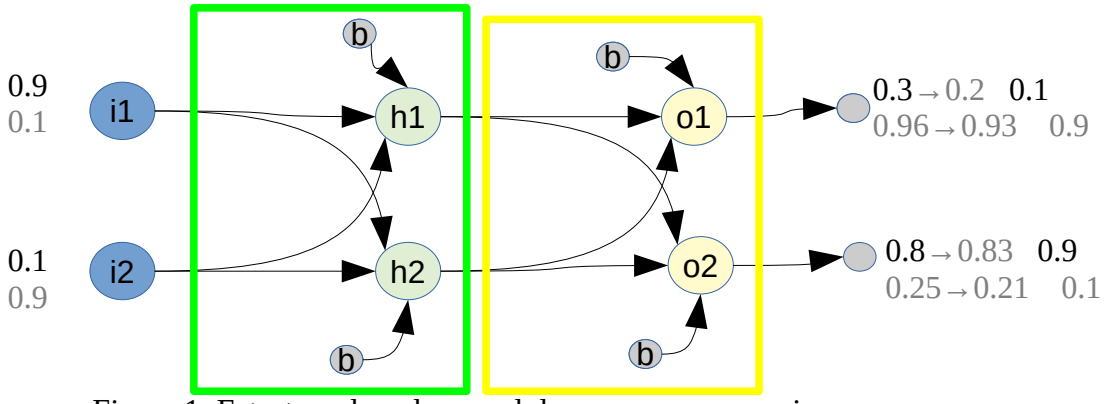


Figura 1: Estrutura da rede neural do programa regression.py

A implementação desta rede em Keras/Tensorflow está no programa 1a. As linhas 2-7 importam os módulos a serem usados no programa. A linha 2 indica que não queremos que TensorFlow mostre informações excessivas. As linhas 10-12 constroem o modelo da rede sequencial da figura 1. Uma rede sequencial é formada por uma sequência linear de camadas (sem desvios nem recorrências).

Nota: Utilize sempre o módulo “tensorflow.keras” e não o módulo “keras”, pois o primeiro é a implementação de Keras específica para Tensorflow e acrescenta várias facilidades para Tensorflow.

A implementação desta rede em Keras/PyTorch está no programa 1b.

```python
#regression.py - 2024
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Input
from tensorflow.keras import optimizers
import numpy as np

#Define modelo de rede
model = Sequential(); model.add(Input(shape=(2,)))
model.add(Dense(2, activation='sigmoid'))
model.add(Dense(2, activation='linear'))

sgd=optimizers.SGD(learning_rate=1.0)
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')
print("AX"); print(AX)
print("AY"); print(AY)

# As opcoes sao usar batch_size=2 ou 1
model.fit(AX, AY, epochs=100, batch_size=2, verbose=2)

QX = np.matrix('0.9 0.1; 0.1 0.9; 0.8 0.0; 0.2 0.9',dtype='float32')
print(QX)
QP=model.predict(QX, verbose=2)
print(QP)
```

Programa 1a: Regression.py (com backend TensorFlow)

https://colab.research.google.com/drive/1RRUsnmMIJPFiVd3v1af2DLx5R4MhoPTA?usp=sharing [~/deep/algpi/densa/regression]

```python
!pip3 install keras-core
```

```python
#regression_pt.py - 2024
import os; os.environ["KERAS_BACKEND"] = "torch"
import keras_core as keras
from keras_core.models import Sequential
from keras_core.layers import Dense, Activation, Input
from keras_core import optimizers
import numpy as np

#Define modelo de rede
model = Sequential(); model.add(Input(shape=(2,)))
model.add(Dense(2, activation='sigmoid'))
model.add(Dense(2, activation='linear'))

sgd=optimizers.SGD(learning_rate=1.0)
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')
print("AX"); print(AX)
print("AY"); print(AY)

# As opcoes sao usar batch_size=2 ou 1
model.fit(AX, AY, epochs=100, batch_size=1, verbose=2)

QX = np.matrix('0.9 0.1; 0.1 0.9; 0.8 0.0; 0.2 0.9',dtype='float32')
print(QX)
QP=model.predict(QX, verbose=2)
print(QP)
```

Programa 1b: Regression_pt.py (com backend PyTorch).

[~/deep/algpi/densa/regression/regression_pt.py]

https://colab.research.google.com/drive/1gmcPZ_dRGz9YdLhH9YU2E9nmpnKENpw1#scrollTo=6lpKU7B7Mv94

Cada neurônio da rede (figura 1) recebe as duas entradas i_1 e i_2 (isto é, dois números em ponto flutuante), multiplica-as pelos respectivos pesos w_1 e w_2, soma o viés (bias) b e aplica uma função de ativação não-linear ao resultado (no exemplo, a função sigmoide), gerando a saída a (um número em ponto flutuante). A figura 2 ilustra o funcionamento de um neurônio da rede.

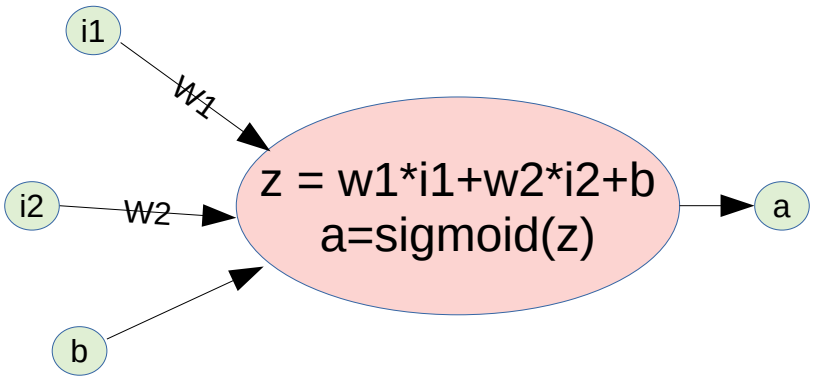


Figura 2: Funcionamento de um neurônio.

A função “sigmoide” ou “logistic” é definida como (figura 3):

$$sigmoid(x)=\frac{1}{1+e^{-x}}$$

A entrada *x* é qualquer número real e a saída fica limitada entre zero e um.

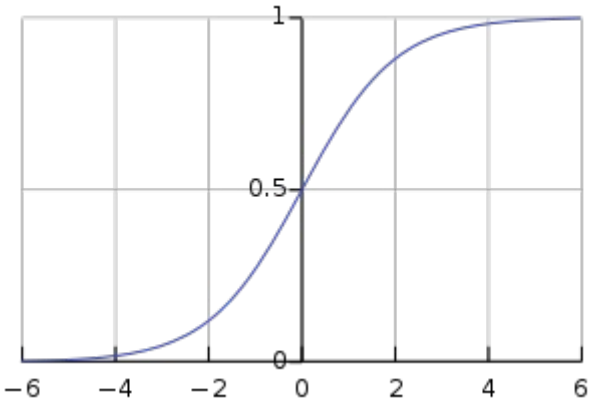


Figura 3: Função sigmoide ou logistic.

No programa, a linha 11 especifica que queremos usar a função de ativação sigmoide. A linha 12 especifica que não queremos usar função de ativação (a saída da função de ativação “linear” é igual à entrada).

Na figura 1, os 4 neurônios estão organizados em 2 camadas: verde e amarela. Cada camada recebe duas entradas e gera duas saídas. A camada verde (h1 e h2) é a camada escondida (linha 11 do programa 1) que recebe os dados de entrada. A camada amarela (o1 e o2) é a camada de saída (linha 12 do programa).

Uma rede neural *M* possui uma função custo (ou de erro ou de perda) *C* que mede o quanto a saída *qp*=*M*(*qx*), para a entrada *qx*, é diferente da saída desejada *qy: C*(*q*) = *C*(*M*(*qx*), *qy*). No nosso programa, vamos usar o erro quadrático médio (mean squared error ou mse, linha 15).

Retro-propagação (backpropagation) é o processo que ajusta, aos poucos, os parâmetros da rede para diminuir a função custo (diferença entre a saída da rede e a saída desejada), alterando “um pouco” os valores dos pesos e dos bias. Para isso, devemos calcular as derivadas parciais da função custo *C* em relação a cada peso e cada viés:

$$\frac{\partial C}{\partial w_k} \quad \text{e} \quad \frac{\partial C}{\partial b_l}$$

Nota: A apostila autodiff explica como as derivadas parciais são calculadas.

A função custo é calculada nos exemplos de treinamento organizados em lotes (mini-batches). O nosso programa possui apenas dois exemplos de treinamento (linhas 17 e 18).

$a = \{a_1, a_2\}$ = {[ax1; ay1]; [ax2; ay2]} = { [(0.9, 0.1); (0.1, 0.9)]; [(0.1, 0.9); (0.9, 0.1)] }

Assim:

a) É possível calcular a função custo para cada um dos dois exemplos de treinamento $C(a_1)$ e $C(a_2)$. Neste caso, o tamanho do lote será um e o treino tentará minimizar alternadamente cada uma das duas funções custo por vez.

b) Ou é possível calcular a média das funções de erro $C(a) = [C(a_1)+C(a_2)]/2$. Neste caso, o tamanho do batch será dois e o treino tentará minimizar a média dos erros das duas funções custo cada vez.

Vamos usar batch de tamanho dois (linha 23).

Cada derivada parcial indica para onde se deve mover para <u>*aumentar*</u> a função custo. Como queremos <u>*diminuir*</u> a função custo, movemos "um pouco" cada peso *w* e cada viés *b* no sentido contrário à sua derivada parcial (figura 4). Para especificar "um pouco", utiliza-se a taxa de aprendizado (learning rate) η:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

No programa 1, a taxa de aprendizado (learning_rate) está especificada na linha 13 como sendo 1,0. Normalmente, utiliza-se uma taxa de aprendizado menor, como 0,01 ou 0,001. Porém, neste exemplo simples, é possível usar taxa de aprendizado grande.

Existem três formas de fazer descida do gradiente.

1. *Batch gradient descent* ou *gradient descent* ou simplesmente *gradient descent* utiliza todas os *m* dados de treino para calcular a função custo médio *C*. Depois, efetua-se retro-propagação.
2. *Stochastic gradient descent*, comumente abreviado como SGD, utiliza un único dado de treino para calcular a função de custo *C* e atualiza os parâmetros.
3. *Mini-batch gradient descent:* Na descida de gradiente em mini-lotes, a função custo é calculada em subconjuntos de dados de treino (lotes). Portanto, se houver *m* observações, o número de observações em cada lote (ou mini-batch) deve ser maior que 1 e menor que *m*.

Em Keras, "stochastic gradient descent" (*sgd* na linha 14) pode significar qualquer uma das três estratégias acima, dependendo do parâmetro *batch_size* escolhido (linha 23).

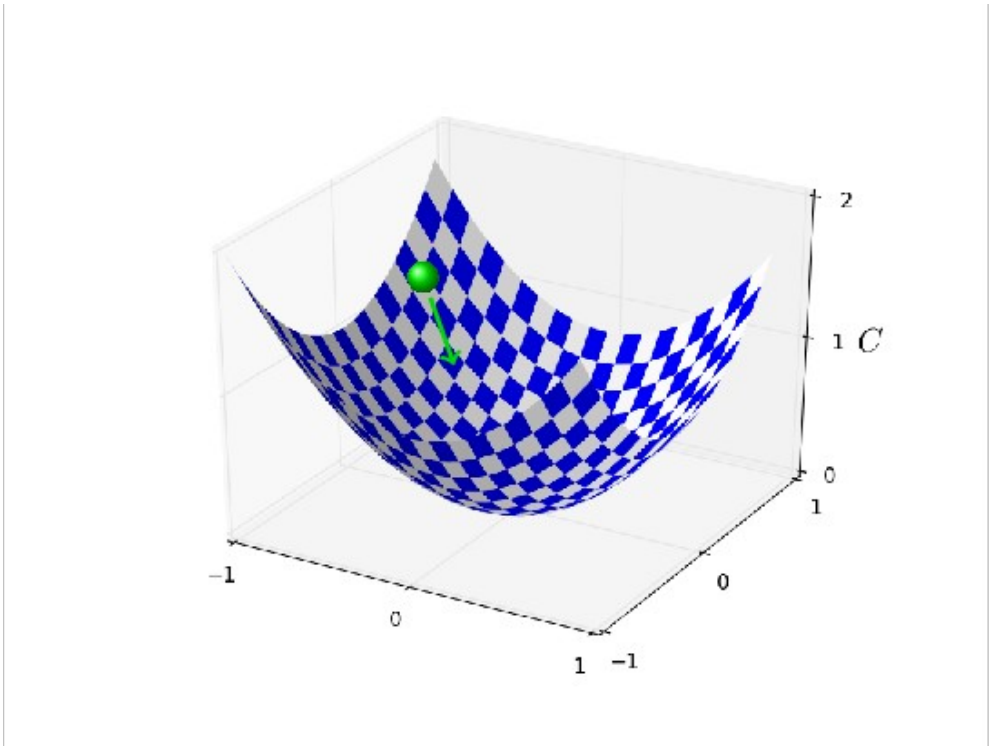


Figura 4: Retro-propagação muda os valores dos pesos e viés para diminuir a função custo (descida do gradiente, figura extraída de [Nielsen]).

A linha 23 do programa 1 executa a retro-propagação da rede neural, com 100 épocas (epochs). Cada época corresponde a executar a descida de gradiente para cada lote, até que todas as amostras de treino tenham sido usadas. No nosso caso, cada época vai executar uma única descida de gradiente (pois o número de amostras é 2 e tamanho do mini-batch é 2).

Nas linhas 25-28, aplicamos a rede neural treinada nos dados de teste {qx1, qx2, qx3, qx4} = { [0.9 0.1]; [0.1 0.9]; [0.8 0.0]; [0.2 0.9] }.

Executando o programa, obtemos:

```
AX
[[0.9 0.1]
 [0.1 0.9]]
AY
[[0.1 0.9]
 [0.9 0.1]]
Epoch   1/100 1/1 - 3s - loss: 0.5400 - 3s/epoch - 3s/step
Epoch  25/100 1/1 - 0s - loss: 0.0596 - 2ms/epoch - 2ms/step
Epoch  50/100 1/1 - 0s - loss: 8.3997e-04 - 1ms/epoch - 1ms/step
Epoch  75/100 1/1 - 0s - loss: 3.7465e-06 - 2ms/epoch - 2ms/step
Epoch 100/100 1/1 - 0s - loss: 2.4597e-08 - 1ms/epoch - 1ms/step
[[0.9 0.1]
 [0.1 0.9]
 [0.8 0. ]
 [0.2 0.9]]
1/1 - 0s - 46ms/epoch - 46ms/step
[[0.10018468 0.90000564]
 [0.89978045 0.09999689]
 [0.06250972 0.9125622 ]
 [0.88061875 0.13193148]]
```

O treino foi feito durante 100 épocas. Veja como a função custo foi diminuindo com o aumento das épocas.

As saídas obtidas estão de acordo com o treino:
Entrando [0.9, 0.1] na rede, retornou [0.100, 0.900].
Entrando [0.1, 0.9] na rede, retornou [0.900, 0,100].

Entrando na rede valores próximos aos dados de treino ([0.8 0.0] e [0.2 0.9]), as saídas foram semelhantes às saídas dos exemplos de treino parecidos.

A cada execução, a saída será um pouco diferente, pois o programa inicializa os pesos e os vieses aleatoriamente (veja Anexo A.1).

[PSI3471 aula 11. Pular a partir daqui]

P1. PyTorch - Introdução

Como vimos, Keras é uma biblioteca de alto nível que permite criar aplicações rapidamente. Como o código é muito sintético, não é possível executar tarefas "diferentes" daquelas pré-programadas. PyTorch, por outro lado, é uma biblioteca de baixo nível onde é necessário escrever explicitamente cada passo de execução. Isto torna o programa mais longo, mas é possível fazer tarefas "diferentes" das padronizadas. Por exemplo, com o único comando abaixo de Keras faz retro-propagação:

```
model.fit(AX, AY, epochs=1000)
```

Isto é bom, pois agiliza desenvolver programas. O problema é que não há uma forma direta de visualizar o que está acontecendo com a rede durante o treino, nem alterar a forma como são feitos os cálculos. O comando equivalente de PyTorch:

```
for i in range(1000):
  optimizer.zero_grad()
  AP = net(AX)
  loss = criterion(AP, AY)
  loss.backward()
  optimizer.step()
```

permite acessar os resultados intermediários e, se necessário, modificá-los.

Se o computador tiver GPU e as bibliotecas corretas estiverem instaladas, Keras vai usar automaticamente GPU para acelerar o processamento. Em PyTorch, o programador deve instruir se cada variável vai ficar em CPU ou GPU, e transferir explicitamente o conteúdo das variáveis de CPU para GPU e vice-versa. Fazer tudo automaticamente tem a vantagem de ser mais simples. Mas às vezes queremos controlar explicitamente em qual dispositivo cada variável vai ficar, principalmente quando GPU não tiver memória suficiente.

Uma desvantagem grande de PyTorch em relação a Keras/Tensorflow, neste momento, é a sua dificuldade em usar TPU (tensor processing unit) da Google. Keras consegue utilizar TPU facilmente, pois tanto Keras/Tensorflow quanto TPU são desenvolvidos pela Google. TPU costuma ter quantidade de memória muito maior (tipo 64-128 GB) do que GPU (tipo 8-16 GB). PyTorch não consegue utilizar (ainda) TPU facilmente.

P2. Regressão em PyTorch

A tradução do programa 1a/1b para PyTorch está no programa P1.

```
 1 # ~/haepi/deep/torch/densa/regression2.py
 2 """ regression2.py - pytorch - baseado em:
 3 - https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
 4 - https://towardsdatascience.com/handwritten-digit-mnist-pytorch-977b5338e627
 5 """
 6
 7 import torch
 8 import torch.nn as nn
 9 import torch.nn.functional as F
10 import numpy as np
11 import torch.optim as optim
12
13 net = nn.Sequential(
14         nn.Linear(2, 2),
15         nn.Sigmoid(),
16         nn.Linear(2, 2),
17         nn.Sigmoid())
18 print(net)
19
20 AX = torch.from_numpy( np.matrix('0.9 0.1; 0.1 0.9',dtype='float32') )
21 AY = torch.from_numpy( np.matrix('0.1 0.9; 0.9 0.1',dtype='float32') )
22 criterion = nn.MSELoss()
23 optimizer = optim.SGD(net.parameters(), lr=1)
24
25 for i in range(1000):
26   optimizer.zero_grad()   # zero the gradient buffers
27   AP = net(AX); loss = criterion(AP, AY)
28   loss.backward(); optimizer.step()
29
30 QX = torch.from_numpy( np.matrix('0.9 0.1; 0.1 0.9; 0.8 0.0; 0.2 0.9',dtype='float32') );
31 with torch.no_grad():
32   net.eval()
33   QP = net(QX)
34   print(QP.detach().numpy())
```

Programa P1: Regression2.py em PyTorch.

A estrutura da rede é definida pelo método *Sequential* (linhas 13-18). O código de treino que era um único comando “model.fit” em Keras torna-se um loop com vários comandos (linhas 25-28). A linha 18 imprime a estrutura da rede.

PyTorch trabalha com tensores próprios. Assim, é necessário converter matrizes NumPy para tensores PyTorch (linhas 20-21 e 30). E também converter tensor do PyTorch para NumPy (linha 34): detach() retira do tensor PyTorch as informações que permitem fazer retro-propagação, ficando somente com o tensor. Método numpy() faz conversão de tensor PyTorch para formato NumPy.

Para acelerar a predição, “with torch.no_grad()” (linha 31) desliga o cálculo dos gradientes. O comando “net.eval()” (linha 32) faz a rede entrar no modo de predição, pois algumas camadas funcionam de forma diferente no treino e na predição.

Saída do programa P1 em PyTorch:	Saída do programa 1a Keras:

`Sequential(`
`  (0): Linear(in_features=2, out_features=2, bias=True)`
`  (1): Sigmoid()`
`  (2): Linear(in_features=2, out_features=2, bias=True)`
`  (3): Sigmoid()`
`)`
`[[0.1120, 0.8883],`
` [0.8829, 0.1167],`
` [0.1114, 0.8896],`
` [0.8751, 0.1248]]`
	`(...)`
	`[[0.10639106 0.89681536]`
	` [0.89438605 0.10273475]`
	` [0.11006932 0.8937683 ]`
	` [0.8765248  0.12138095]]`

Os resultados de PyTorch e Keras são praticamente idênticos. Dão resultados um pouco diferentes pois os pesos são inicializados aleatoriamente.
[PSI3471 aula 11. Pular até aqui.]

3. Classificação ABC (adulto, bebê ou criança) em Keras

Vamos resolver em Keras o problema de classificar indivíduos em "adulto, bebê ou criança" a partir do seu peso em kg, usando rede neural artificial.

Nota: Numa aula passada, resolvemos este problema usando vizinho mais próximo e árvore de decisão.

Figura 5 mostra uma possível estrutura da rede para resolver este problema. Cada "flecha" da rede possui um peso associado e cada neurônio da rede possui um viés associado. Para simplificar, vamos chamar de "parâmetros da rede" o conjunto formado pelos pesos e vieses. O problema é encontrar os parâmetros que, dado o peso de um indivíduo em *kg*, o classifiquem corretamente em A, B ou C.

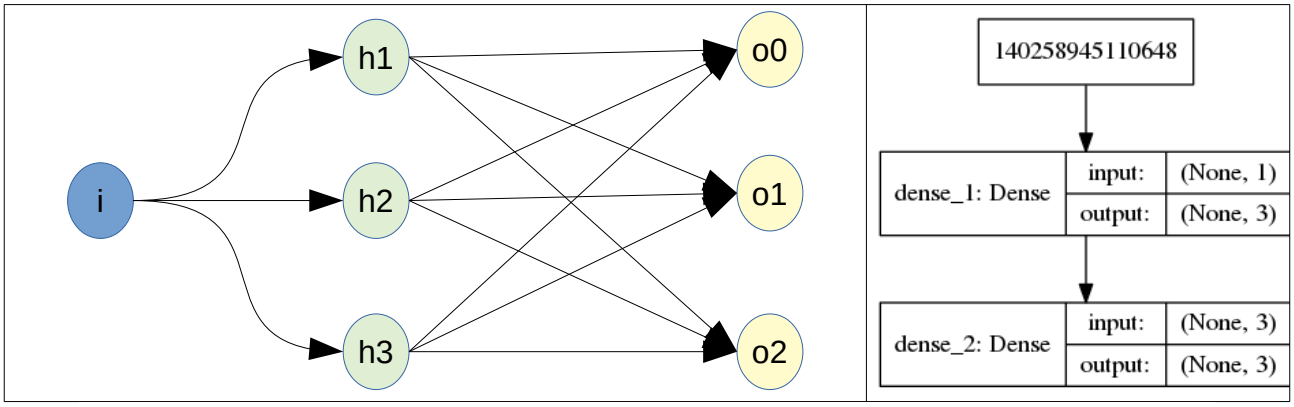


Figura 5: Estrutura da rede neural para resolver problema "Adulto", "Bebê" e "Criança".

Para usar rede neural, vamos converter os rótulos A, B e C em vetores de categorias "one-hot encoding" (1, 0, 0), (0, 1, 0) e (0, 0, 1), pois a rede da figura 5 possui três saídas (tabelas 1 e 2). Precisamos encontrar os parâmetros que fazem a classificação desejada. Para isso, o programa inicializa aleatoriamente os parâmetros (veja Anexo A.1 para maiores detalhes). Para treinar a rede, o programa efetua a retro-propagação.

Vou descrever intuitivamente *a retro-propagação* usando mini-batch de 1 elemento. O programa apresenta à rede um exemplo de treinamento (por exemplo "4") e pega a saída fornecida pela rede. A saída desejada é o vetor (0, 1, 0) significando "Bebê". Então, o programa aumenta ou diminui um pouco cada parâmetro para que a saída obtida fique um pouco mais próxima da desejada.

Tabela 1: Amostras de treinamento (*ax*, *ay*) e vetor de categoria *ay2*.

ax (características, entradas)	*ay* (rótulos, saídas)	*ay2* (one-hot-encoding)
4	B	0 1 0
15	C	0 0 1
65	A	1 0 0
5	B	0 1 0
18	C	0 0 1
70	A	1 0 0

Tabela 2: Instâncias de teste a classificar (*qx*), classificação verdadeira (*qy*) e vetor de categoria verdadeira (*qy2*).

qx (instância de teste)	*qy* (classificação verdadeira)	*qy2* (one-hot-encoding)
16	C	0 0 1
3	B	0 1 0
75	A	1 0 0

A implementação em Keras está no programa 2.

```
 1 #abc1.py - 2024
 2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
 3 import tensorflow as tf
 4 import tensorflow.keras as keras
 5 from tensorflow.keras.models import Sequential
 6 from tensorflow.keras.layers import Dense, Activation
 7 from tensorflow.keras import optimizers
 8 import numpy as np
 9 import sys
10
11 model = Sequential(); model.add(Input(shape=(1,)))
12 model.add(Dense(3, activation='sigmoid'))
13 model.add(Dense(3, activation='sigmoid'))
14 sgd=optimizers.SGD(learning_rate=10.0);
15 model.compile(optimizer=sgd, loss='mse', metrics=['accuracy'])
16
17 ax = np.matrix('4;     15;    65;    5;     18;    70   ',dtype="float32")
18 ax=2*(ax/100-0.5) #-1 a +1
19 ay = np.matrix('0 1 0; 0 0 1; 1 0 0; 0 1 0; 0 0 1; 1 0 0',dtype="float32")
20 model.fit(ax, ay, epochs=200, batch_size=2, verbose=2)
21
22 qx = np.matrix('16;    3;     75   ',dtype="float32")
23 qx=2*(qx/100-0.5) #-1 a +1
24 qy = np.matrix('0 0 1; 0 1 0; 1 0 0',dtype="float32")
25 teste = model.evaluate(qx,qy)
26 print("Custo e acuracidade de teste:",teste)
27
28 qp2=model.predict(qx)
29 print("Classificacao de teste:\n",qp2)
30 qp = qp2.argmax(axis=1)
31 print("Rotulo de saida:\n",qp)
32
33 from tensorflow.keras.utils import plot_model
34 plot_model(model, to_file='abc1.png', show_shapes=True)
35 model.summary()
```

Programa 2: Resolução do problema ABC em Keras.
https://colab.research.google.com/drive/1pNUbinzKFvQ_mAnB19RsjQB9GVpbn4eA?usp=sharing
Nota: Não pode salvar este programa com nome “abc.py”, pois parece que “abc” é uma palavra reservada em Python.

As linhas 11-15 especificam o modelo de rede, o otimizador “stochastic gradient descent” (sgd) com “learning rate” 10, e compila a rede usando “mean squared error” (mse) como medida de erro.

As linhas 17-19 especificam as entradas *ax* e saídas *ay* de treinamento. Estou normalizando o intervalo da entrada de 0 a 100 (kg) para -1 a +1. Experimente rodar o programa sem esta normalizaçao – no meu teste, a rede não convergiu e a taxa de acerto final foi de 33%, equivalendo a “chute”.

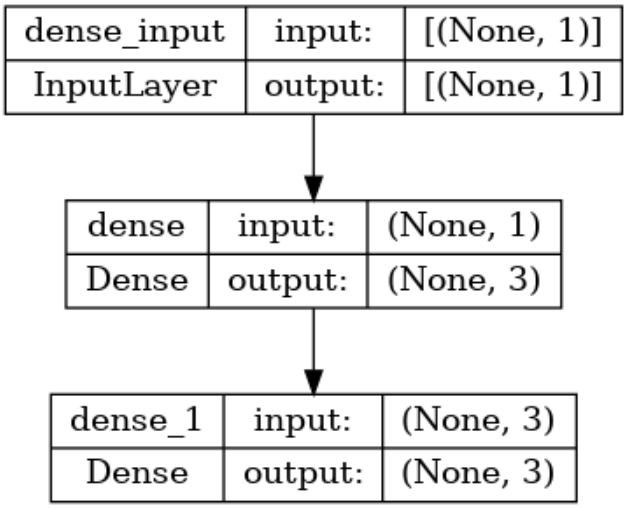
Apesar de backpropagation conseguir ajustar os parâmetros para variáveis de entradas com distribuições arbitrárias, a rede converge mais facilmente quando as variáveis de entrada estão no intervalo “semelhante” a uma distribuição normal de média zero e desvio um (por exemplo, números no intervalo [-1, +1], [-0.5, +0.5], etc).

A linha 20 efetua retro-propagação em 200 épocas, usando mini-batches de tamanho 2.

As linhas 22-24 especificam as entradas *qx* e saídas *qy* de teste. A linha 25 calcula o custo e acuracidade da rede nos dados de teste.

Nas linhas 28-31, o programa efetua as inferências de *qx* como vetores one-hot-encoding *qp2* e os converte em rótulos *qp* calculando *argmax*. Os rótulos de saída [2 1 0] estão de acordo com o que deveria dar [C B A]. As linhas 33-35 imprimem o modelo de rede da figura 5. A saída obtida é:

```
python3 abc1.py
Using TensorFlow backend.
Epoch   1/200 - 1s - loss: 0.3036 - accuracy: 0.1667 - 789ms/epoch
Epoch 100/200 - 0s - loss: 0.0740 - accuracy: 0.8333 - 4ms/epoch
Epoch 200/200 - 0s - loss: 0.0040 - accuracy: 1.0000 - 5ms/epoch
Custo e acuracidade de teste: [0.0023460739757865667, 1.0]
Classificacao de teste:
 [[4.6338744e-02 4.1173872e-02 9.0481442e-01]
 [3.3631730e-03 9.6565908e-01 6.3102752e-02]
 [9.6162802e-01 1.8744668e-05 3.9583176e-02]]
Rotulo de saida:
 [2 1 0]
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 3)                 6

 dense_1 (Dense)             (None, 3)                 12

=================================================================
Total params: 18
Trainable params: 18
Non-trainable params: 0
```



A rede apresenta acuracidade de 100%!

O número total de parâmetros é 18, pois a rede possui 12 pesos (as flechas da figura 5) e 6 vieses (os neurônios da figura 5).

Exercício: Modifique o programa abc1.py para que aceite dois atributos na entrada: peso em kg e cor da pele (0=escuro, 100=claro). Treine a rede com os dados abaixo:

ax		ay
peso	cor da pele	
4	6	B
15	8	C
65	70	A
5	90	B
18	70	C
70	10	A

Faça teste com os dados abaixo:

Instância a processar ou a classificar (*qx*):		Classificação ideal ou verdadeira desconhecida (*qy*):	Classificação pelo aprendiz (*qp*):
peso	cor da pele		
20	6	C	
3	50	B	
75	55	A	
5	80	B	
25	10	C	
65	20	A	

A rede neural artificial conseguiu identificar “cor da pele” como ruído?

Nota: A resposta esperada é que a rede neural consiga classificar corretamente todos os indivíduos mesmo com o atributo “cor da pele”. Se o seu exercício classificar incorretamente, tem algum erro no seu programa. Apesar deste exercício ser extremamente simples, há alunos que cometem diferentes tipos de erros.

[Solução privada em ~/deep/algpi/densa/abc1/pele1.py]

Exercício:
Nos sites:
https://archive.ics.uci.edu/ml/datasets/banknote+authentication OU
https://machinelearningmastery.com/standard-machine-learning-datasets item (5)
há o arquivo *data_banknote_authentication.txt* com 4 atributos extraídos de notas de banco verdadeiras (classe 0) e falsas (classe 1). Como o problema original é fácil demais, para dificultar a tarefa de classificação, acrescentei mais 3 atributos que são números aleatório entre -30 e +30.

Peguei as linhas pares desse arquivo (começando da linha 0) para treino e as ímpares para teste, gerando os arquivos *ax.txt*, *ay.txt*, *qx.txt* e *qy.txt*, conforme a convenção adotada nesta apostila. Esses arquivos estão em:
http://www.lps.usp.br/hae/apostila/noisynote.zip

Crie uma rede neural artificial em Keras que classifica as notas de teste (qx.txt) em verdadeiras (classe 0) e falsas (classe 1). Procure obter a menor taxa de erro possível Qual foi a taxa de erro de teste obtida?

Nota: Obtive erro de teste de 0,0% a 0,2% (esta taxa muda a cada execução).

Ajuda 1: O seguinte programa baixa o BD no seu diretório atual e o descompacta:

```
url='http://www.lps.usp.br/hae/apostila/noisynote.zip'
import os; nomeArq=os.path.split(url)[1]
if not os.path.exists(nomeArq):
  print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
  os.system("wget -nc -U 'Firefox/50.0' "+url)
else:
  print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
print("Descompactando arquivos novos de",nomeArq)
os.system("unzip -u "+nomeArq)
```

Ajuda 2: O seguinte código lê as matrizes ax, ay, qx e qy:

```
import numpy as np
def le(nomearq):
  with open(nomearq,"r") as f:
    linhas=f.readlines()
  linha0=linhas[0].split()
  nl=int(linha0[0]); nc=int(linha0[1])
  a=np.empty((nl,nc),dtype=np.float32)
  for l in range(nl):
    linha=linhas[l+1].split()
    for c in range(nc):
      a[l,c]=np.float32(linha[c])
  return a

ax = le("ax.txt"); ay = le("ay.txt")
qx = le("qx.txt"); qy = le("qy.txt")
```

[Solução acesso restrito: https://colab.research.google.com/drive/1FPhYI9C569EJmPq_2BmgkfjJ0KlBLEBR?usp=sharing]

[PSI3471 aula 11. Pular a partir daqui]

P3. Classificação em adulto, bebê e criança em PyTorch

O programa 2 traduzido para PyTorch está em programa P2:

```
 1 # ~/haepi/deep/torch/densa/abc1.py
 2 """abc1.py - Baseado em:
 3 - https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
 4 """
 5 import torch
 6 import torch.nn as nn
 7 import torch.optim as optim
 8 import numpy as np
 9 import sys
10
11 class Net(nn.Module):
12   def __init__(self):
13     super(Net, self).__init__()
14     self.fc1 = nn.Linear(1, 3)
15     self.fc2 = nn.Linear(3, 3)
16   def forward(self, x):
17     x = torch.sigmoid(self.fc1(x))
18     x = torch.sigmoid(self.fc2(x))
19     return x
20
21 def acuracidade(p, y):
22   p2=torch.argmax(p, 1); y2=torch.argmax(y, 1)
23   soma = (p2==y2).sum()
24   return soma
25
26 #<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
27 net = Net()
28 AX = torch.from_numpy( np.matrix('4; 15; 65; 5; 18; 70',dtype='float32') ); AX = AX/100
29 AY = torch.from_numpy( np.matrix('0 1 0; 0 0 1; 1 0 0; 0 1 0; 0 0 1; 1 0 0',dtype='float32') )
30 criterion = nn.MSELoss()
31
32 optimizer = optim.SGD(net.parameters(), lr=10)
33 batch_size=2; epochs=500
34 for i in range(epochs):
35   net.train()
36   indices = torch.randperm(AX.shape[0])
37   for j in range(AX.shape[0]//batch_size):
38     indices2=indices[batch_size*j:batch_size*(j+1)]
39     ax=AX[indices2]; ay=AY[indices2]
40     optimizer.zero_grad()   # zero the gradient buffers
41     ap = net(ax)
42     loss = criterion(ap, ay); loss.backward(); optimizer.step()
43   if i%50==0:
44     with torch.no_grad():
45       net.eval(); AP=net(AX)
46       loss=criterion(AP,AY).detach().numpy()
47       ac=acuracidade(AP,AY).numpy()/AY.shape[0]
48     print("iteracao=%3d MSE=%6.4f acurac=%3.2f%%"%(i,loss,100*ac))
49
50 QX = torch.from_numpy( np.matrix('16; 3; 75',dtype='float32') ); QX=QX/100
52 with torch.no_grad():
53   net.eval(); QP = net(QX)
54 print(QP.detach().numpy())
```

Programa P2: Resolução do problema ABC em PyTorch. Nota: Não se pode chamar o programa de "abc.py", pois parece que "abc" é uma palavra reservada em Python.

A rede tem 3 saídas (para uma saída para cada categoria A, B e C). Assim, para poder calcular a taxa de acerto, deve calcular *argmax* (o índice do neurônio de saída com a maior resposta) e verificar se os índices desejado e obtido são iguais. Para isso, foi escrita a função *acuracidade* (linhas 21-24).

A rede recebe 6 exemplos de treinamento (AX e AY). Esses 6 exemplos são divididos aleatoriamente em 3 batches de 2 exemplos cada. Linha 35 gera índices aleatórios. Linhas 37-38 pegam 2 exemplos aleatórios em cada batch. Linhas 42-47 imprimem o progresso do treino a cada 50 épocas.

Os comandos "with torch.no_grad()" e "net.eval()" fazem entrar a rede no modo de predição. O comando "net.train()" faz a rede entrar no modo de treino.

A saída obtida em PyTorch é:	A saída obtida em Keras é:

`(...)`
`[[6.1053205e-03 8.8202953e-02 8.4803396e-01]`
` [3.3946228e-04 9.5721781e-01 5.3885687e-02]`
` [9.8553991e-01 8.2046314e-10 7.2643034e-02]]`
	`(...)`
	` [[2.0841250e-02 3.5288446e-02 9.5049566e-01]`
	` [3.2123618e-04 9.8374093e-01 1.9736269e-02]`
	` [9.8454732e-01 1.4877369e-06 1.9679543e-02]]`
	`(...)`

Os resultados das duas redes são bem semelhantes.

[PSI3471 aula 11. Pular até aqui.]

4. Leitura de MNIST em Keras

Vamos recordar a nomenclatura adotada e o problema que estamos resolvendo. No aprendizado supervisionado, um "professor" fornece amostras ou exemplos de treinamento de entrada/saída (*ax*, *ay*). O "algoritmo de aprendizado" *M* classifica uma nova instância de entrada *qx* baseado nos exemplos fornecidos pelo professor, gerando a classificação *qp*. Para cada instância de entrada *qx* existe uma classificação correta *qy*. Se a classificação do algoritmo *qp*=*M*(*qx*) for igual à classificação verdadeira *qy* então o algoritmo acertou a classificação.

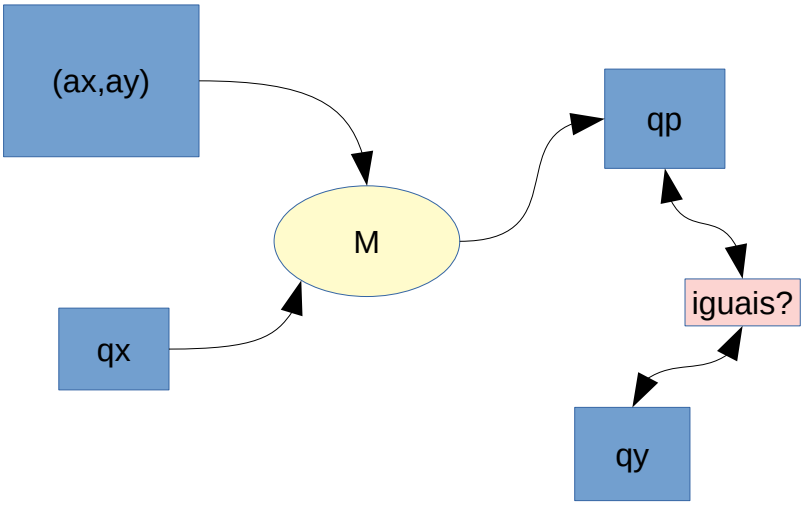


Figura 6: Nomenclatura utilizada no aprendizado supervisionado.

Além disso, vamos denotar como *ay2*, *qy2* e *qp2* os vetores "one-hot encoding" de *ay*, *qy* e *qp* respectivamente. Por exemplo, se a categoria AY=5, o vetor one-hot-encoding correspondente é um vetor de 10 elementos (número de categorias) completamente preenchido com zeros exceto na posição correspondente ao índice AY que será preenchido com um.

```
AY[0]: 5
AY2[0]: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

O banco de dados MNIST consiste de 60000 imagens de treino (AX) e 10000 imagens de teste (QX) 28×28 de dígitos manuscritos, juntamente com rótulos de 0 a 9 (AY e QY). O problema é treinar um algoritmo de aprendizado com (AX, AY) e fazer inferência com QX, obtendo QP. Quanto mais QP e QY forem iguais (maior taxa de acerto), melhor é o algoritmo.

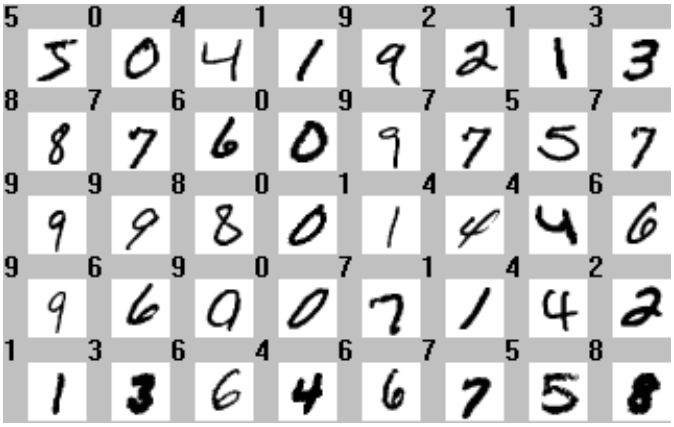


Figura 7: Alguns dígitos da MNIST, com as cores invertidas (as imagens originais possuem dígitos brancos sobre fundo preto). MNIST consiste de 60000 imagens de treino e 10000 imagens de teste 28×28.

Vamos classificar os dígitos do MNIST usando rede neural densa (não-convolucional) construído em Keras. Para isso, precisamos saber ler o banco de dados MNIST em Python.

```
 1 # le_mnist.py 2024
 2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'; import sys
 3 import tensorflow.keras as keras
 4 from tensorflow.keras.datasets import mnist
 5 from matplotlib import pyplot as plt
 6
 7 (AX, AY), (QX, QY) = mnist.load_data()
 8 print("AX:",AX.shape, AX.dtype)
 9 print("AY:",AY.shape, AY.dtype)
10 print("QX:",QX.shape, QX.dtype)
11 print("QY:",QY.shape, QY.dtype)
12
13 AX=255-AX; QX=255-QX
14 plt.imshow(AX[0],cmap="gray",interpolation="nearest")
15 plt.show()
16
17 nclasses = 10
18 AY2 = keras.utils.to_categorical(AY, nclasses)
19 QY2 = keras.utils.to_categorical(QY, nclasses)
20 print("AY2:",AY2.shape, AY2.dtype)
21 print("QY2:",QY2.shape, QY2.dtype)
22 print("AY[0]:",AY[0])
23 print("AY2[0]:",AY2[0])
```

Programa 3: Leitura de MNIST em Python/Keras.

https://colab.research.google.com/drive/1nf7foxy9N_ZZJYRS1C4LiPlDuZdfhtbr?usp=sharing [~/deep/algpi/densa/mnist/le_mnist.py]

O programa 3 lê MNIST (linha 7), imprime os formatos e os tipos das matrizes lidas (linhas 8-11):

```
AX: (60000, 28, 28) uint8
AY: (60000,) uint8
QX: (10000, 28, 28) uint8
QY: (10000,) uint8
```

Isto significa que AX é um tensor com 60.000 imagens 28×28 em níveis de cinza. AY é um vetor de uint8 com 60.000 rótulos.

Nota: a vírgula em (60000,) indica que AX.shape é um vetor com apenas um elemento: 60000. Sem vírgula, (60000) indicaria o número 60000 e não um vetor com apenas um elemento 60000.

Depois, inverte preto com branco (linha 13) e mostra na tela a primeira imagem de treino (linhas 14-15, figura 8).

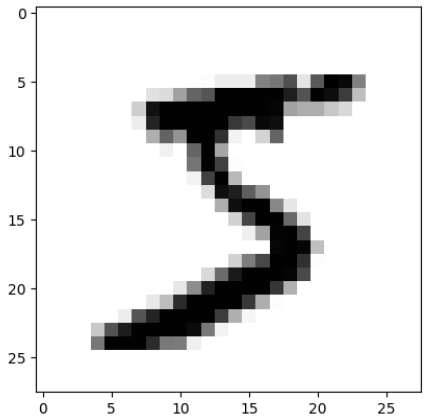


Figura 8: Imagem AX[0], primeira imagem de treino de MNIST, lida em Python.

Depois disso, o método *to_categorical* converte os rótulos AY e QY (números entre 0 e 9) em saídas categóricas "one-hot encoding" AY2 e QY2 (linhas 17-22), para poder alimentar uma rede neural com 10 neurônios de saída. As saídas impressas para compreender o que está acontecendo (linhas 20-22) são:

```
AY2: (60000, 10) float32
QY2: (10000, 10) float32
AY[0]: 5
AY2[0]: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Significando que a categoria AY[0] da primeira imagem de treino é "5". Este rótulo foi convertido em one-hot-encoding com zero em todas as posições exceto na posição de índice 5 (6ª saída) que tem valor 1. Também note que vetor de categorias AY2 é tipo float32.

[PSI3471 aula 11. Pular a partir daqui]

P4. Leitura de MNIST em PyTorch

P4.1 Leitura de MNIST como tensor

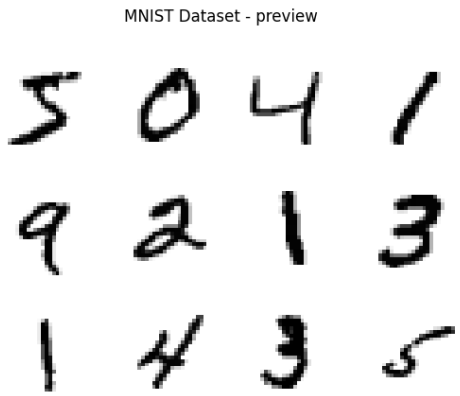
O programa P3a mostra como ler MNIST como tensores NumPy em PyTorch. A função *datasets.MNIST* faz a leitura. O parâmetro *root* indica o diretório onde ficará armazenada o BD. Após a leitura, *A.data* e *Q.data* contêm as imagens e *A.targets* e *Q.targets* contém os rótulos (0 a 9).

```
# -*- coding: utf-8 -*-
"""mnist1.py
- https://www.aiworkbox.com/lessons/load-mnist-dataset-from-pytorch-torchvision
"""
import numpy as np
import torch
import torchvision
import torchvision.datasets as datasets
import matplotlib.pyplot as plt

A = datasets.MNIST(root='/home/hae/deep/torch/data', train=True, transform=None, download=True)
Q = datasets.MNIST(root='/home/hae/deep/torch/data', train=False, transform=None, download=True)

AX=255-np.array(A.data)
AY=np.array(A.targets)
print(AX.shape,AX.dtype,AY.shape,AY.dtype)
QX=255-np.array(Q.data)
QY=np.array(Q.targets)
print(QX.shape,QX.dtype,QY.shape,QY.dtype)

nl=3; nc=4
fig = plt.figure()
for i in range(nc*nl):
  plt.subplot(nl, nc, i+1); plt.axis('off')
  plt.imshow(AX[i],cmap='gray')
fig.suptitle('MNIST Dataset - preview')
plt.show()
```

Programa P3a: Leitura de MNIST em PyTorch. [~/haepi/deep/torch/densa/mnist1.py]

Saída:

```
(60000, 28, 28) uint8 (60000,) int64
(10000, 28, 28) uint8 (10000,) int64
```

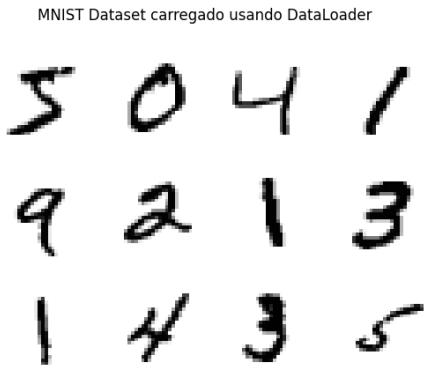


P4.2 Leitura de MNIST usando DataLoader

O programa P3b mostra como ler MNIST usando *DataLoader*, que é a forma padrão para trabalhar com imagens em PyTorch. Após fazer a leitura com *datasets.MNIST* (linhas 16 e 20), a função *torch.utils.data.DataLoader* cria *DataLoader* que irá retornar exemplos de treinamento do tamanho *batch_size* embaralhados ou não (linhas 17 e 21). Para pegar um *batch*, deve criar um iterator (linhas 18 e 22) e iterar usando *next()* (linha 28).

```
 1 # -*- coding: utf-8 -*-
 2 #mnist2.py
 3 import numpy as np
 4 import torch
 5 import torchvision
 6 from torchvision import datasets, transforms
 7 import matplotlib.pyplot as plt
 8 import sys
 9
10 #output[channel] = (input[channel] - mean[channel]) / std[channel]
11 #As imagens vão de 0 a 1. Subtrair 1 e dividir por -1 faz trocar preto com branco.
12 transform = transforms.Compose([transforms.ToTensor(),
13                                transforms.Normalize((1.0,), (-1.0,)), ] )
14
15 A = datasets.MNIST('/home/hae/deep/torch/data', download=True, train=True, transform=transform)
16 Aloader = torch.utils.data.DataLoader(A, batch_size=1, shuffle=False)
17 Aiter=iter(Aloader)
18
19 Q = datasets.MNIST('/home/hae/deep/torch/data', download=True, train=False, transform=transform)
20 Qloader = torch.utils.data.DataLoader(Q, batch_size=1, shuffle=False)
21 Qiter=iter(Qloader)
22
23 nl=3; nc=4
24 fig = plt.figure()
25 for i in range(nc*nl):
26   plt.subplot(nl, nc, i+1); plt.axis('off')
27   AX,_=Aiter.next() # ou AX,_=next(Aiter)
28   AX=AX.squeeze()
29   plt.imshow(AX,cmap='gray',vmin=0,vmax=1)
30 fig.suptitle('MNIST Dataset carregado usando DataLoader')
31 plt.savefig("mnist2.png")
32 plt.show()
```

Programa 3b: Leitura de MNIST em PyTorch usando DataLoader. [~/haepi/deep/torch/densa/mnist2.py]



[PSI3471 aula 11. Pular até aqui.]

5. Classificar MNIST em Keras usando rede neural densa

5.1 Primeira rede densa para classificar MNIST

Agora, estamos prontos para escrever programa Keras que classifica MNIST usando rede neural densa. Programa 4 é essa implementação.

```
 1 # mlp1.py
 2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
 3 import tensorflow.keras as keras
 4 from tensorflow.keras.datasets import mnist
 5 from tensorflow.keras.models import Sequential
 6 from tensorflow.keras.layers import Dense, Flatten, Normalization
 7 from tensorflow.keras import optimizers
 8 import numpy as np; import sys
 9
10 (AX, AY), (QX, QY) = mnist.load_data()
11 AX=255-AX; QX=255-QX
12
13 nclasses = 10
14 AY2 = keras.utils.to_categorical(AY, nclasses)
15 QY2 = keras.utils.to_categorical(QY, nclasses)
16
17 nl, nc = AX.shape[1], AX.shape[2] #28, 28
18
19 #1) Pseudo-normalizacao para intervalo [-0.5, +0.5]
20 #AX = (AX.astype('float32')/255.0)-0.5 # -0.5 a +0.5
21 #QX = (QX.astype('float32')/255.0)-0.5 # -0.5 a +0.5
22
23 #2) Normalizacao - distribuicao normal de media zero e desvio 1
24 #media=np.mean(AX); desvio=np.std(AX)
25 #AX=AX.astype('float32'); AX=AX-media; AX=AX/desvio; QX=QX.astype('float32'); QX=QX-media; QX=QX/desvio
26
27 #3) Normalizacao - inserir camada de normalizacao na rede
28 model = Sequential()
29 model.add(Normalization(input_shape=(nl,nc))) #Normaliza
30 model.add(Flatten())
31 model.add(Dense(400, activation='sigmoid'))
32 model.add(Dense(nclasses, activation='sigmoid'))
33
34 from tensorflow.keras.utils import plot_model
35 plot_model(model, to_file='mlp1.png', show_shapes=True)
36 model.summary()
37
38 opt=optimizers.Adam(learning_rate=0.0005)
39 model.compile(optimizer=opt, loss='mse', metrics=['accuracy'])
40
41 model.get_layer(index=0).adapt(AX) #Calcula media e desvio
42 model.fit(AX, AY2, batch_size=100, epochs=40, verbose=2)
43 score = model.evaluate(QX, QY2, verbose=0)
44 print('Test loss:', score[0])
45 print('Test accuracy:', score[1])
46 model.save('mlp1.keras')
```

Programa 4: Classificação de MNIST com rede neural densa.
https://colab.research.google.com/drive/1cdA3TbL9OH1g7fDVzSFLUQFPry7zS4u7?usp=sharing [~/deep/algpi/densa/mnist/mlp1.py]

Os números de 0 a 255 (uint8) não são adequados para alimentar uma rede neural. Para resolver isso, o programa apresenta 3 alternativas:

1) Podemos descomentar as linhas 19-21, e as matrizes AX e QX serão convertidas para o intervalo [-0.5, +0.5] (float32).

2) Outra possibilidade é normalizar subtraindo a média e dividindo pelo desvio, para que os valores sigam distribuição normal com média zero e desvio padrão um (linhas 23-25). O problema desta solução é que, quando salvar o modelo obtido num arquivo, é necessário armazenar em algum lugar a *média* e o *desvio* dos dados do treino AX, para usar os mesmos valores durante o teste.

Pergunta: Por que não se pode usar média a média e desvio dos dados de teste QX?
1) Pode ser que queira classificar uma única imagem (ou poucas imagens) e neste caso não há como calcular média e desvio dos dados de teste.
2) Pode ser que nos dados de teste predomine algumas classes. Por exemplo, se há só dígitos “0” e “1” nos dados de teste, a média e o desvio serão diferentes dos dados de treino.

3) Em vez de efetuar normalização “manualmente”, pode-se usar uma camada Keras que efetua esta operação e armazena a média e desvio dentro da rede treinada como mais dois parâmetros da rede. Esta é a solução que o programa usa. A linha 29 insere a camada de normalização e a linha 41 cal-

cula a média e o desvio de AX antes de treinar a rede. A rede vai normalizar as entradas tanto durante o treino como durante o teste usando média e desvio dos dados de treino AX. A média e o desvio de AX ficarão armazenadas na rede mlp1.keras.

Nota: A figura 10 mostra que a normalização tem 57 parâmetros em vez de 2, pois consegue fazer mais tarefas do que simplesmente subtrair média e dividir por desvio-padrão. Veja [https://keras.io/api/layers/preprocessing_layers/numerical/normalization/]

As linhas 28-32 especificam o modelo da rede neural, com uma única camada escondida com 400 neurônios. Na primeira camada é necessário especificar o formato dos dados de entrada (input_shape).

A camada ("Flatten", linha 30) é uma camada artificial que não faz nenhum processamento: só converte matriz 2D 28×28 para vetor 1D de 784 elementos, pois a próxima camada (Dense) deve obrigatoriamente receber vetores 1D.

A terceira camada (linha 31) é a camada escondida e possui 400 neurônios e ativação sigmoide.

A quarta camada (linha 32) é a camada de saída com 10 neurônios e ativação também sigmoide.

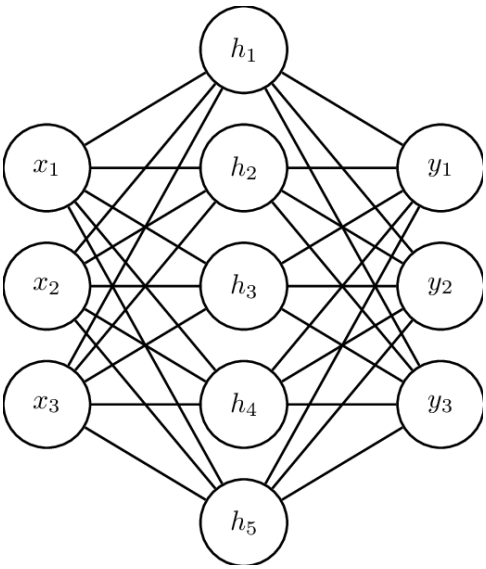
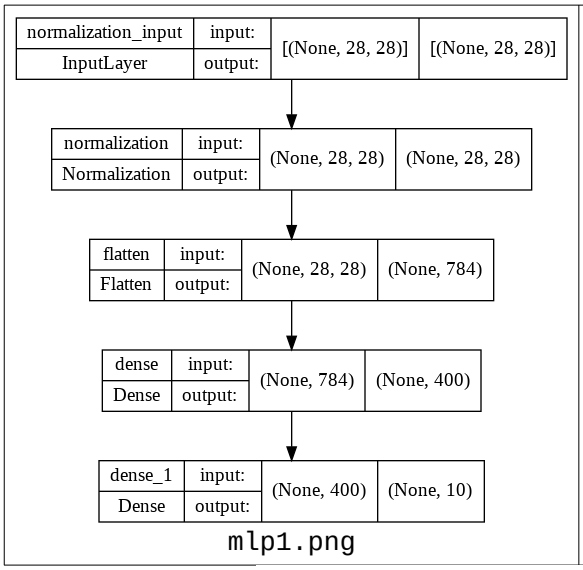


Figura 9: A rede do programa 4 recebe 28×28=784 dados de entrada (x), possui uma camada escondida com 400 neurônios (h) e uma camada de saída (y) com 10 neurônios – esta figura apresenta quantidades bem menores de neurônios.

As linhas 34-36 mostram duas diferentes formas de imprimir o modelo da rede. A saída do modelo impressa pelo programa, mostrando as 3 camadas da rede está abaixo:




```
 Layer (type)                Output Shape              Param #
=================================================================
 normalization (Normalizati  (None, 28, 28)            57
 flatten (Flatten)           (None, 784)               0
 dense (Dense)               (None, 400)               314000
 dense_1 (Dense)             (None, 10)                4010
=================================================================
Total params: 318,067
Trainable params: 318,010
Non-trainable params: 57
```

Figura 10: Duas formas de Keras mostrar a estrutura da rede.

As linhas 38-39 descrevem o otimizador (Adam), a função custo (MSE) e as métricas adicionais para exibir durante o treino (accuracy). O otimizador Adam é um otimizador melhor do que SGD e provavelmente é o mais usado na prática. Adam usa "momentum" e taxa de aprendizado adaptativo para convergir mais rapidamente à solução ótima. A função de erro continua sendo MSE e vamos imprimir também a acuracidade para melhor poder acompanhar o progresso da qualidade do modelo.

A linha 41 inicializa a camada de normalização, calculando a média e o desvio dos dados de treino. Não se esqueça de chamar *adapt* para calcular média e desvio se você colocou uma camada de normalização.

A linha 42 faz o treino. Tamanho de mini-batch é 100 (vai calcular gradiente e fazer back-propagation ajustando os parâmetros para cada lote de 100 amostras de treino), e vai rodar 40 épocas (todas as amostras de treino serão utilizadas 40 vezes). A saída obtida durante o treino é:

```
Epoch  1/40 - 1s - loss: 0.0265 - accuracy: 0.8728 - 1s/epoch - 2ms/step
Epoch 10/40 - 1s - loss: 0.0034 - accuracy: 0.9840 - 1s/epoch - 2ms/step
Epoch 20/40 - 1s - loss: 9.5126e-04 - accuracy: 0.9958 - 1s/epoch - 2ms/step
Epoch 30/40 - 1s - loss: 3.0444e-04 - accuracy: 0.9983 - 1s/epoch - 2ms/step
Epoch 40/40 - 1s - loss: 1.3329e-04 - accuracy: 0.9990 - 1s/epoch - 2ms/step
```

As funções custo e acuracidade listados acima são calculados nos dados de treino AX, AY. A função custo e acuracidade nos dados de teste são calculados e impressos nas linhas 43-45:

```
Test loss: 0.003645111806690693
Test accuracy: 0.9799000024795532
```

Isto é, a taxa de erro de teste obtida foi 2,0%, uma das menores taxas obtidas para MNIST na aula "classif-ead" usando algoritmos clássicos (o único método com taxa de erro de teste abaixo de 2% foi SVM, com 1,95%). Esta taxa é menor do que aquela que obtivemos com 1-NN. A taxa de erro de treino foi 0,01%.

Nota: Estes valores mudam a cada execução do programa, pois os pesos são inicializados aleatoriamente.

Por fim, a linha 46 salva o modelo de rede obtida. Este arquivo *mlp1.keras* permite fazer outros testes ou continuar o treino de onde parou.

5.2 Segunda rede densa para classificar MNIST

Vamos tentar melhorar a rede densa utilizando técnicas mais modernas. Já trocamos o otimizador clássico SGD (stochastic gradient descent) pelo otimizador moderno Adam (adaptive moment estimation). Vamos:

1) Colocar mais uma camada escondida.
2) Trocar função de ativação de camadas escondidas (sigmoide) pela função moderna relu (anexo A.2).
3) Trocar função de ativação da última camada (sigmoide) pela função softmax (anexo A.5).
4) Mudar a função de perda de MSE para categorical_crossentropy (anexo A.6).
5) Aumentar número de épocas de 40 para 80.

Essas alterações estão marcados em amarelo no programa 5.

```
 1 # mlp2.py
 2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
 3 import tensorflow.keras as keras
 4 from tensorflow.keras.datasets import mnist
 5 from tensorflow.keras.models import Sequential
 6 from tensorflow.keras.layers import Dense, Flatten, Normalization
 7 from tensorflow.keras import optimizers
 8 import numpy as np; import sys
 9
10 (AX, AY), (QX, QY) = mnist.load_data()
11 AX=255-AX; QX=255-QX
12
13 nclasses = 10
14 AY2 = keras.utils.to_categorical(AY, nclasses)
15 QY2 = keras.utils.to_categorical(QY, nclasses)
16
17 nl, nc = AX.shape[1], AX.shape[2] #28, 28
18
19 model = Sequential()
20 model.add(Normalization(input_shape=(nl,nc))) #Normaliza
21 model.add(Flatten())
22 model.add(Dense(400, activation='relu'))
23 model.add(Dense(100, activation='relu'))
24 model.add(Dense(nclasses, activation='softmax'))
25
26 opt=optimizers.Adam(learning_rate=0.0005)
27 model.compile(optimizer=opt, loss='categorical_crossentropy',
28   metrics=['accuracy'])
29 model.get_layer(index=0).adapt(AX) #Calcula media e desvio
30 model.fit(AX, AY2, batch_size=100, epochs=80, verbose=2);
31
32 score = model.evaluate(QX, QY2, verbose=False)
33 print('Test loss: %.4f'%(score[0]))
34 print('Test accuracy: %.2f %%'%(100*score[1]))
35 print('Test error: %.2f %%'%(100*(1-score[1])))
36 model.save('mlp2.keras')
```

Programa 5: Classificação de MNIST com rede neural denso melhorado.
https://colab.research.google.com/drive/1joGRic4qr66dcrpqU4VTCgAP8JC-iUEv?usp=sharing

Executando o programa 5 (mlp2.py), obtemos:

```
Epoch  1/80 - 2s - loss: 0.2514 - accuracy: 0.9261 - 2s/epoch - 3ms/step
Epoch 20/80 - 1s - loss: 0.0060 - accuracy: 0.9979 - 1s/epoch - 2ms/step
Epoch 40/80 - 1s - loss: 0.0025 - accuracy: 0.9992 - 1s/epoch - 2ms/step
Epoch 60/80 - 1s - loss: 1.0177e-05 - accuracy: 1.0000 - 1s/epoch - 2ms/step
Epoch 80/80 - 1s - loss: 6.7866e-08 - accuracy: 1.0000 - 1s/epoch - 2ms/step
Test loss: 0.1706
Test accuracy: 98.39 %
Test error: 1.61 %
```

Onde o erro de treino é 0,00% (não errou nada) e o erro de teste é 1,61% (era 2,01% no programa anterior). Esta taxa de erro de teste é a menor já obtida até agora usando algoritmos clássicos. Porém, não conseguimos obter uma taxa de erro substancialmente menor que as que obtivemos até agora. Usando rede neural convolucional, chegaremos a taxa de erro de menos de 0,4%.

Exercício: Os resultados obtidos acima não são comparáveis aos dos métodos vistos na apostila "classif-ead", pois lá trabalhamos com dígitos eliminando linhas/colunas brancas e redimensionado as imagens para 14×14 pixels. Modifique os programas acima para que trabalhem nas mesmas condições da aula "classificação" e compare as acurácias assim obtidas com as dos métodos daquela aula.
Nota: Fazendo isso para mlp1.py, obtive erro de 1,9%. Solução em ~/deep/algpi/densa/mnist/mlp3.py.

Exercício: Modifique os programas 4 ou 5 para obter taxa de erro de teste menor que 1,61%, sem usar rede neural convolucional. Algumas alterações possíveis são: (a) Acrescentar ou eliminar camadas. (b) Mudar o número de neurônios das camadas. (c) Mudar função de ativação. (d) Mudar o otimizador e/ou os seus parâmetros. (e) Mudar o tamanho do batch. (f) Mudar o número de épocas. (g) Eliminar linhas/colunas brancas das imagens de entrada. (h) Redimensionar as imagens de entrada. (i) Aumentar artificialmente os dados de treinamento, criando versões distorcidas das imagens. Descreva (como comentários dentro do seu programa .cpp ou .py) a taxa de erro que obteve, o tempo de processamento, as alterações feitas e os testes que fez para chegar ao seu programa com baixa taxa de erro.

Exercício: Use *SparseCategoricalCrossentropy* (anexo A.6.b2) para eliminar o cálculo *one-hot-encoding* (to_categorical) no programa 5.
[Solução privada em https://colab.research.google.com/drive/1cgBL9MkRPNebRzH9YQkzBvM0lYGpa7dm?usp=share_link]

5.3 Exemplo de carregar rede já treinada para fazer teste

O programa abaixo mostra como carregar uma rede já treinada para fazer testes. Uma rede já treinada também pode ser carregada para continuar o treino do ponto em que parou. Este programa deve ser executado após ter executado o programa 5 (mlp2.py).

```
1  #pred2.py
2  import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
3  import tensorflow.keras as keras
4  from tensorflow.keras.datasets import mnist
5  from tensorflow.keras.models import load_model
6  from tensorflow.keras.utils import to_categorical
7
8  (_,_), (QX, QY) = mnist.load_data()
9  QX=255-QX
10
11 nclasses = 10
12 QY2 = keras.utils.to_categorical(QY, nclasses)
13 nl, nc = QX.shape[1], QX.shape[2] #28, 28
14
15 model=load_model('mlp2.keras')
16 score = model.evaluate(QX, QY2, verbose=False)
17 print('Test loss: %.4f'%(score[0]))
18 print('Test accuracy: %.2f %%'%(100*score[1]))
19 print('Test error: %.2f %%'%(100*(1-score[1])))
20
21 QP2 = model.predict(QX); QP = QP2.argmax(axis=-1)
22 print(QP2[0])
23 print(QP[0])
24 #Aqui QP contem as 10000 predicoes
25 print("Imagem-teste 0: Rotulo verdadeiro=%d, predicao=%d"%(QY[0],QP[0]))
26 print("Imagem-teste 1: Rotulo verdadeiro=%d, predicao=%d"%(QY[1],QP[1]))
```

Programa 6: Lê modelo já treinado e faz predição.

https://colab.research.google.com/drive/1joGRic4qr66dcrpqU4VTCgAP8JC-iUEv?usp=sharing no segundo bloco de código.

Saída:

```
Test loss: 0.1653
Test accuracy: 98.24 %
Test error: 1.76 %
313/313 [==============================] - 1s 2ms/step
[1.0139429e-24 9.3989936e-26 1.8574848e-25 1.7209503e-16 3.3879387e-24
 2.2341992e-22 3.2218744e-31 1.0000000e+00 1.3698717e-21 7.5743600e-15]
7
Imagem-teste 0: Rotulo verdadeiro=7, predicao=7
Imagem-teste 1: Rotulo verdadeiro=2, predicao=2
```

[PSI3471 aula 11. Pular a partir daqui]

Exercício: Fashion_MNIST é um BD muito semelhante à MNIST. Consiste de 60000 imagens de treino e 10000 imagens de teste 28×28, em níveis de cinza, com as categorias:

```
categories=["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
categorias=["Camiseta", "Calça", "Pulôver", "Vestido", "Casaco", "Sandália", "Camisa", "Tênis", "Bolsa", "Botins"]
```

Esse BD pode ser carregada com os comandos:

```
from keras.datasets import fashion_mnist

(AX, AY), (QX, QY) = fashion_mnist.load_data()
```

Modifique um dos programas de rede neural densa que fizemos nesta aula para que classifique fashion_mnist (em vez de MNIST). *Ajuste os parâmetros para atingir a menor taxa de erros.* Qual foi a taxa de erro obtida? Deixe claro no vídeo e como comentário de programa a taxa de erro. Mostre as saídas das 20 primeiras imagens de teste com a classificação correta e a classificação dada pelo algoritmo. A figura abaixo mostra em azul a classificação verdadeira e em vermelho a classificação dada pelo algoritmo. Se vocês conseguirem deixar a saída mais "bonita", melhor.

```
f = plt.figure()
for i in range(20):
  f.add_subplot(4,5,i+1)
  plt.imshow( [i-ésima imagem], cmap="gray")
  plt.axis("off");
  plt.text(0,-3,[i-ésima classificação verdadeira],color="b")
  plt.text(0, 2,[i-ésima classificação do algoritmo],color="r")
plt.savefig("nome_imagem.png")
plt.show()
```

P5. Classificação de MNIST em PyTorch usando rede neural densa

P5.1a Primeiro programa de rede neural densa em CPU (usando MSE e sem usar DataLoader)

O programa 4 traduzido para PyTorch está no programa P4a

```
1  # -*- coding: utf-8 -*-
2  """mlp1.py
3  - https://www.aiworkbox.com/lessons/load-mnist-dataset-from-pytorch-torchvision
4  - https://colab.research.google.com/drive/1db-E6mWSrphqI-IMwYajBDxNHyDEjpQ1#scrollTo=XCsoAdjdLjPb
5  - http://datahacker.rs/005-pytorch-convolutional-neural-network-on-mnist-in-pytorch/
6  - https://stackoverflow.com/questions/55627780/evaluating-pytorch-models-with-torch-no-grad-vs-model-eval
7  """
8  import numpy as np
9  import torch; import torch.nn as nn
10 import torch.optim as optim
11 import torchvision; import torchvision.datasets as datasets
12 import matplotlib.pyplot as plt
13 from time import time
14
15 #<<<<<<<<<<<<<<<<<<<<<<<<<<<<
16 class Net(nn.Module):
17   def __init__(self):
18     super(Net, self).__init__()
19     self.fc1 = nn.Linear(nl*nc, 400)
20     self.fc2 = nn.Linear(400, nclasses)
21   def forward(self, x):
22     x = torch.flatten(x, 1)
23     x = torch.sigmoid(self.fc1(x))
24     x = torch.sigmoid(self.fc2(x))
25     return x
26
27 def to_categorical(y, nclasses):
28   return np.eye(nclasses, dtype='float32')[y]
29
30 def acuracidade(p, y):
31   p2=torch.argmax(p, 1); y2=torch.argmax(y, 1)
32   soma = (p2==y2).sum()
33   return soma
34
35 #<<<<<<<<<<<<<<<<<<<<<<<<<<<<
36 A = datasets.MNIST(root="/home/hae/deep/torch/data", train=True, transform=None, download=True)
37 Q = datasets.MNIST(root="/home/hae/deep/torch/data", train=False, transform=None, download=True)
38
39 AX=(255-A.data)/255.0; AY=A.targets
40 QX=(255-Q.data)/255.0; QY=Q.targets
41 nl, nc = AX.shape[1], AX.shape[2] #28, 28
42 nclasses=10
43 AY2=to_categorical(AY, nclasses); AY2=torch.from_numpy(AY2)
44 QY2=to_categorical(QY, nclasses); QY2=torch.from_numpy(QY2)
45
46 net=Net(); criterion=nn.MSELoss()
47 optimizer=optim.Adam(net.parameters())
48 batch_size=100; epochs=40
49
50 tempo0=time()
51 for i in range(epochs):
52   net.train()
53   indices = torch.randperm(AX.shape[0])
54   for j in range(AX.shape[0]//batch_size): #batch_size de 100
55     batchIndices=indices[batch_size*j:batch_size*(j+1)]
56     optimizer.zero_grad()
57     ax=AX[batchIndices]; ay2=AY2[batchIndices]
58     ap2=net(ax)
59     loss=criterion(ap2, ay2); loss.backward(); optimizer.step()
60   if (i+1)%5==0:
61     with torch.no_grad():
62       net.eval(); AP2=net(AX)
63       lossTotal=criterion(AP2,AY2).cpu().detach().numpy()
64       ac=acuracidade(AP2,AY2).cpu().numpy()/AY2.shape[0]
65       print("iteracao=%3d train_MSE=%6.4f train_acc=%3.2f%%"%(i+1,lossTotal,100*ac))
66 tempo1=time(); print("Tempo de treino=%fs"%(tempo1-tempo0))
67
68 with torch.no_grad():
69   net.eval(); QP2=net(QX)
70   lossTotal=criterion(QP2,QY2).detach().numpy()
71   ac=acuracidade(QP2,QY2).numpy()/QY2.shape[0]
72   print("Desempenho nos dados de teste:")
73   print("Test_MSE=%6.4f test_acc=%3.2f%%"%(lossTotal,100*ac))
74
75 torch.save(net.state_dict(), 'mlp1.pth')
```

Programa P4a: Classificação de MNIST com rede neural densa em PyTorch.

PyTorch não tem função pronta equivalente a função *to_categorical* do Keras. Relembrando, esta função converte:

```
5 (com nclasses=10) para [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Ela foi implementada nas linhas 27-28.

A função que calcula a acuracidade entre duas matrizes com *nclasses* colunas está nas linhas 30-34.

A linha 72 mostra como salvar a rede PyTorch. PyTorch não permite salvar a rede toda, mas somente o seu estado.

Saída PyTorch:	Saída Keras:

`iteracao=  5 train_MSE=0.0110 train_acc=93.86%`
`iteracao= 10 train_MSE=0.0071 train_acc=96.18%`
`iteracao= 15 train_MSE=0.0051 train_acc=97.27%`
`iteracao= 20 train_MSE=0.0043 train_acc=97.79%`
`iteracao= 25 train_MSE=0.0039 train_acc=98.07%`
`iteracao= 30 train_MSE=0.0030 train_acc=98.56%`
`iteracao= 35 train_MSE=0.0027 train_acc=98.68%`
`iteracao= 40 train_MSE=0.0024 train_acc=98.89%`
`Tempo de treino=40.856250s`
`Desempenho nos dados de teste:`
`Test_MSE=0.0043 test_acc=97.47%`
	`Epoch 1/40 - 2s 39us/step - loss: 0.0448 - acc: 0.7178`
	`Epoch 2/40 - 1s 22us/step - loss: 0.0191 - acc: 0.8928`
	`(...)`
	`Epoch 39/40 - 1s 22us/step - loss: 0.0024 - acc: 0.9889`
	`Epoch 40/40 - 1s 22us/step - loss: 0.0024 - acc: 0.9890`
	`Test loss: 0.004075948440702632`
	`Test accuracy: 0.9775`

As duas saídas são muito parecidas. PyTorch demorou 40s.

P5.1b Primeiro programa de rede neural densa em GPU (usando MSE e sem usar DataLoader)

```python
# -*- coding: utf-8 -*-
"""mlp1c.py
- https://www.aiworkbox.com/lessons/load-mnist-dataset-from-pytorch-torchvision
- https://colab.research.google.com/drive/1db-E6mWSrphqI-IMwYajBDxNHyDEjpQ1#scrollTo=XCsoAdjdLjPb
- http://datahacker.rs/005-pytorch-convolutional-neural-network-on-mnist-in-pytorch/
- https://stackoverflow.com/questions/55627780/evaluating-pytorch-models-with-torch-no-grad-vs-model-eval
- https://stackoverflow.com/questions/58216000/get-total-amount-of-free-gpu-memory-and-available-using-pytorch
"""
import numpy as np
import torch; import torch.nn as nn; import torch.cuda as cuda
import torch.optim as optim
import torchvision; import torchvision.datasets as datasets
import matplotlib.pyplot as plt
from time import time

#<<<<<<<<<<<<<<<<<<<<<<<<<<<<
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(nl*nc, 400)
    self.fc2 = nn.Linear(400, nclasses)
  def forward(self, x):
    x = torch.flatten(x, 1)
    x = torch.sigmoid(self.fc1(x))
    x = torch.sigmoid(self.fc2(x))
    return x

def to_categorical(y, nclasses):
  return np.eye(nclasses, dtype='float32')[y]

def acuracidade(p, y):
  p2=torch.argmax(p, 1); y2=torch.argmax(y, 1)
  soma = (p2==y2).sum()
  return soma

from pynvml import *
nvmlInit()
h = nvmlDeviceGetHandleByIndex(0)
def imp_gpu_mem(ponto):
  info = nvmlDeviceGetMemoryInfo(h)
  print('[{0}] total={1:,} free={2:,} used={3:,}'.
    format(ponto, info.total, info.free, info.used) )

#<<<<<<<<<<<<<<<<<<<<<<<<<<<<
device = 'cuda' if torch.cuda.is_available() else 'cpu'; print("Device=",device)
A = datasets.MNIST(root="/home/hae/deep/torch/data", train=True, transform=None, download=True)
Q = datasets.MNIST(root="/home/hae/deep/torch/data", train=False, transform=None, download=True)

AX=(255-A.data)/255.0; AY=A.targets
QX=(255-Q.data)/255.0; QY=Q.targets
nl, nc = AX.shape[1], AX.shape[2] #28, 28
nclasses=10
AY2=to_categorical(AY, nclasses); AY2=torch.from_numpy(AY2)
QY2=to_categorical(QY, nclasses); QY2=torch.from_numpy(QY2)

imp_gpu_mem("P1")
net=Net(); net.to(device); #net esta em GPU
criterion=nn.MSELoss()

optimizer=optim.Adam(net.parameters())
batch_size=100; epochs=40

tempo0=time()
AX=AX.to(device); AY2=AY2.to(device)  #AX e AY2 estao em GPU
for i in range(epochs):
  net.train()
  indices = torch.randperm(AX.shape[0])
  for j in range(AX.shape[0]//batch_size): #batch_size de 100
    batchIndices=indices[batch_size*j:batch_size*(j+1)]
    optimizer.zero_grad()
    ax=AX[batchIndices]; ay2=AY2[batchIndices] #ax e ay2 estao em GPU
    ap2=net(ax) #ap2 esta em GPU
    loss=criterion(ap2, ay2); loss.backward(); optimizer.step()
  if (i+1)%5==0:
    with torch.no_grad():
      net.eval(); AP2=net(AX)
      lossTotal=criterion(AP2,AY2).cpu().detach().numpy()
      ac=acuracidade(AP2,AY2).cpu().numpy()/AY2.shape[0]
      print("iteracao=%3d train_MSE=%6.4f train_acc=%3.2f%%"%(i+1,lossTotal,100*ac))
tempo1=time(); print("Tempo de treino=%fs"%(tempo1-tempo0))
imp_gpu_mem("P2")
del AX, AY2, indices, batchIndices, ax, ay2, ap2, AP2; torch.cuda.empty_cache()
imp_gpu_mem("P3")

with torch.no_grad():
  net.eval();
  QX=QX.to(device); QY2=QY2.to(device) #QX e QY2 estao em GPU
  QP2=net(QX) #QP2 esta em GPU
  QP2=QP2.cpu(); QY2=QY2.cpu() #QP2 e QY2 estao em CPU
  lossTotal=criterion(QP2,QY2).detach().numpy()
  ac=acuracidade(QP2,QY2).numpy()/QY2.shape[0]
print("Test_MSE=%6.4f test_acc=%3.2f%%"%(lossTotal,100*ac))
torch.save(net.state_dict(), 'mlp1.pth')
imp_gpu_mem("P4")
del QX, QY2, QP2, net; torch.cuda.empty_cache()
imp_gpu_mem("P5")
```

Programa P4b: Classificação de MNIST com rede neural densa em PyTorch, para GPU.

Em PyTorch, programa para CPU e GPU podem não ser iguais. O programa P4b mostra a versão para GPU (que também poderia rodar em CPU).

Saída:

```
Device= cuda
[P1] total=6,220,873,728 free=5,743,509,504 used=477,364,224
iteracao=  5 train_MSE=0.0109 train_acc=93.68%
iteracao= 10 train_MSE=0.0069 train_acc=96.13%
iteracao= 15 train_MSE=0.0051 train_acc=97.26%
iteracao= 20 train_MSE=0.0041 train_acc=97.84%
iteracao= 25 train_MSE=0.0033 train_acc=98.36%
iteracao= 30 train_MSE=0.0029 train_acc=98.56%
iteracao= 35 train_MSE=0.0026 train_acc=98.81%
iteracao= 40 train_MSE=0.0030 train_acc=98.66%
Tempo de treino=21.868091s
[P2] total=6,220,873,728 free=4,362,534,912 used=1,858,338,816
[P3] total=6,220,873,728 free=4,746,313,728 used=1,474,560,000
Test_MSE=0.0047 test_acc=97.50%
[P4] total=6,220,873,728 free=4,681,302,016 used=1,539,571,712
[P5] total=6,220,873,728 free=4,746,313,728 used=1,474,560,000
```

A função *imp_gpu_mem* mostra o estado de memória do GPU. Para isso, deve instalar:
pip3 install pynvml

É possível apagar as variáveis do GPU, liberando mais memória, com o comando “del” seguido por “torch.cuda.empty_cache()”. Porém, não é recomendado fazer isso frequentemente, pois pode deixar o programa mais lento.

P5.2 Segundo programa de rede neural densa em GPU (usando CrossEntropy e sem usar DataLoader)

```
# -*- coding: utf-8 -*-
"""mlp3.py
- https://www.aiworkbox.com/lessons/load-mnist-dataset-from-pytorch-torchvision
- https://pytorch.org/tutorials/beginner/saving_loading_models.html
- https://cnvrg.io/pytorch-cuda/?gclid=Cj0KCQiApY6BBhCsARIsAOI_GjaawBAcn04Rb2IaGCmsJfZ6QOrer_4cHpaJ9gbbA3ngmNIdQkSuuiYaAi-
OREALw_wcB
"""
import numpy as np
import torch; import torch.nn as nn; import torch.optim as optim
import torchvision; import torchvision.datasets as datasets
import matplotlib.pyplot as plt
from time import time

#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(nl*nc, 400)
    self.fc2 = nn.Linear(400, 100)
    self.fc3 = nn.Linear(100, nclasses)
  def forward(self, x):
    x = torch.flatten(x, 1)
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)  #Ou usar "x=torch.log_softmax(self.fc3(x),dim=1)" com "criterion=nn.NLLLoss()"
    return x

def acuracidade(p, y):
  p2=torch.argmax(p, 1)
  soma = (p2==y).sum() #Repare que compara p2 com y
  return soma

#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu'); print(DEVICE)
A = datasets.MNIST(root='/home/hae/deep/torch/data', train=True, transform=None, download=True)
Q = datasets.MNIST(root='/home/hae/deep/torch/data', train=False, transform=None, download=True)

AX=(255-A.data)/255.0; AY=A.targets
QX=(255-Q.data)/255.0; QY=Q.targets
nl, nc = AX.shape[1], AX.shape[2] #28, 28
nclasses=10

net = Net(); net.to(DEVICE)
criterion = nn.CrossEntropyLoss() #Ou usar "criterion=nn.NLLLoss()" com log_softmax
optimizer = optim.Adam(net.parameters())
batch_size=100; epochs=160

tempo0=time()
AX=AX.to(DEVICE); AY=AY.to(DEVICE); #AX e AY estao em GPU. print(AX.is_cuda)
for i in range(epochs):
  net.train()
  indices = torch.randperm(AX.shape[0]).to(DEVICE)
  for j in range(AX.shape[0]//batch_size):
    batchIndices=indices[batch_size*j:batch_size*(j+1)]
    optimizer.zero_grad()
    ax=AX[batchIndices]; ay=AY[batchIndices] #ax e ay2 estao em GPU
    ap2=net(ax) #ap2 esta em GPU
    loss=criterion(ap2, ay); loss.backward(); optimizer.step()
  if (i+1)%20==0:
    with torch.no_grad():
      net.eval(); AP2=net(AX)
      lossTotal=criterion(AP2,AY).cpu().detach().numpy()
      ac=acuracidade(AP2,AY).cpu().numpy()/AY.shape[0]
      print("[%3d/%3d] Train_CrossEntropy=%6.4f Train_acc=%3.2f%%"%(i+1,epochs,lossTotal,100*ac))
tempo1=time(); print("Tempo de treino=%fs"%(tempo1-tempo0))
del AX, AY, indices, batchIndices, ax, ay, ap2, AP2; torch.cuda.empty_cache() #Libera memoria GPU

with torch.no_grad():
  net.eval()
  QX=QX.to(DEVICE); QY=QY.to(DEVICE) #QX e QY estao em GPU
  QP2=net(QX)
  QP2=QP2.cpu(); QY=QY.cpu() #QP2 e QY estao em CPU
  lossTotal=criterion(QP2,QY).detach().numpy()
  ac=acuracidade(QP2,QY).numpy()/QY.shape[0]
print("CrossEntropy=%6.4f acurac=%3.2f%%"%(lossTotal,100*ac))

torch.save(net.state_dict(), 'mlp3.pth')
```

Programa P5: Classificação de MNIST com rede neural densa em PyTorch, para GPU.

O programa 5 traduzido para PyTorch resulta no programa P5. No criterion, o primeiro parâmetro é float32 e tem 10 colunas, enquanto que o segundo parâmetro é int64 e tem 1 coluna.

Saída:

```
cuda
[ 20/160] Train_CrossEntropy=0.0550 Train_acc=98.20%
[ 40/160] Train_CrossEntropy=0.0335 Train_acc=98.86%
[ 60/160] Train_CrossEntropy=0.0144 Train_acc=99.51%
[ 80/160] Train_CrossEntropy=0.0219 Train_acc=99.20%
[100/160] Train_CrossEntropy=0.0317 Train_acc=98.94%
[120/160] Train_CrossEntropy=0.0093 Train_acc=99.65%
[140/160] Train_CrossEntropy=0.0070 Train_acc=99.77%
[160/160] Train_CrossEntropy=0.0018 Train_acc=99.94%
Tempo de treino=109.502449s
CrossEntropy=0.1300 acurac=98.12%
```

Há duas possibilidades para implementar o mesmo programa:

```
1) x = self.fc3(x)    e   criterion = nn.CrossEntropyLoss()
2) x=torch.log_softmax(self.fc3(x),dim=1)   e   ecriterion=nn.NLLLoss()
```

P5.3 Exemplo de carregar rede já treinada para fazer predição

O programa abaixo mostra como carregar uma rede já treinada para fazer predição.

```python
# -*- coding: utf-8 -*-
"""pred1.py
- https://www.aiworkbox.com/lessons/load-mnist-dataset-from-pytorch-torchvision
- https://pytorch.org/tutorials/beginner/saving_loading_models.html
- https://cnvrg.io/pytorch-cuda/?gclid=Cj0KCQiApY6BBhCsARIsAOI_GjaawBAcn04Rb2IaGCmsJfZ6QOrer_4cHpaJ9gbbA3ngmNIdQkSuuiYaAi-OREALw_wcB
"""
import numpy as np
import torch; import torch.nn as nn; import torch.optim as optim
import torchvision; import torchvision.datasets as datasets
import matplotlib.pyplot as plt
from time import time

#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(nl*nc, 400)
    self.fc2 = nn.Linear(400, 100)
    self.fc3 = nn.Linear(100, nclasses)
  def forward(self, x):
    x = torch.flatten(x, 1)
    x = torch.relu(self.fc1(x))
    x = torch.relu(self.fc2(x))
    x = self.fc3(x)  #Ou usar "x=torch.log_softmax(self.fc3(x),dim=1)" com "criterion=nn.NLLLoss()"
    return x

def acuracidade(p, y):
  p2=torch.argmax(p, 1)
  soma = (p2==y).sum() #Repare que compara p2 com y
  return soma

#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu'); print(DEVICE)
Q = datasets.MNIST(root='/home/hae/deep/torch/data', train=False, transform=None, download=True)

QX=(255-Q.data)/255.0; QY=Q.targets
nl, nc = QX.shape[1], QX.shape[2] #28, 28
nclasses=10

net = Net(); net.load_state_dict(torch.load("mlp3.pth")); net.to(DEVICE)
criterion = nn.CrossEntropyLoss() #Ou usar "criterion=nn.NLLLoss()" com log_softmax
with torch.no_grad():
  net.eval()
  QX=QX.to(DEVICE); QY=QY.to(DEVICE) #QX e QY estao em GPU
  QP2=net(QX)
  QP2=QP2.cpu(); QY=QY.cpu() #QP2 e QY estao em CPU
  lossTotal=criterion(QP2,QY).detach().numpy()
  ac=acuracidade(QP2,QY).numpy()/QY.shape[0]
print("CrossEntropy=%6.4f acurac=%3.2f%%"%(lossTotal,100*ac))
```

Programa 6: Lê modelo já treinado e faz predição.

[PSI3471 aula 11. Pular até aqui.]

Referências:

[Mazur] http://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
[Nielsen] http://neuralnetworksanddeeplearning.com/
[WikiRelu] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[Optimizers] https://mlfromscratch.com/optimizers-explained/#/
[WikiSoftmax] https://en.wikipedia.org/wiki/Softmax_function

Usar Wisconsin breast cancer data
https://www.kaggle.com/code/thebrownviking20/intro-to-keras-with-breast-cancer-data-ann
Usar TensorBoard

Anexo A: Definições matemáticas (funções de ativação e loss)

A.1 Inicialização dos pesos e vieses

Cada vez que roda um programa Keras, resulta numa taxa de erro um pouco diferente, pois os pesos e vieses são inicializados aleatoriamente.

O inicializador default de Keras é (para maioria das camadas) glorot_uniform ou Xavier uniform. Do manual do Keras:

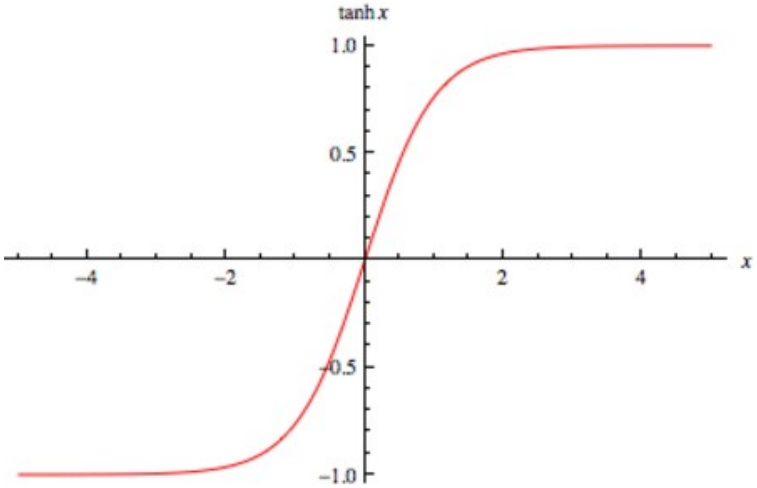
> keras.initializers.glorot_uniform(seed=None)
> Glorot uniform initializer, also called Xavier uniform initializer.
>
> It draws samples from a uniform distribution within [-limit, limit] where limit is sqrt(6 / (fan_in + fan_out)) where fan_in is the number of input units in the weight tensor and fan_out is the number of output units in the weight tensor.

Arguments
seed: A Python integer. Used to seed the random generator.

A.2 Tangente hiperbólico

$$\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

A saída vai de -1 a +1.



Keras/Tensorflow

```
# tanh.py
import tensorflow as tf
import numpy as np
a = tf.constant([-4, -2, 0, 2, 4], dtype = tf.float32)
b = tf.keras.activations.tanh(a)
print(b.numpy())
```

Saída:

```
[-0.9993292 -0.9640276  0.         0.9640276  0.9993292]
```

A sua derivada é:

$$\tanh'(x)=sech^{2}(x)=1/\cosh^{2}(x)$$

A.3 Sigmoide ou logistic

A função de ativação *sigmoide* ou *logistic* é definida como:

$$\sigma(x)=\frac{1}{1+e^{-x}}$$

e o seu gráfico está na figura abaixo. Note que a saída vai de 0 a 1.

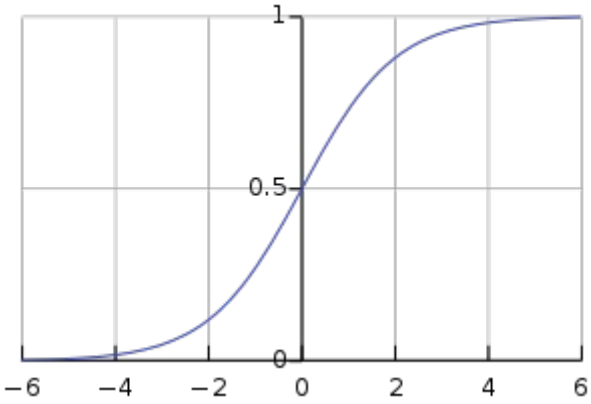


Figura: Função logística sigmoide [extraída da Wikipedia].

Keras/Tensorflow:

```
# sigmoid.py
import tensorflow as tf
import numpy as np
a = tf.constant([-4, -2, 0, 2, 4], dtype = tf.float32)
b = tf.keras.activations.sigmoid(a)
print(b.numpy())
```

Saída:

```
[0.01798621 0.11920292 0.5        0.880797   0.98201376]
```

A função inversa de *sigmoide* chama-se *logit*:

$$logit(p)=\log\left(\frac{p}{1-p}\right)$$

A derivada da sigmoide é:

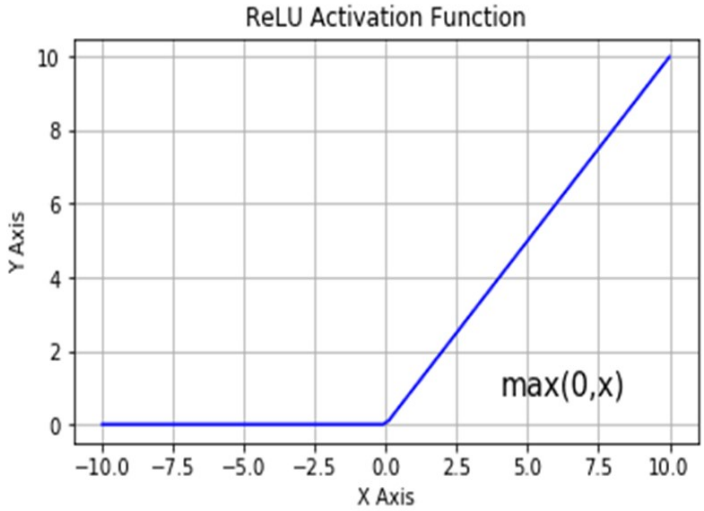
$$\sigma'(x)=\sigma(x)(1-\sigma(x))$$

A.4 ReLU

ReLU (rectified linear unit) é uma função de ativação introduzida no ano 2000 e é a função de ativação mais popular em 2017 [WikiRelu]. Foi demonstrada em 2011 que permite melhor treinamento de redes profundas. A sua definição é:

$$f(x) = max(0, x)$$

Parece a função de transferência $V_{in} \times V_{out}$ de um diodo ideal.



Curva relu (rectified linear unit).

A grande vantagem desta função de ativação é a propagação eficiente de gradiente: minimiza problema de "vanishing or exploding gradient" (veja mais detalhes no livro [Nielsen]). Note que a derivada da relu é zero ou um, o que ajuda a evitar que gradiente exploda ou desapareça.

Tensorflow/Keras

```
# relu.py
import tensorflow as tf
import numpy as np
a = tf.constant([-4, -2, 0, 2, 4], dtype = tf.float32)
b = tf.keras.activations.relu(a)
print(b.numpy())
```

Saída:

```
[0. 0. 0. 2. 4.]
```

A derivada da relu $r(x)$ é:

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

A.5 Softmax

A função *softmax*, também conhecido como *softargmax* ou *normalized exponential function*, é uma função que pega como entrada um vetor de K números reais e normaliza-o em distribuição de probabilidade com K probabilidades. Após aplicar a função *softmax*, cada componente do vetor estará no intervalo (0,1) e a soma dos componentes será 1, de forma que eles podem ser interpretados como probabilidades. Além disso, componentes de entrada maiores irão corresponder a probabilidades maiores.

A função softmax padrão $softmax: \mathbb{R}^K \rightarrow \mathbb{R}^K$ é definida como:

$$softmax(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Exemplo numérico:

Vetor de entrada z=(-0.1, 1.3, 0.3)

Calculando a soma dos exp's: $\sum_{j=1}^{K} e^{z_j} = e^{-0.1} + e^{1.3} + e^{0.2} = 5.7955$

Calculando as saídas:

$$softmax(\boldsymbol{z})_1 = \frac{e^{-0.1}}{5.7955} = 0.15613$$

$$softmax(\boldsymbol{z})_2 = \frac{e^{1.3}}{5.7955} = 0.63313$$

$$softmax(\boldsymbol{z})_3 = \frac{e^{0.2}}{5.7955} = 0.21075$$

Assim:

$$softmax(-0.1, 1.3, 0.3) = (0.15613, 0.63313, 0.21075)$$

Keras:

```
# softmax.py
import tensorflow as tf
import numpy as np
inp = np.asarray([-0.1, 1.3, 0.2])
layer = tf.keras.layers.Softmax()
print(layer(inp).numpy())
```

Saída:

```
[0.1561266  0.6331246  0.21074888]
```

Softmax pode levar diferentes entradas em saídas iguais e portanto não é possível calcular a função inversa (a não ser que se disponha de mais informações).

A.6 Cross-entropy loss:

Para entender cross-entropy loss, vou seguir a explicação de:
https://gombru.github.io/2018/05/23/cross_entropy_loss/
https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html
As funções de perda em Keras:
https://neptune.ai/blog/keras-loss-functions

a) Problemas multi-classe, multi-rótulo e binário:

Primeiro, vamos distinguir três tipos de problemas de classificação: *multi-class*, *multi-label* e binário.

1) Uma classificação multi-classe consiste em categorizar uma instância em uma de um conjunto classes possíveis. Por exemplo, classificar um dígito manuscrito MNIST nas classes ‘0’ a ‘9’.

2) Um problema multi-rótulo consiste em atribuir a uma instância um ou mais rótulos. Por exemplo, classificar um indivíduo em jovem/idoso, homem/mulher e usa/não usa óculos.

As definições de *multi-class* e *multi-label* ficam evidentes pela figura abaixo.

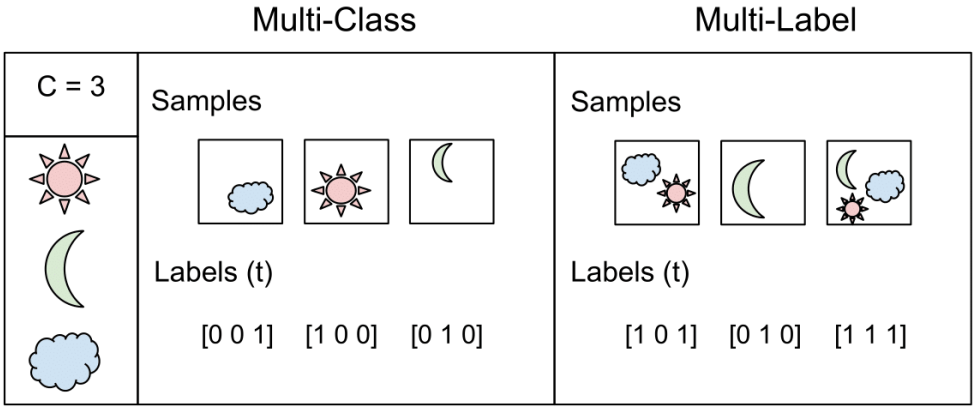


Figura de [https://gombru.github.io/2018/05/23/cross_entropy_loss/]

3) Uma classificação binária ou booleana consiste em classificar instâncias em 2 grupos, por exemplo: cachorro ou gato, câncer ou não-câncer, etc. Uma rede neural que resolve este tipo de problema pode gerar na saída um vetor com dois elementos (por exemplo, p_0 = probabilidade de ser cachorro, p_1 = probabilidade de ser gato) ou uma única saída (por exemplo, quanto mais próximo de 1 for a saída, mais provável que a imagem seja de gato). No primeiro caso, este problema pode ser considerado como um problema multi-classe de 2 classes. No segundo caso, este problema pode ser considerado como um problema multi-rótulo com apenas um rótulo.

b) Categorical cross-entropy loss:

Categorical cross-entropy é usada na classificação multi-classe ou classificação binária em redes com 2 saídas.

Normalmente, a função *softmax* é aplicada no vetor das predições *p* antes de calcular *categorical cross-entropy*, para que os elementos do vetor *p* se comportem como probabilidades. Neste caso, chame *CategoricalCrossentropy* com parâmetro *from_logits=False* (default). Se *softmax* não foi aplicada em *p*, é possível pedir que *CategoricalCrossentropy* aplique-a antes de calcular a função de perda colocando parâmetro *from_logits=True*.

Nota: O que é *logits*? Em aprendizado de máquina, *logits* indica as predições antes de passar pela normalização *softmax* [https://stackoverflow.com/questions/41455101/what-is-the-meaning-of-the-word-logits-in-tensorflow]. Também pode significar a função inversa da *sigmoide*.

Nota: A recomendação de Keras é, para maior estabilidade numérica, não calcular *softmax* e chamar *CategoricalCrossentropy* com *from_logits=True*.

Categorical cross-entropy é definida como:

$$L(p,y)=-\sum_{i=1}^{K}\log(p_i)y_i \quad \text{, sendo } log \text{ natural.}$$

onde p_i e y_i são respectivamente saída fornecida pela rede no intervalo [0, 1] e o rótulo verdadeiro {0, 1} para uma certa entrada *x* e para cada classe *i* de *K*.

Tanto em "multi-class classification" como em "binary classification" com 2 saídas, o vetor *y* contém um único componente 1, sendo o resto 0 (chamado "one-hot" labels). Assim, a equação acima reduz-se a:

$$L(p,y)=(-\log p_i)_{y_i=1} \quad .$$

Tensorflow/Keras possui duas funções para categorical cross-entropy:

b1) CategoricalCrossentropy calcula a perda entre o vetor de predições da rede e one-hot encoding:

p=(0.1, 0.8, 0.1) y=(0, 1, 0) L(p,y) = -log(0.8) = 0.22314

```
# cce.py - CategoricalCrossentropy
import tensorflow as tf
import numpy as np
y_true = [0, 1, 0]
y_pred = [0.1, 0.8, 0.1]
bce = tf.keras.losses.CategoricalCrossentropy(from_logits=False)(y_true, y_pred)
print(bce.numpy())
```

Saída: 0.22314353

b2) SparseCategoricalCrossentropy calcula a perda entre o vetor de predições da rede e o rótulo verdadeiro:

p=(0.1, 0.8, 0.1) y=1 L(p,y) = -log(y_1) = -log(0.8) = 0.22314

```
# scce.py - SparseCategoricalCrossentropy
import tensorflow as tf
import numpy as np
y_true = 1 #A categoria verdadeira e' 1
y_pred = [0.1, 0.8, 0.1]
bce = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)(y_true, y_pred)
print(bce.numpy())
```

Saída: 0.22314355

Estamos indicando, com *from_logits=False*, que já aplicamos *softmax* no vetor de predições *p*. Colocaríamos *from_logits=True* para indicar que não aplicamos *softmax*.

Quando o número de classes K é dois, o problema torna-se uma classificação binária. Para usar categorical cross-entropy, é necessário que a rede tenha 2 saídas.

c) Binary cross-entropy loss:

Binary cross-entropy é usada para "multi-label classification" e classificação binária em redes com uma única saída. Antes de aplicar binary cross-entropy, muitas vezes função sigmoide é aplicada em cada elemento do vetor de predições p para que se comporte como probabilidade.
[https://medium.com/ensina-ai/uma-explica%C3%A7%C3%A3o-visual-para-fun%C3%A7%C3%A3o-de-custo-binary-cross-entropy-ou-log-loss-eaee662c396c]

Binary cross-entropy para classificação multi-label com K classes é definida como:

$$L(p,y)=-\frac{1}{K}\left(\sum_{i=1}^{K} y_i \log(p_i)+(1-y_i)\log(1-p_i)\right)$$

Considerando que os rótulos y_i são sempre ou 0 ou 1, essa equação pode ser reescrita como:

$$L(p,y)=-\frac{1}{K}\left(\sum_{y_i=1} \log(p_i)+\sum_{y_i=0} \log(1-p_i)\right)$$

c1) Binary Cross Entropy para multi-label classification:
Exemplo:

```
p=(0.4, 0.8, 0.6)  y=(0, 1, 1)
L(p,y) = (-1/3) * ( log(1-0.4) + log(0.8) + log(0.6) ) =
         (-1/3) * ( -0.51083 -0.22314 -0.51083 ) = 0.41493
```

```python
# bce.py - BinaryCrossentropy
import tensorflow as tf
import numpy as np
y_true = [0, 1, 1]
y_pred = [0.4, 0.8, 0.6]
bce = tf.keras.losses.BinaryCrossentropy(from_logits=False)(y_true, y_pred)
print(bce.numpy())
```

Saída: 0.4149314

c2) Binary Cross Entropy para classificação binária:
Exemplo – classificar em cachorro=0 ou gato=1.

```
p=0.4  y=0
L(p,y) = log(1-0.4) = 0.51083
```

```python
# bce.py - BinaryCrossentropy
import tensorflow as tf
import numpy as np
y_true = [0, 1, 1]
y_pred = [0.4, 0.8, 0.6]
bce = tf.keras.losses.BinaryCrossentropy(from_logits=False)(y_true, y_pred)
print(bce.numpy())
```

Saída: 0.5108254

Estamos indicando, com *from_logits=False*, que já aplicamos *sigmoid* em cada elemento do vetor de predições p.

Exemplo de multi-label classification: Considere que um programa identifica se numa imagem há avião, navio, automóvel e bicicleta. Dada uma imagem *x*, o programa devolveu a saída s=[-2, -1, 2, 3], indicando que é pouco provável que haja avião e navio na imagem, mas é provável que haja automóvel e bicicleta. Os elementos deste vetor devem ser convertidos em números entre 0 e 1, para que se tornem espécie de "probabilidades". Para isso, utiliza-se a função sigmoide:

`p = sigmoide([-2, -1, 2, 3]) = [0.119, 0.269, 0.881, 0.9526]`

Suponha que os rótulos verdadeiros são *y*=[0, 0, 1, 1]. Agora, é possível aplicar *binary cross-entropy loss* a cada elemento de *p*:

BCE(p, y) = [0.1269, 0.3133, 0.1269, 0.0486]

Calculando a média, obtemos:

L(p, y) = 0.1539

Exemplo em Keras:

```
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
import keras; import numpy as np
s = np.array([-2, -1, 2, 3],dtype=np.float32)
y = np.array([0, 0, 1, 1],dtype=np.float32)
p = keras.activations.sigmoid(s); print("p=",p.numpy())
loss = keras.losses.BinaryCrossentropy(from_logits=False)
L = loss(y,p); print("L=",L.numpy())
loss = keras.losses.BinaryCrossentropy(from_logits=True)
L = loss(y,s); print("L=",L.numpy())
```

```
Saída:
p= [0.11920292 0.26894143 0.880797   0.95257413]
L= 0.15392613
L= 0.15392627
```

[Nota para mim: Acrescentar Accuracy, CategoricaAccuract e SparseCategoricalAccuracy.]

[PSI5790-2025 aula 5 parte 2. Fim]

Anexo B: Outras dicas sobre Google Colab

Descarregar arquivos de internet para Google Colab

a) É possível fazer download de um arquivo da internet e gravar no diretório de Colab usando o comando *wget* [https://linuxize.com/post/wget-command-examples/]:

```
!wget [opções] url
```

O comando acima faz download do arquivo especificado em *url* no diretório atual. Por exemplo:

```
!wget https://upload.wikimedia.org/wikipedia/en/7/7d/Lenna_%28test_image%29.png
```

baixa imagem *Lenna_(test_image).png* do site Wikipedia. Vários sites, incluindo muitos sites da USP, bloqueiam download via *wget*. É possível contornar este problema especificando a opção "--user-agent" ou "-U", emulando a requisição por um aplicativo diferente. O comando abaixo emula a requisição de download por Firefox 50.0, contornando o bloqueio:

```
!wget -nc -U 'Firefox/50.0' 'http://www.lps.usp.br/hae/apostila/hummingbird.jpg'
```

A opção "-nc" baixa o arquivo somente se precisar. O programa abaixo descarrega a imagem *hummingbird.jpg* do meu site no diretório /content do Colab (se a imagem já não estiver lá) e mostra-a na tela. Pode ser executado do Colab ou de bash:

```
#Funciona chamando python3 em bash ou Colab
url="http://www.lps.usp.br/hae/apostila/hummingbird.jpg"
import os; nomeArq=os.path.split(url)[1]
if not os.path.exists(nomeArq):
  os.system("wget -nc -U 'Firefox/50.0' "+url)
from matplotlib import pyplot as plt
a=plt.imread(nomeArq)
plt.imshow(a); plt.axis("off"); plt.show()
```

b) Normalmente, é muito mais rápido transferir um arquivo grande do que vários arquivos pequenos. Assim, se você precisa transferir um banco de dados com muitas imagens, é melhor transferir o arquivo compactado (por exemplo, como .zip) e descompactar no Colab. O programa abaixo transfere BD feiCorCorp com 400 imagens compactadas para diretório local da máquina Colab e o descompacta.

```
url='http://www.lps.usp.br/hae/apostila/feiCorCrop.zip'
import os; nomeArq=os.path.split(url)[1]
if not os.path.exists(nomeArq):
  print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
  os.system("wget -nc -U 'Firefox/50.0' "+url)
else:
  print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
print("Descompactando arquivos novos de",nomeArq)
os.system("unzip -u "+nomeArq)
```

c) Se você precisa trabalhar com base de dados (BD) com muitos arquivos em Colab, ficam lentas ambas as opções a seguir: (1) baixar BD da internet frequentemente; (b) transferir muitos arquivos do Google Drive para diretório local da máquina Colab. A solução é compactar todos os arquivos num grande arquivo BD.zip e deixá-lo no Google Drive. Toda vez que precisar, transfere-se BD.zip do Google Drive para um diretório local da máquina Colab e o descompacta:

```
# Montar drive Colab
from google.colab import drive
drive.mount('/content/drive')

# criar diretorio local na maquina Colab
!mkdir my_dir
%cd my_dir/

# Copiar BD
!cp '/content/drive/MyDrive/BD.zip' .

# Descompactar BD
!unzip BD.zip
```

d) Você pode baixar arquivo do Google Drive para Colab ou para computador local sem ter que montar drive. Para isso, use "gdown".
Nota: Esta técnica só funciona para baixar arquivos com permissão "Qualquer pessoa na Internet com o link pode ver."

No computador local, é preciso instalar este módulo:

```
pip install gdown OU
pip3 install gdown
```

No Colab, esse programa já está instalado.

Depois, digamos que queira fazer download do arquivo lenna.jpg de link:

```
https://drive.google.com/file/d/1AsE1VMFQN5vc0y0Tc5E1ZHPSnIWxocKq/view?usp=sharing
```

Para isso, dê o seguinte comando:

```
Computador_local$ gdown --id 1AsE1VMFQN5vc0y0Tc5E1ZHPSnIWxocKq  OU
Computador_local$ gdown https://drive.google.com/uc?id=1AsE1VMFQN5vc0y0Tc5E1ZHPSnIWxocKq
Google_Colab$ !gdown --id 1AsE1VMFQN5vc0y0Tc5E1ZHPSnIWxocKq  OU
Google_Colab$ !gdown https://drive.google.com/uc?id=1AsE1VMFQN5vc0y0Tc5E1ZHPSnIWxocKq
```

e) Digamos que queira fazer download do arquivo zip de Google Drive com link e descompactá-lo no diretório default:

```
https://drive.google.com/file/d/XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/view?usp=sharing
```

Para isso:

```
import os; import gdown
nomeArq="nomearq.zip"
if not os.path.exists(nomeArq):
  !gdown --id XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
!unzip -u resnet_transf
```

Ou, em Python:

```
import os; import gdown
#url="https://drive.google.com/file/d/1LrJz-IhAixFnFtcaFh3H8v3G69Dhoodn"
id="1LrJz-IhAixFnFtcaFh3H8v3G69Dhoodn"
nomeArq="nomearq.ext"
if not os.path.exists(nomeArq):
  print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
  gdown.download(id=id, output=nomeArq)
else:
  print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
```

f) Para baixar um arquivo de Colab para seu computador local:
https://predictivehacks.com/?all-tips=how-to-download-files-and-folders-from-colab

```
from google.colab import files
files.download('arquivo.ext')
```

Se quiser baixar vários arquivos ou um diretório inteiro, primeiro crie um ZIP e baixe ZIP:

```
#Zipa arquivos cnn*.png como cnn.zip e faz download no computador local
!zip cnn.zip cnn*.png
from google.colab import files
files.download('cnn.zip')
```

g) Para baixar um arquivo de Google Colab com acesso “restrito” mas ao qual você tem acesso por ser uma das “pessoas com acesso”. A forma de fazer isso está em:
https://towardsdatascience.com/different-ways-to-connect-google-drive-to-a-google-colab-notebook-pt-1-de03433d2f7a
Infelizmente esta receita não funciona para arquivos muito grandes (OverflowError: signed integer is greater than maximum). Funciona para arquivos de menos de 1 GB.

Copio abaixo a fórmula:

g1) Rode a célula abaixo:

```
from pydrive.auth import GoogleAuth
from google.colab import drive
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
import pandas as pd
```

g2) Rode a célula, depois preencher corretamente google_drive_file_id e nome_arquivo.ext.

```
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

```
file_id = 'google_drive_file_id'
download = drive.CreateFile({'id': file_id})
download.GetContentFile('nome_arquivo.ext')
```

Onde google_drive_file_id é o link do arquivo google drive. Você consegue esta informação clicando o botão direita do mouse em cima do arquivo google drive, depois clicar em “gerar link” e “copiar link”. Fazendo isso, obtém algo como:

```
https://drive.google.com/file/d/4XJ3PcBKMJoP8lLI8xHXZD_kIzsd-j5bv/view?usp=share_link
```

Desse link, você deve copiar somente a parte amarela.

Você deve substituir nome_arquivo.ext pelo nome do arquivo com que gostaria que o arquivo seja salvo.

h) Fazer download do BD da Kaggle

O site abaixo descreve como baixar um BD da Kaggle no Colab ou no seu computador local:
https://www.geeksforgeeks.org/how-to-import-kaggle-datasets-directly-into-google-colab/

Resumindo as informações desse site, você primeiro precisa criar uma conta na Kaggle:
https://www.kaggle.com/
Depois de criar a conta, na página principal da Kaggle, clique em “Account” que levará à página “Settings”. Nessa página, clique em “Create New Token”. Isso baixará um arquivo “kaggle.json”, onde você obterá o seu *username* e uma *key*, algo como:

```
{"username":"seunome","key":"9e9f84b521cd43657418455fce99120b"}
```

Agora, no seu ambiente Colab, copie o conteúdo da célula 1.

```
#https://www.geeksforgeeks.org/how-to-import-kaggle-datasets-directly-into-google-colab/
!pip3 install -q kaggle
!mkdir ~/.kaggle
!echo '{"username":"seunome","key":"9e9f84b521cd43657418455fce99120b"}' > ~/.kaggle/kaggle.json
!chmod 600 ~/.kaggle/kaggle.json
!kaggle datasets download anasmohammedtahir/covidqu
!unzip -u covidqu.zip
```

Célula 1: Célula para baixar o BD da Kaggle. Substitua parte amarela com as suas informações.

Você deve substituir o conteúdo em amarelo pelas suas informações obtidas de “kaggle.json”. A execução dessa célula irá baixar o BD no seu diretório default Colab (/content) e o descompactará.