# AULA 7
# FERRAMENTAS DE DESENVOLVIMENTO

## 1. Objetivos

- Introduzir a ferramenta de desenvolvimento Keras.
- Iniciar o desenvolvimento de soluções de problemas reais usando RNAs profundas com a ferramenta de desenvolvimento Keras.
- Desenvolvimento de RNAs com modelo sequencial usando o Keras do TensorFlow.

## 2. Ferramentas de desenvolvimento

- Praticamente ninguém desenvolve o seu próprio código para implementar e treinar uma RNA $\Rightarrow$ existem inúmeras ferramentas de desenvolvimento, já testadas, que realizam a maior parte desse trabalho e que são amplamente utilizadas.
- A grande vantagem de utilizar uma dessas ferramentas advém do fato que somente precisamos definir a configuração da RNA, ou seja, definir as operações que a RNA realiza nos dados de entrada para prever as saídas, ou seja, precisamos somente definir como é realizada a propagação para frente.
- Nessas ferramentas ao definirmos a propagação para frente, fazendo uso de manipulação simbólica são gerados automaticamente os cálculos da retro-propagação, que é de fato a parte mais difícil da codificação de uma RNA.
- As ferramentas mais utilizadas de deep-learning atualmente são as seguintes:
  - TensorFlow;
  - Keras;
  - Pytorch;
  - Caffe;
  - Theano;
  - MXNET;
  - CNTK;
  - Outras.

**Quase todas essas ferramentas são de uso livre e estão disponíveis grátis na internet.**

- Na Figura 1 é apresentada a classificação dessas ferramentas pelos usuários da área de deep-learning, o que também fornece uma indicação da porcentagem relativa de utilização dessas ferramentas, e na Figura 2 é apresentado o interesse desses usuários nessas ferramentas.

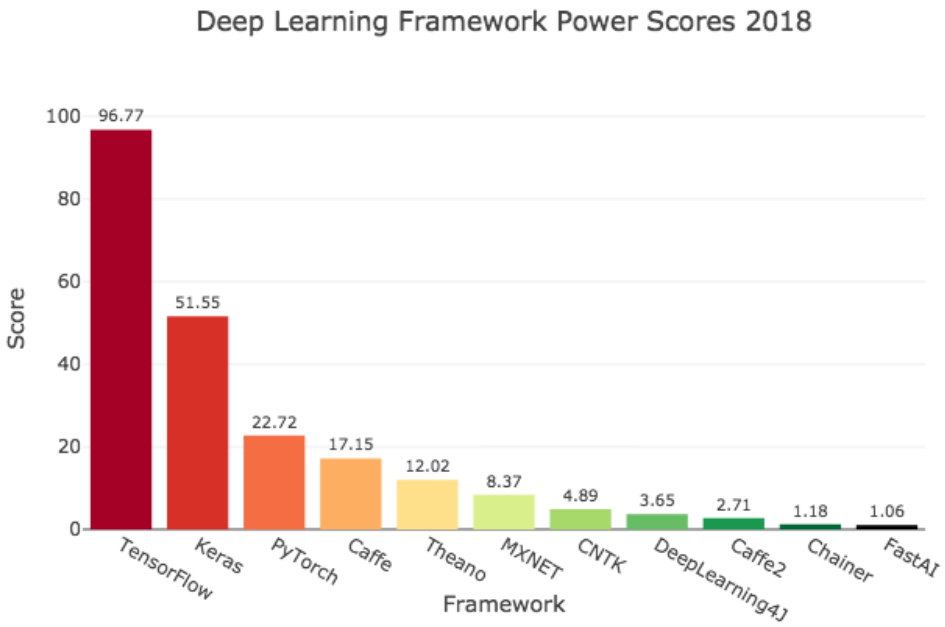



**Figura 1.** Classificação das ferramentas de desenvolvimento de sistemas de deep-learning (Jeff Hale, https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a).

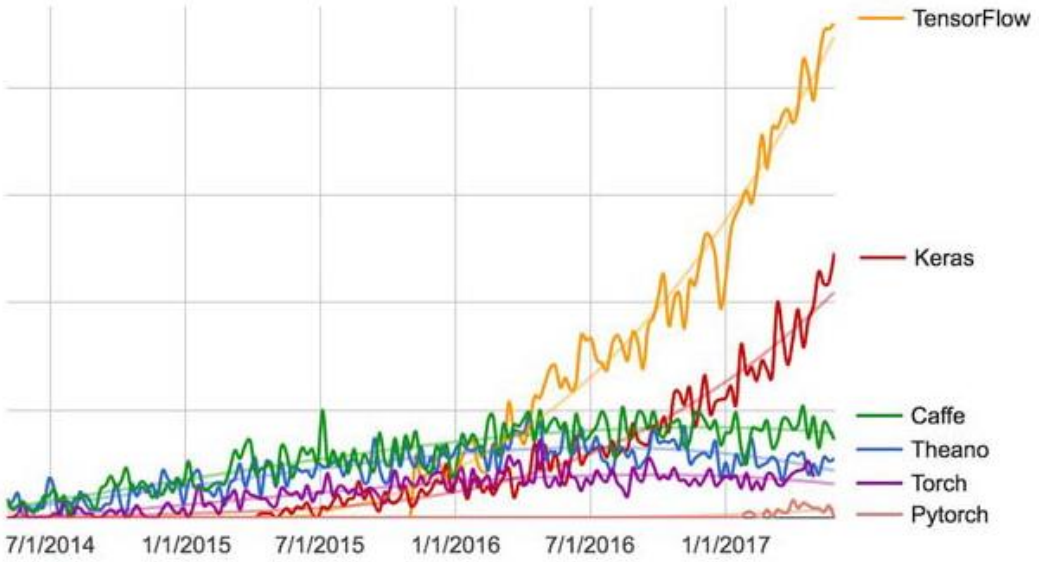



**Figura 2.** Interesse dos usuários nas diversas ferramentas de desenvolvimento de sistemas de deep-learning (François Chollet, Deep Learning with Python).

- A ferramenta mais popular na área de deep-learning é o TensorFlow, que foi desenvolvido pelo Google. Contudo, a sua utilização não é muito simples e exige conhecimento profundo de programação em Python.

- O Keras foi desenvolvido no MIT e é uma das ferramentas mais simples de usar para desenvolver novas aplicações de RNA profundas ⇒ por essa razão nesse curso usaremos o Keras.

- O Keras consiste de fato em somente uma interface mais amigável para outras ferramentas, assim, para usar o Keras devemos ter também instalado no computador o TensorFlow, ou o Theano, ou o CNTK. O Keras somente funciona com uma dessas três ferramentas.

- **Nesse curso usaremos o Keras em conjunto com o TensorFlow.**

- **Ressalta-se que em razão da popularidade e da facilidade de uso do Keras, as novas versões do TensorFlow (a partir de 1.13) incorporaram o Keras ⇒ assim, nessas versões do TensorFlow existe a possibilidade de usar o Keras do TensorFlow.**

- **Alguns comandos do Keras do TensorFlow são um pouco diferentes do Keras original, assim, um programa feito para o Keras deve ser modificado para ser executado com o Keras do TensorFlow, mas as modificações são poucas e simples.**

- **Deve-se ter muito cuidado com as inúmeras versões do TensorFlow e do Keras, porque um programa feito para uma versão mais antiga pode não funcionar com uma versão mais nova.**

# 3. Introdução ao Keras

- O keras é uma ferramenta para desenvolvimento de RNAs deep-learning, baseado na linguagem Python, que fornece uma forma simples e conveniente de construir, treinar e testar uma RNA.

- A estrutura fundamental das RNAs são as camadas, que recebem como dado de entrada um tensor e geram como saída outro tensor.

- Existem diversos tipos de camadas, sendo que cada tipo é especifica para um determinado formato de tensor e para um determinado tipo de processamento. Assim, por exemplo:

  - Dados na forma de vetores são armazenados em tensores 2D (1º eixo: exemplos; 2º eixo: características) e processados tipicamente em camadas densamente conectadas, chamadas de camadas densas;

  - Dados de imagens tons de cinza são armazenados em tensores 3D (1º eixo: exemplos; 2º eixo: altura; 3º eixo: largura) e processadas tipicamente em camadas convolutivas;

  - Seqüências de dados temporais são armazenadas em tensores 3D (1º eixo: exemplos; 2º eixo: tempo; 3º eixo: características) e processados tipicamente em camadas recorrentes, por exemplo, camadas LSTM ou GRU.

- Como visto na Aula 6, as camadas podem ser vistas como sendo blocos que utilizamos para construir uma RNA.

- A construção de uma RNA usando o Keras é feita juntando camadas compatíveis para formar um processo de transformação de dados.

- A compatibilidade de uma camada com as outras consiste no fato de que cada camada somente aceita como entrada um tensor de um dado formato e gera na sua saída outro tensor de um determinado formato.

- **No Keras existem duas formas de definir uma RNA:**

  - **Usando a classe Seqüencial, que serve para definir RNAs com uma única seqüência de camadas, que é o tipo mais comum de RNAs;**

  - **Usando a classe Funcional, que serve para configurar modelos que possuem seqüência de camadas cíclicas ou em árvore, essa forma permite configurar RNAs com arquiteturas totalmente arbitrárias.**

- Começaremos com a forma mais simples de criar uma RNA no Keras ⇒ que são os modelos sequenciais.

- A criação, treinamento e teste de uma RNA com o Keras é feito nas seguintes etapas:

  - Definição dos dados de treinamento e de teste;

  - Configuração da RNA, que consiste na definição das camadas da RNA para realizar o mapeamento das entradas nas saídas desejadas;

  - Compilação da RNA, que também inclui a configuração do processo de treinamento pela escolha da função de custo, do otimizador e da métrica para avaliar o desempenho;

  - Treinamento da RNA;

  - Teste e avaliação do desempenho da RNA.

- A documentação do Keras fornece detalhes sobre a sua utilização ⇒ https://keras.io/

- O manual do Keras do TensorFlow está disponível no link https://www.tensorflow.org/api_docs/python/tf/keras.

## 4. Dados de treinamento

- Nessa etapa os dados de treinamento e teste são organizados em tensores.

- A forma como os dados são organizados depende do tipo de dados ⇒ veremos diversos exemplos ao longo da disciplina.

- **Formato esperado dos dados no Keras ⇒ o Keras espera que o primeiro eixo dos dados, tanto de entrada, como de saída da RNA seja o número de exemplos. CUIDADO: esse**

**formato é diferente do que vimos nas aulas anteriores e também diferente do que é usado na prática comum.**

- Por exemplo ⇒ em um **problema de classificação binária** onde os dados de entrada de cada exemplo é um vetor coluna com dimensão ($n_x$, 1), a saída é um escalar e existem $m$ exemplos, então os tensores de entrada e de saída da RNA esperado pelo Keras são os seguintes:
  - Dimensão do tensor de entrada ⇒ ($m$, $n_x$), ou seja, nesse caso temos que transpor os vetores de entrada para transformá-los em vetores linha, de forma que quando colocados juntos em uma matriz, cada linha corresponde à entrada de um exemplo;
  - Dimensão do tensor de saída ⇒ ($m$, 1), ou seja, o vetor com as saídas é um vetor coluna, onde cada elemento é a saída esperada de um exemplo.

## 5. Configuração da RNA

- A estrutura de dados de uma RNA no Keras é chamada de **modelo** ⇒ um modelo é uma instância que organiza as camadas da RNA.
- Um modelo seqüencial, ou RNA seqüencial, consiste de um empilhamento de camadas.
- Antes de poder usar o Keras temos primeiramente que importar para o nosso ambiente de programação o TensorFlow. Isso é feito da seguinte forma:

```
import tensorflow as tf
tf.__version__
```

```
'1.15.0'
```

O comando `tf __version__` fornece a informação de qual versão do TensorFlow estamos usando.

- Após importar o TensorFlow podemos importar as funções do Keras que queremos usar da seguinte forma:

```
from tensorflow.keras import models
from tensorflow.keras.layers import Dense, Activation
```

Esses dois comandos importam a estrutura de modelo seqüencial e as camadas do tipo totalmente conectadas (densa) e as funções de ativação.

- Uma RNA sequencial pode ser criada simplesmente passando uma lista de instâncias de camadas para o construtor do Keras, da seguinte forma:

```
model = models.Sequential([
    Dense(32, input_shape=(1024,)),
    Activation('relu'),
    Dense(1),
    Activation('sigmoid'),
])
```

- Esse comando define uma RNA de uma camada intermediária e uma camada de saída com as seguintes características:
  - Os dados de entrada de cada exemplo de treinamento são vetores linha de dimensão (1024);
  - Observa-se que não é incluída a dimensão do segundo eixo do tensor de entrada no argumento `input_shape`, porque nesse momento não se conhece o número de exemplos que serão usados no treinamento;
  - **Cuidado ⇒ apesar de parecer que no argumento `input_shape` o segundo eixo é o número de exemplos, o Keras espera que o primeiro eixo do tensor de entrada seja o número de exemplos;**
  - A camada escondida é do tipo densa (totalmente conectada), possui 32 neurônios e a sua função de ativação é Relu;
  - A camada de saída é do tipo densa (totalmente conectada), possui 1 único neurônio e a sua função de ativação é sigmóide.
- **O nome utilizado para essa RNA foi `model`, mas poderia ser dado qualquer outro nome.**

➢ A mesma RNA poderia também ser criada em etapas usando comandos para adicionar uma camada por vez, usando o método `.add()`, como segue:

```
from tensorflow.keras import models
from tensorflow.keras import layers

rna = models.Sequential()
rna.add(layers.Dense(units=32, activation='relu', input_dim=1024))
rna.add(layers.Dense(units=1, activation='sigmoid'))
```

- Nesse caso o nome dado ao modelo é `rna`.
- Nesse caso no lugar de importar somente a estrutura de modelo seqüencial, o primeiro comando importa do Keras todos os tipos de modelos.
- O segundo comando importa do Keras todos os tipos de camadas, no lugar de somente as camadas do tipo densa, como feito anteriormente.
- O terceiro comando cria a instância da RNA usando um modelo sequencial.
- O quarto comando adiciona a primeira camada da RNA do tipo densa, com 32 neurônios, com função de ativação Relu, cuja entrada é um vetor linha de dimensão (1024).

- O quinto comando adiciona uma segunda camada tipo densa, com um único neurônio, com função de ativação sigmóide.

## Dimensões dos dados de entrada

- A RNA precisa saber as dimensões dos dados de entrada ⇒ por essa razão a primeira camada em um modelo seqüencial precisa receber a informação sobre as dimensões dos eixos dos dados de entrada.

- Observa-se que somente a primeira camada precisa dessa informação ⇒ o Keras automaticamente infere as dimensões dos dados de entrada das outras camadas da RNA usando a informação dos números de neurônios de cada camada.

- Existem diversas formas para definir a dimensão dos dados de entrada da RNA:

  - Passando o argumento `input_shape` para a primeira camada. Esse argumento é uma tuple de inteiros ou simplesmente `None`, onde `None` indica que qualquer número inteiro positivo pode ser esperado;

  - **No argumento `input_shape` o número de exemplos não é incluído ⇒ o Keras infere esse número automaticamente a partir dos dados de entrada fornecidos;**

  - Algumas camadas 2D, tais com as densas, suportam a especificação dos dados de entrada também via argumento `input_dim`.

  Como exemplo, os seguintes comandos são equivalentes:

```
from tensorflow.keras import models
from tensorflow.keras.layers import Dense, Activation

rna = models.Sequential()
rna.add(Dense(32,input_shape=(1024,)))
```

- Se for necessário especificar um número fixo de exemplos, pode-se passar o argumento `batch_size` para a camada de entrada. Assim, por exemplo, se for usado `batch_size=32` e `input_shape=(6, 8)` para a primeira camada, será esperado como dados de entrada um tensor de dimensão (32, 6, 8).

- O desenvolvimento de uma RNA exige muitas iterações até se obter um resultado desejável, assim, para evitar executar inúmeras vezes os mesmos comandos de configuração de uma RNA, que podem ser muitos dependendo do tamanho da RNA, pode-se criar uma função para configurar a RNA. Para isso temos, por exemplo, a seguinte função:

```
def build_model(data_shape):
    rna = models.Sequential()
    rna.add(layers.Dense(units=64, activation='relu',
                         input_shape=data_shape))
    rna.add(layers.Dense(units=32, activation='relu'))
    rna.add(layers.Dense(units=1, activation='sigmoid'))
    return rna
```

- Nesse caso o argumento `data_shape` representa a dimensão dos dados de entrada sem considerar o número de exemplos.

## Visualização da RNA configurada

➢ Para visualizar a arquitetura de uma RNA que foi criada usa-se no Keras o método `summary()`, com segue:

```
from tensorflow.keras import models
from tensorflow.keras import layers

data_shape = (12288,)
rna = build_model(data_shape)
rna.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 64)                786496
_________________________________________________________________
dense_7 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 33
=================================================================
Total params: 788,609
Trainable params: 788,609
Non-trainable params: 0
_________________________________________________________________
```

➢ A arquitetura da RNA é apresentada em uma tabela, cujo conteúdo é o seguinte:

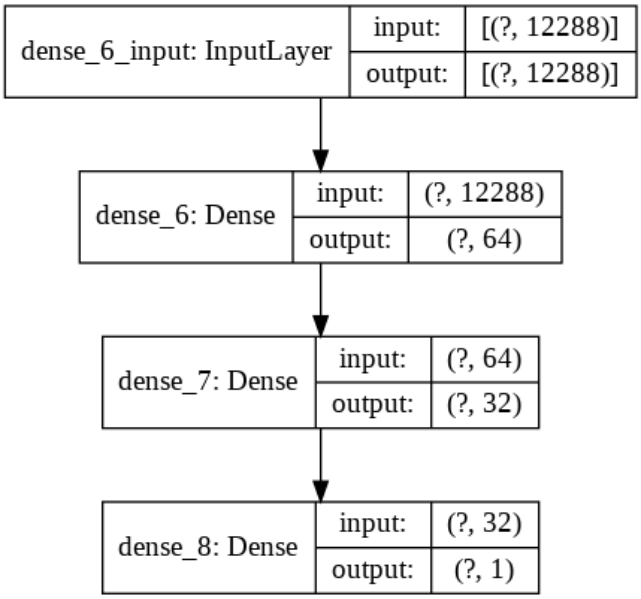
- Primeira coluna fornece os tipos de camadas da rede e numera as camadas;
- Segunda coluna apresenta o número de neurônios da camada;
- Terceira coluna apresenta o número de parâmetros da camada ⇒ **como calcular esses números?**

➢ É importante visualizar a arquitetura da RNA porque qualquer mensagem de erro na sua compilação ou no treinamento, referência a camada pelo seu número, que pode ser obtido pelo comando `summary`.

- Outra forma de visualizar uma RNA no Keras é fazer um gráfico da mesma usando a função `plot_model`, como segue:

```
from tensorflow.keras.utils import plot_model
Import pydot

plot_model(rna, to_file='rna.png', show_shapes=True)
plot_model(rna, to_file='/Meu_diretorio/rna.png', show_shapes=True)
```




- Esse código gera um gráfico da `rna` e também salva esse gráfico em um arquivo tipo png com o diagrama das conexões da `rna`.
- No caso, o gráfico da `rna` é salvo no arquivo `rna.png` que fica no diretório `Meu_diretorio`. Se você estiver executando o seu código no mesmo diretório onde estão os seus arquivos, então, não precisa incluir o diretório, basta incluir o nome do arquivo que ele será salvo no mesmo diretório onde está o seu notebook.
- Antes de usar a função `plot_model` é necessário importá-la do Keras e também a biblioteca pydot do Python, que fornece funções para processar gráficos.

## 6. Compilação

- A geração da RNA é realizada na etapa de compilação, onde é escolhida a função de custo, uma métrica para avaliação da RNA e configurado o método de treinamento.
- Um exemplo simples de compilação de uma RNA é o seguinte:

```
from tensorflow.keras import optimizers

rna.compile(optimizer='SGD', loss='binary_crossentropy',
            metrics=['accuracy'])
```

- O primeiro comando importa do Keras os otimizadores.
- O segundo comando gera e compila a RNA definindo o seguinte:
  - Método de otimização ⇒ gradiente descendente;
  - Função de custo ⇒ função logística, também chamada de entropia cruzada binária;
  - Métrica ⇒ exatidão (veremos isso com detalhes nas próximas aulas).
- A compilação de uma RNA exige escolher inúmeros hiperparâmetros para o método de otimização calcular os parâmetros da RNA ⇒ nesse exemplo usamos valores padrão para esses hiperparâmetros.

➢ O Keras usa como princípio tornar a cozias simples, mas ao mesmo tempo permite ao usuário controlar tudo o que for necessário ⇒ se precisarmos e quisermos podemos configurar completamente o otimizador.

➢ Para definir os parâmetros do otimizador podemos realizar a compilação, por exemplo, com os seguintes comandos:

```
from tensorflow.keras import optimizers

Sgd = optimizers.SGD(lr=0.02)
rna.compile(loss='binary_crossentropy', metrics=['accuracy'],
            optimizer=sgd)
```

- A única diferença desse exemplo para o anterior é que definimos a taxa de aprendizado como sendo igual a 0,02.
- Existem muitos outros hiperparâmetros que podemos definir para o otimizador, veremos alguns desses posteriormente.

➢ Podemos incluir mais de uma métrica. Por exemplo, para calcular além da exatidão o erro absoluto médio, o método compile é usado da seguinte forma:

```
Sgd = optimizers.SGD(lr=0.02)
rna.compile(loss='binary_crossentropy',
            metrics=['accuracy', 'mae'], optimizer=sgd)
```

- Observe que o termo `mae` no código acima representa as primeiras letras do termo em inglês “mean absolute error”.

# 7. Treinamento

- O treinamento de uma RNA é realizado com o método `fit` ⇒ para treinar uma RNA usando 10 épocas o comando usado é o seguinte:

```
rna.fit(x_train, y_train, epochs=10)
```

  - O método `fit` realiza o treinamento da RNA com os exemplos de treinamento compostos pelos dados de entrada, `x_train`, e os dados de saída, `y_train`.
  - Existem diversos argumentos para o método `fit`, que veremos posteriormente.
  - Todas as opções para o método `fit` podem ser vistas com detalhes na documentação do Keras.

## Salvar o processo de treinamento

- Salvar o processo de treinamento permite fazer gráficos da função de custo e da métrica, permitindo uma análise mais detalhada do processo. Para isso usamos:

```
history = rna.fit(x_train, y_train, epochs=10)
```

  Nesse comando de treinamento os valores da função de custo e da métrica em função das épocas são salvos no objeto `history`.

- O objeto `history` contém um dicionário com os valores da função de custo e das métricas em função do número de épocas, que podem ser acessadas fazendo-se:

```
history_dict = history.history
history_dict.keys()
```

```
dict_keys(['loss', 'acc', 'mean_absolute_error'])
```

  - O comando `history_dict.keys` apresenta o conteúdo do dicionário salvo durante o treinamento, que no caso são: função de custo (`loss`), métrica exatidão (`acc`) e métrica erro absoluto médio (`mean_absolute_error`).

- Após recuperar e verificar o conteúdo do objeto `history`, podemos colocar os valores da função de custo e das métricas em vetores, usando os seguintes comandos:

```
# Salva custo,métrica e épocas em vetores
custo = history_dict['loss']
exatidao = history_dict['acc']
erro = history_dict['mean_absolute_error']

# Cria vetor de épocas
epocas = range(1, len(custo) + 1)
```

- O último comando cria um vetor com o número de épocas para podermos fazer os gráficos.

## Gráficos do processo de treinamento

➢ Os gráficos da função de custo e das métrica em função das épocas podem ser realizados usando as funções da bilbioteca Python Matplotlib da seguinte forma:

```
import matplotlib.pyplot as plt

# Gráfico do custo
plt.plot(epocas, custo, 'b')
plt.title('Valor da função de custo')
plt.xlabel('Épocas')
plt.ylabel('Custo')
plt.show()

# Gráfico da exatidão
plt.plot(epocas, exatidao, 'b')
plt.title('Valor da exatidão')
plt.xlabel('Épocas')
plt.ylabel('Exatidão')
plt.show()

# Gráfico do erro absoluto médio
plt.plot(epocas, erro, 'b')
plt.title('Valor do erro absoluto médio')
plt.xlabel('Épocas')
plt.ylabel('Erro médio')
plt.show()
```

- No comando `plot` o símbolo `'b'` significa que queremos uma curva contínua azul (`b` de "blue");

- Observe que o gráfico somente é mostrado após o comando `show`.

# 8. Avaliação do desempenho da RNA

- Após treinar a RNA é importante avaliar o seu desempenho com dados que não foram utilizados no treinamento.

- A avaliação da RNA com o conjunto de dados de teste pode ser feita usando o método `evaluate` da seguinte forma:

```
custo_e_metricas = rna.evaluate(x_test, y_test)
print(custo_e_metricas)
```

```
10000/10000 [==============================] - 1s 82us/sample - loss:
                        0.3859 - acc: 0.8797 - mean_absolute_error: 4.4200
[0.3858579118728638, 0.8797, 4.419997]
```

A saída do método `evaluate` é uma lista com os valores da função de custo e das métricas para os exemplos do conjunto de teste.

- A avaliação da RNA também pode ser realizada calculando-se as saídas previstas dos exemplos do conjunto de teste usando o método `predict`, como segue:

```
import numpy as np

# Cálculo das classes previstas
y_prev = rna.predict(x_test)
classes = np.round(y_prev)

# Gráfico das classes reais e previstas
plt.plot(y_test, 'ro', label='Classes reais')
plt.plot(classes, 'bo', label='Classes previstas')
plt.title('Classes reais e previstas')
plt.xlabel('Exemplos')
plt.ylabel('Classes')
plt.legend()
plt.show
```

  - A saída do método `predict` é uma lista com os valores das saídas calculadas para cada exemplo do conjunto de teste.

  - Para um problema de classificação binária, a última camada da RNA é uma função sigmóide cujas as saídas calculadas são valores entre 0 e 1.

  - Para determinar a classe prevista podemos simplesmente adotar, como feito no código anterior que:
    - Se y_prev >= 0,5, então, classe = 1;
    - Se y_prev < 0.5, então, classe = 0;

  - Os sete últimos comandos do código anterior fazem o gráfico das classes reais e previstas para cada exemplo do conjunto de teste. Alguns detalhes desses comandos são os seguintes:

- Nos comandos `plot` os símbolos `'ro'` e `'bo'` significam que queremos marcar essas curvas com círculos vermelhos e azuis respectivamente;
- Observe que podemos colocar vários comandos `plot` em seqüência para fazermos várias curvas no mesmo gráfico e o gráfico somente é mostrado após o comando `show`;
- Para incluir uma legenda incluímos a opção de `label` nos comandos `plot` e após esses comandos incluímos o comando `legend()` para mostrar a legenda no gráfico.

# 9. Salvar uma RNA treinada

- Após desenvolvermos uma RNA podemos salvá-la em um arquivo para podemos utilizá-la posteriormente.
- Podemos salvar somente a arquitetura da rede, ou somente seus parâmetros, ou ainda a arquitetura e seus parâmetros.

## Salvando uma RNA completa

- Para salvar uma RNA desenvolvida com o Keras usamos o método `save(file_path_and_name)` ⇒ nesse caso a RNA é salva em um arquivo no formato hdf5 que contém:
  - A arquitetura da RNA;
  - Os parâmetros da RNA;
  - Os parâmetros do otimizador usado para treinar a RNA;
  - O estado do otimizador para permitir continuar o treinamento exatamente de onde parou.
- Tendo uma RNA salva em um arquivo do formato HDF5 podemos usar o método `load_model(file_path_and_name)` para restabelecer a RNA da mesma forma que era quando foi salva.
- Um exemplo de uso desses dois métodos é o seguinte:

```
# Importa função do Keras para salvar modelos e biblioteca para
manipular arquivos no formato HDF5
from tensorflow.keras.models import load_model
import h5py

# Salva RNA e cria um arquivo formato HDF5 de nome RNA.h5 no
diretório /Meu_dretorio/
rna.save('/Meu_diretorio/RNA.h5')
# RNA.h5 é o nome do arquivo onde a rna é salva

# Apaga a rna existente
del rna

# Recupera a rna do arquivo RNA.h5
rna = load_model('/Meu_diretorio/RNA.h5')
```

- Para esses métodos funcionarem é necessário que a biblioteca h5py do Python esteja instalada em seu computador.
- Não precisa incluir o nome do diretório antes do arquivo que deseja salva a RNA, pois ela será salva no mesmo diretório onde está o seu notebook.

## Salvando os parâmetros de uma RNA

➢ Se desejarmos salvar somente os parâmetros de uma RNA usamos o método `save_weigths(file_path_and_name)` da seguinte forma:

```
rna.save_weigths('/Meu_diretorio/RNA_parametros.h5')
```

➢ Para carregar esses parâmetros em outra RNA (rna2) com mesma arquitetura usada para obter esses parâmetros, usamos o seguinte método:

```
rna2.load_weigths('/Meu_diretorio/RNA_parametros.h5')
```

Observe que a `rna2` deve ser criada com a mesma arquitetura da rede que gerou os parâmetros.

➢ Existem situações onde queremos desenvolver uma nova RNA (`rna_nova`) utilizando como base outra RNA (`rna_base`) já treinada. Porém, queremos aproveitar somente os parâmetros de algumas camadas da `rna_base` na `rna_nova` ⇒ isso é possível de fazer usando os métodos `save_weigths` e `load_weigths`, com pequenas alterações (veremos isso posteriormente).

# 10. Exemplo de classificação binária com RNA deep-learning

- Nesse exemplo queremos determinar se uma imagem mostra ou não um gato.
- O problema consiste em: dada uma imagem, a RNA avalia a probabilidade de existir ou não um gato na imagem.
- O objetivo desse problema é treinar uma RNA que recebe como entrada uma imagem e calcula a probabilidade da imagem mostrar ou não um gato.
- Uma imagem colorida no formato RGB é armazenada no computador na forma de 3 matrizes de números inteiros, variando de 0 a 255, correspondendo às cores vermelho (R), verde (G) e azul (B) presentes em cada ponto (pixel) da imagem (Figura 2).
- Cada pixel da imagem possui 3 valores, um para cada cor RGB ⇒ os valores de cada pixel fornecem as intensidades das cores correspondentes a essa posição da imagem.

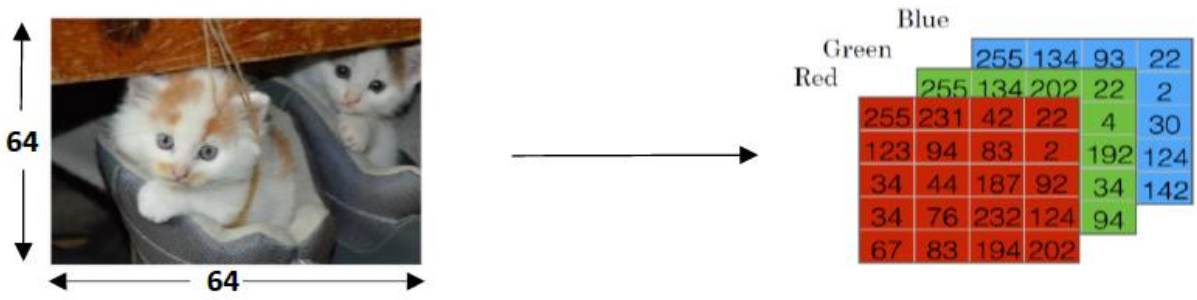



**Figura 2.** Exemplo de uma imagem de gato (Andrew Ng, deeplearning.ai).

## Conjunto de dados de treinamento

- Nesse exemplo usaremos imagens com resolução de 64x64 pixels, ou seja, cada uma das 3 matrizes que representam a imagem tem 64x64 elementos.
- Para criar o vetor de entrada, as matrizes com as intensidades das cores de cada pixel são redimensionadas criando um vetor coluna.
- A dimensão de cada vetor de entrada, $\mathbf{x}$, é ⇒ $n_x$ = 64x64x3 = 12.288, ou seja:

$$\mathbf{x} = \begin{bmatrix} 255 \\ 231 \\ 42 \\ \vdots \\ 255 \\ 134 \\ 202 \\ \vdots \\ 255 \\ 134 \\ 93 \\ \vdots \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \text{Vermelho (R)} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \text{Verde (G)} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} \text{Azul (B)} \end{matrix}$$

- Para facilitar o treinamento da RNA as entradas são normalizadas ⇒ no caso de imagens a forma de normalização mais utilizada é fazer com que todos os elementos dos vetores de entrada fiquem no intervalo de 0 a 1. Assim, para normalizar os vetores de entrada basta dividi-los pela intensidade máxima de cor, que no caso de imagens de 8 bits para cada cor é 256, ou seja:

$$\mathbf{x} = \frac{1}{256}\mathbf{x} \qquad (1)$$

- Um elemento do conjunto de treinamento consiste da imagem transformada no vetor **x**, de dimensão (12.288), e a sua saída correspondente, $y$, que classifica a imagem como mostrando ou não um gato:
  - $y = 0 \Rightarrow$ imagem não mostra gato;
  - $y = 1 \Rightarrow$ imagem mostra gato.

## Definição do problema

- Matematicamente podemos definir esse problema da seguinte forma:

  Dado o vetor **x** ⇒ calcular a probabilidade $\hat{y} = P(y = 1/\mathbf{x})$, onde $0 \leq \hat{y} \leq 1$.

- No treinamento da RNA são usados conjuntos de dados de treinamento e de teste:
  - Numero de exemplos de treinamento ⇒ $m$;
  - Número de exemplos de teste ⇒ $m_{teste}$.

- Conjunto de exemplos de treinamento:
  - Cada exemplo ⇒ $(\mathbf{x}, y)$, $\mathbf{x} \in \mathrm{R}^{nx}$ e $y \in \{0, 1\}$, ($n_x$ = 12.288);
  - Os $m$ exemplos ⇒ $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$;

- As *m* entradas são agrupadas em uma matriz de entradas:

$$\mathbf{X} = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}_{(n_x, m)} \Rightarrow \mathbf{X} \in \mathrm{R}^{(nx,\, m)}, \text{ dimensão } (n_x, m)$$

- As *m* saídas são agrupadas em um vetor de saídas:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}_{(1,m)} \Rightarrow \mathbf{y} \in \mathrm{R}^{(1x m)}, \text{ dimensão } (1, m)$$

- Como o Keras espera que o primeiro eixo seja o dos exemplos, então, temos que transpor a matriz de entradas e o vetor de saídas, assim:

$$\mathbf{X} = \mathbf{X}^t = \begin{bmatrix} \leftarrow & \mathbf{x}^{(1)} & \rightarrow \\ & \vdots & \\ \leftarrow & \mathbf{x}^{(m)} & \rightarrow \end{bmatrix}_{(m, n_x)} \Rightarrow \mathbf{X} \in \mathrm{R}^{(m,\, nx)}, \text{ dimensão } (m, n_x)$$

$$\mathbf{y} = \mathbf{y}^t = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}_{(m,1)} \Rightarrow \mathbf{y} \in \mathrm{R}^{(m,\, 1)}, \text{ dimensão } (m, 1)$$

- No próximo trabalho iremos implementar uma RNA deep learning usando o Keras para realizar essa tarefa de classificação binária e analisaremos a influência de alguns parâmetros, tais como, número de camadas, número de neurônios e taxa de aprendizado no desempenho da RNA.