DEEP LEARNING

UNIDADE III
REDES CONVOLUCIONAIS, RECORRENTES E ADVERSÁRIAS

Elaboração

Natasha Sophie Pereira

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

SUMÁRIO

REDES CONVOLUCIONAIS, RECORRENTES E ADVERSÁRIAS

UNIDADE III

CAPÍTULO 1

REDES NEURAIS CONVOLUCIONAIS

Um dos principais problemas com as Redes Neurais simples é que elas transformam todas as entradas em dados vetoriais, o que pode prejudicar a análise dos dados, caso o formato original seja uma matriz. Apresentadas inicialmente por LeCun (1989), as Redes Convolucionais, também conhecidas por Redes Neurais Convolucionais (CNN, do inglês *Convolutional Neural Networks*), são um tipo de Redes Neurais para o processamento de dados organizados em forma de grade, ou matriz, por exemplo, dados de séries temporais (grade unidimensional) e imagens (grade de *pixels* bidimensional ou tridimensional). Essas redes são muito eficientes para aplicações práticas, e receberam este nome visto que aplicam uma operação matemática chamada convolução, que é um tipo de operação linear específica, nos dados que estão analisando.

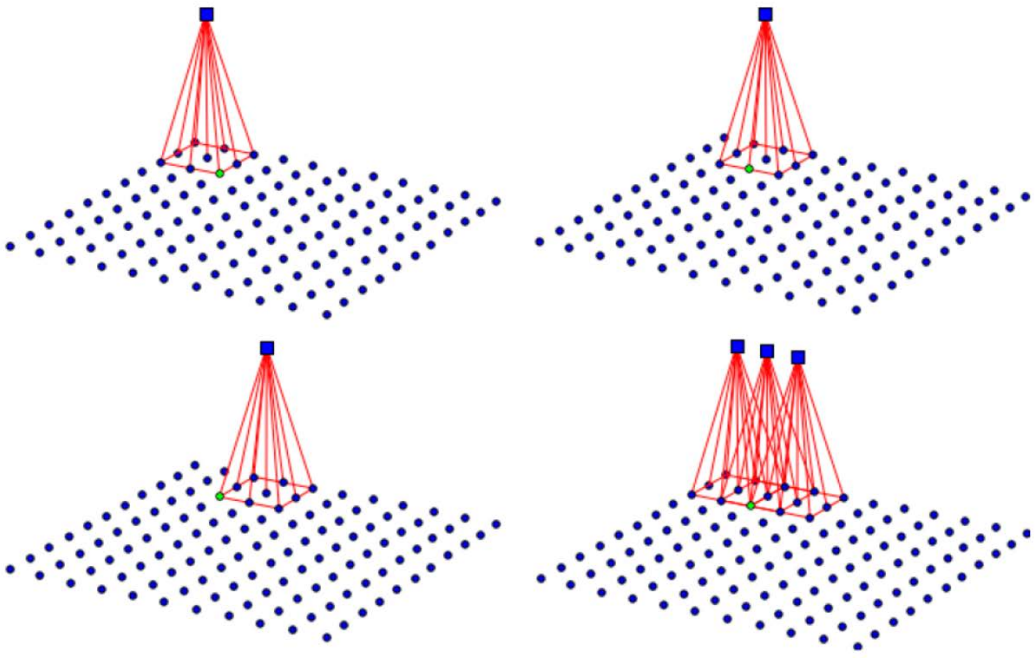
De acordo com GoodFellow, Bengio e Courville (2016), as Redes Convolucionais não passam de redes neurais que aplicam, em pelo menos uma de suas camadas, a convolução ao invés de uma multiplicação matricial comum. A convolução possui três argumentos, o primeiro é a entrada de dados, o segundo é o Kernel, e a saída é conhecida como mapa de características. Vasilev *et al.* (2019) apresentam que, em uma CNN, a rede neural não é completamente conectada (*fully-connected*), ou seja, os neurônios são conectados apenas com outros neurônios que correspondem a células próximas na matriz. Sendo assim, os neurônios são obrigados a receberem como entrada apenas células que são espacialmente próximas na matriz. Os autores também explicam que uma CNN compartilha alguns pesos por todos os neurônios de uma camada, ou seja, é um tipo de rede que utiliza o compartilhamento de parâmetros (*parameter sharing*). Essas duas características das CNNs ajudam na redução da quantidade de pesos na rede.

Camadas convolucionais

As camadas convolucionais são as mais importantes em uma CNN. Elas são formadas por um conjunto de filtros, também conhecidos como Kernels, e todos os filtros são aplicados em toda a extensão dos dados de entrada. Um kernel é definido por um conjunto de pesos que podem ser aprendidos.

O exemplo apresentado na Figura 16 mostra uma camada de entrada bidimensional de uma rede neural, onde cada neurônio representa a intensidade de cor de um *pixel*. Inicialmente o filtro, no caso do exemplo de tamanho 3x3 (mas poderia ser outro tamanho a sua escolha), é aplicado no primeiro conjunto de *pixels* da imagem, de modo que cada um dos nove neurônios será associado a um dos pesos do kernel, a saída do filtro será uma soma ponderada de suas entradas, também conhecida como ativação dos neurônios de entrada, que tem como objetivo destacar uma característica específica da entrada de dados. O conjunto de neurônios apresentados ao kernel é chamado de campo receptivo, e a saída do kernel representa o valor de ativação para os neurônios da próxima camada, de modo que o neurônio só será ativado se a característica representada por ele estiver presente naquela localização espacial da matriz. O filtro percorrerá toda a matriz de entrada, e a soma ponderada será calculada para cada novo conjunto de entrada. É bom reforçar que os valores dos pesos do kernel não se alteram enquanto ele percorre a matriz de entrada, o mesmo conjunto de pesos será utilizado para calcular o neurônio de ativação de toda a matriz.

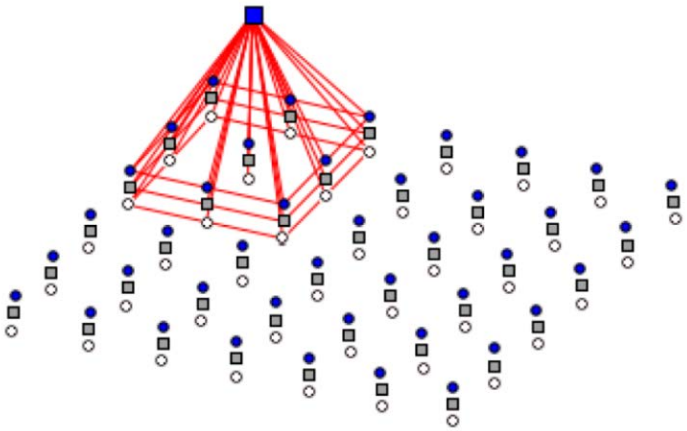
Figura 16. Percorrendo uma matriz bidimensional de entrada com o Kernel.



Fonte: Vasilev *et al.*, 2019, p. 96.

O compartilhamento de parâmetros é feito com o intuito de reduzir a quantidade de pesos utilizados e para destacar características importantes em partes específicas da matriz de entrada. O processo exemplificado com a Figura 16 deve ser repetido para todas as células da matriz de entrada. Ao final, será gerado um conjunto de neurônios organizados espacialmente, chamado de fatia de profundidade (*depth slice*) ou de mapa de características (*feature map*), que será apresentado para a próxima camada da rede. Em seguida, deverá ser aplicada uma função de ativação (a mais comum é a ReLU). É importante lembrar que em uma CNN o bias também será único para toda a entrada.

Figura 17. Percorrendo uma matriz tridimensional de entrada com o Kernel.



Fonte: Vasilev *et al.*, 2019, p. 96.

Para uma matriz com mais de duas dimensões, o processo é similar, basta separar as dimensões, de modo que cada dimensão é uma fatia da profundidade, e a combinação de todas as fatias é chamada de volume da entrada com profundidade igual à quantidade de fatias. Neste caso, um mesmo filtro será aplicado em todas as fatias. Ou seja, a ativação de um neurônio de saída será a soma ponderada do filtro aplicado em cada fatia, como pode ser observado na Figura 17.

A fim de perceber diversas características do conjunto de dados de entrada, é necessário aplicar diversos filtros por ele, de modo que cada filtro irá gerar um mapa de características que enfatiza certa característica. Dessa forma, a combinação dos mapas de saída é chamada de volume de saída, e tem profundidade igual à quantidade de filtros.

Para exemplificar, vamos ver o código apresentado por Vasilev *et al* (2019), que exemplifica uma operação convolucional ao aplicar alguns filtros a uma imagem. Inicialmente, é necessário importar a biblioteca numpy:

```
import numpy as np
```

Em seguida, será definida a função conv, que aplicará a convolução pela imagem. Essa função utiliza dois parâmetros: image para a imagem e filter para o filtro. Esta etapa será realizada em quatro passos: I) calcular do tamanho da imagem de saída, que depende do tamanho da imagem de entrada e do filtro, será chamado de im_c; II) passar o filtro por toda a imagem; III) verificar se algum valor gerado está fora do intervalo [0, 255], que é o intervalo de valores para tons de cinza em uma imagem, e caso algum valor esteja fora, corrigir; IV) mostrar a imagem de entrada e a de saída para comparação:

```
def conv(image, im_filter):
"""
  :param image: grayscale image as a 2-dimensional numpy
    array
  :param im_filter: 2-dimensional numpy array
"""

  # Tamanho da Entrada
  height = image.shape[0]
  width = image.shape[1]

  # Imagem de saída com tamanho reduzido
  im_c = np.zeros((height - len(im_filter) + 1,
    width - len(im_filter) + 1))

  # Passando por todas as linhas e colunas
  for row in range(len(im_c)):
    for col in range(len(im_c[0])):
      # Aplicando o filtro
      for i in range(len(im_filter)):
        for j in range(len(im_filter[0])):
            im_c[row, col] += image[row + i, col
                      + j] * im_filter[i][j]

  # Corrigindo os valores for a do intervalo entre [0,255]
  im_c[im_c > 255] = 255
  im_c[im_c < 0] = 0

  # Mostrando as imagens para comparação
```

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
plt.figure()
plt.imshow(image, cmap=cm.Greys_r)
plt.show()

plt.imshow(im_c, cmap=cm.Greys_r)
plt.show()
```

Com este código pronto, vamos baixar a imagem. O seguinte código baixa a imagem RGB solicitada para uma matriz numpy e a converte para tons de cinza:

```
import requests
from PIL import Image
from io import BytesIO

# Baixando a imagem
url = https://upload.wikimedia.org/wikipedia/commons/
  thumb/8/88/Commander_Eileen_Collins_-_GPN-2000-
  001177.jpg/382px-Commander_Eileen_Collins_-_GPN-
  2000-001177.jpg?download”
resp = requests.get(url)
image_rgb = np.asarray(Image.open(BytesIO(resp.content)
  ).convert(“RGB”))

# Convertendo para tons de cinza
image_grayscale = np.mean(image_rgb, axis=2,
  dtype=np.uint)
```

Com a imagem carregada no sistema, é possível aplicar-lhe diferentes filtros. No exemplo, serão utilizados o filtro blur de tamanho 10x10 e o Sobel para detectar contornos:

```
# Filtro blur
blur = np.full([10, 10], 1. / 100)
conv(image_grayscale, blur)

# Sobel para detectar contornos
sobel_x = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]
```

```
conv(image_grayscale, sobel_x)

sobel_y = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
conv(image_grayscale, sobel_y)
```

A partir deste código, serão geradas as imagens apresentadas na Figura 18.

Figura 18. Imagens geradas. Da esquerda para a direita, imagem original, com aplicação do filtro blur, com aplicação do Sobel horizontal, com aplicação do Sobel vertical.

Fonte: Vasilev *et al.*, 2019, p. 102.

Passo e Preenchimento das Camadas Convolucionais

Nem sempre os filtros passam por cada um dos *pixels* de uma imagem, eles podem pular alguns *pixels*, a esse parâmetro, dá-se o nome de passo, que deve ser constante por toda a imagem. Ao utilizar um passo maior do que 1 (quando o filtro passa por todos os neurônios) acontece a redução do tamanho do mapa de características gerado. Sendo assim, um mesmo neurônio da próxima camada será ativado por uma área maior da imagem de entrada, o que permite que características mais complexas sejam reconhecidas.

É comum gerar saídas menores que a entrada, porém, o ideal é que o tamanho das saídas seja compatível com o tamanho da entrada. Neste caso, é necessário preencher as bordas da imagem de entrada com zeros antes da operação de convolução.

Deconvolução

Da mesma forma que o *Backpropagation* funciona em uma rede neural totalmente conectada, ele funciona na convolução, ou seja, os mesmos neurônios determinarão os gradientes dos passos de volta ao fazer o *backpropagation*. Essa ação é conhecida como convolução transposta, ou deconvolução.

Algumas bibliotecas de *Deep Learning* como Pytorch, Keras e TensorFlow suportam os padrões 1D, 2D e 3D, e também, convoluções em escala profunda. Neste caso, tanto os operadores de entrada quanto os de saída são tensores.

Camadas de agrupamento

Uma camada de agrupamento (*Pooling*) divide a fatia de entrada em uma grade onde cada célula representa um campo receptivo de alguns neurônios, semelhantemente à camada de convolução. Ao gerar essa grade, uma operação de agrupamento é aplicada em cada uma das células. É importante ressaltar que as camadas de agrupamento não alteram o volume da profundidade, pois as operações de agrupamento são aplicadas em cada fatia de forma independente. Note que não existem pesos nas camadas de agrupamento. Existem diferentes tipos de camadas de agrupamento:

- **Max Pooling:** é a forma mais utilizada de agrupamento. Essa operação pega o neurônio com valor de ativação mais algo em cada campo receptivo, ou seja, em cada célula da grade, e propaga apenas este valor. Ao realizar o passo para trás, o gradiente é direcionado apenas ao neurônio que possui maior valor de ativação durante o passo para frente, os outros neurônios propagam zero.
- **Average Pooling:** neste tipo de agrupamento, a saída de cada campo receptivo é a média de todos os valores de ativação do campo.

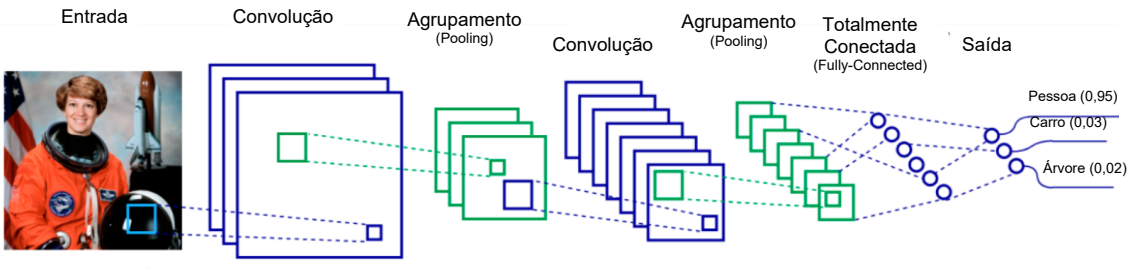
As camadas de agrupamento são definidas por dois parâmetros: I) passo, da mesma forma que as camadas convolucionais; II) tamanho do campo receptivo, que pode ser comparado com o tamanho do filtro, ou kernel, das camadas convolucionais. Apesar de as camadas de agrupamento serem muito utilizadas, algumas vezes é possível alcançar resultados similares ou melhores utilizando camadas convolucionais com passos maiores.

Arquitetura de uma CNN

De acordo com Vasilev *et al.* (2019), as Redes Neurais Convolucionais possuem algumas características básicas, como:

- Geralmente são alternadas uma ou mais camadas de convolução com uma ou mais camadas de agrupamento, como pode ser visto na Figura 19. Isso permite que as camadas convolucionais detectem características em todos os níveis do campo receptivo. À medida que a rede vai ficando mais profunda, os campos receptivos vão aumentando de tamanho, o que permite que a rede colete características mais complexas de regiões maiores da imagem de entrada.

Figura 19. Rede Neural Convolucional.



Fonte: Adaptada de Vasilev *et al.*, 2019, p. 110.

» As camadas convolucionais são utilizadas para extrair características dos dados de entrada. Quanto mais profunda é a camada, mais abstrata é a característica que ela consegue extrair, porém, não são compreensíveis pelo ser humano. Para amenizar este problema, são adicionadas uma ou mais camadas totalmente conectadas após a última camada convolucional ou de agrupamento.

» Quanto mais profundas são as camadas convolucionais, mais filtros ela utiliza. Nas camadas iniciais da rede, um detector de características analisa pequenos campos receptivos, por isso, pode detectar apenas um pequeno número de características, já as camadas mais profundas podem detectar características mais complexas.

Utilizando uma CNN para classificar dígitos escritos manualmente

Vamos voltar a trabalhar com o conjunto de dados MNIST, e seguir outro exemplo apresentado por Vasilev *et al.* (2019), que aplicam uma Rede Neural Convolucional para obter uma melhor acurácia do que utilizando apenas Keras.

Inicialmente, vamos importar as bibliotecas e inicializar o `random seed`:

```
# Para proliferação
from numpy.random import seed

seed(1)
from tensorflow import set_random_seed

set_random_seed(1)
```

Em seguida, vamos fazer mais algumas importações para as camadas de convolução e de agrupamento (*max pooling*):

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Flatten

from keras.utils import np_utils
```

Após a importação das bibliotecas e funções necessárias, vamos importar o conjunto de dados MNIST. Faremos o reajuste da entrada para fragmentos de tamanho 28x28:

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)

Y_train = np_utils.to_categorical(Y_train, 10)
Y_test = np_utils.to_categorical(Y_test, 10)
```

Agora é possível definir o modelo da rede, que será de duas camadas convolucionais, uma camada de agrupamento do tipo *max pooling* e duas camadas totalmente conectadas. Além disso, é necessário utilizar uma Flatten entre a camada de agrupamento e a camada totalmente conectada, pois essa espera receber uma entrada unidimensional e a saída da camada convolucional é tridimensional:

```
model = Sequential([
  # Primeira camada convolucional
  Convolution2D(filters=32,
    kernel_size=(3, 3),
    input_shape=(28, 28, 1)),
  Activation('relu'),

  # Segunda camada convolucional
  Convolution2D(filters=32,
    kernel_size=(3, 3)),
  Activation('relu'),

  # Camada de agrupamento (Max pooling)
```

```
    MaxPooling2D(pool_size=(2, 2)),

    # Transformando o tensor em unidimensional
    Flatten(),

    #Camada oculta totalmente conectada
    Dense(64),
    Activation('relu'),

    # Camada de saída
    Dense(10),
    Activation('softmax')])

print(model.summary())
```

O model.summary() do Keras é um método que apresenta um resumo de como a rede foi executada, conforme apresentado na Figura 20.

Figura 20. Saída da classificação de dígitos manuais usando CNN.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
activation_1 (Activation)    (None, 26, 26, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 24, 24, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 12, 12, 32)     0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4608)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                294976
_________________________________________________________________
activation_3 (Activation)    (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
_________________________________________________________________
activation_4 (Activation)    (None, 10)                0
=================================================================
Total params: 305,194
Trainable params: 305,194
Non-trainable params: 0
```

Fonte: Vasilev *et al.*, 2019, p. 113.

Em seguida, será definido o otimizador. Ao invés do Gradiente Descendente Estocástico (SGD), utilizaremos o ADADELTA, que automaticamente definirá uma taxa de aprendizado maior ou menor de forma inversamente proporcional ao gradiente:

```
model.compile(loss='categorical_crossentropy',
    metrics=['accuracy'], optimizer='adadelta')
```

Agora, com todo o código pronto, é possível treinar a rede. Faremos isso utilizando cinco interações:

```
model.fit(X_train, Y_train, batch_size=100, epochs=5,
    validation_split=0.1, verbose=1)
```

Por último, testaremos a rede:

```
score = model.evaluate(X_test, Y_test, verbose=1)
print('Test accuracy:', score[1])
```

Foi obtida uma acurácia de 98,5% utilizando este modelo, contra 96% de acurácia utilizando uma Rede Neural simples.

Regularização

Overfitting

O aumento desordenado do número de neurônios, assim como o aumento de camadas intermediárias, não assegura a generalização apropriada das redes *perceptrons* multicamadas (ou redes neurais *feedforward*).

Quando isso acontece, a saída das redes *perceptrons* multicamadas tende a memorização excessiva (*overfitting*). O problema do *overfitting* é justamente o baixo ajuste ao restante do problema, uma vez que ele fica superajustado ao subconjunto de teste.

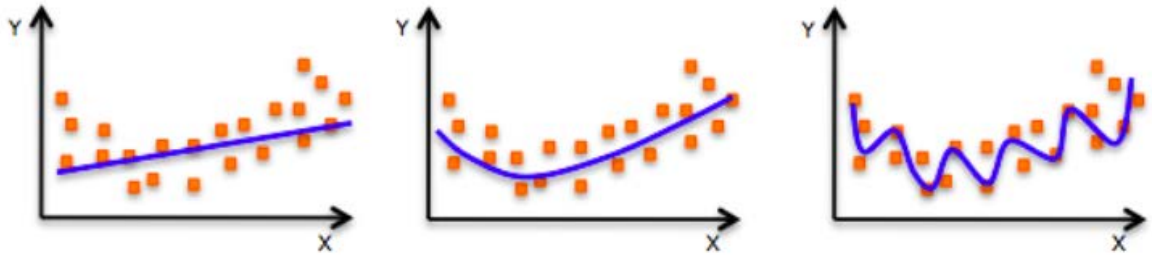
Underfitting

Refere-se a um padrão que não pode modelar os dados de treinamento nem generalizar para novos dados. Um modelo de aprendizagem de máquina com baixo perfil não é um modelo adequado pois terá um desempenho ruim nos dados de treinamento.

Regularização é qualquer tipo de modificação feita em um algoritmo de aprendizado que culmine na redução do erro de generalização, mas não no erro de treinamento.

Para entender o conceito de regularização, vamos observar a Figura 21.

Figura 21. *Underfitting, Just Right* e *Overfitting*



Fonte: Jain, 2018.

Observe que à medida que nos movemos para a direita nesta figura, nosso modelo tenta aprender muito bem os detalhes e o ruído dos dados de treinamento, o que acaba resultando em baixo desempenho nos dados não vistos.

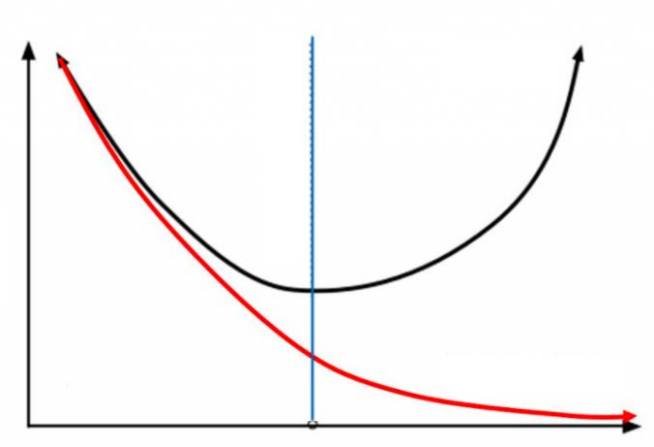
Em outras palavras, ao ir para a direita, a complexidade do modelo aumenta de tal forma que o erro de treinamento diminui, mas o erro de teste não. Isso é mostrado na Figura 22.

Sabemos que Redes Neurais são extremamente complexas. Isso as torna mais propensas ao *overfitting*. Ao utilizar conjuntos de redes neurais com diferentes configurações de modelo pode haver uma redução do *overfitting*. No entanto, isso requer um custo computacional adicional de treinamento e manutenção de múltiplos modelos. Aumentando, então, o erro de generalização devido ao *overfitting*.

Uma abordagem para reduzir o *overfitting* é ajustar todas as possíveis redes neurais no mesmo conjunto de dados e calcular a média das previsões de cada modelo. No entanto, isso não é viável na prática e pode ser aproximado usado uma pequena coleção de modelos diferentes, chamada de conjunto.

Para contornar o problema da generalização, a técnica de regularização foi implementada. Ela faz pequenas modificações no algoritmo de aprendizado, de modo que o modelo se generaliza melhor. A regularização, portanto, ajuda a reduzir o *overfitting* e algumas técnicas podem ser usadas para aplicar a regularização na aprendizagem profunda, são essas técnicas que veremos a seguir.

Figura 22. Gráfico do Treinamento *versus* Erro no Conjunto de Testes.



Fonte: Jain, 2018.

Existem diversos tipos de regularização, alguns deles são:

Regularização L1 e L2

L1 e L2 são os tipos mais comuns de regularização. Eles atualizam a função de custo adicionando outro termo, conhecido como termo de regularização.

Devido ao acréscimo desse termo de regularização, os valores das matrizes de peso diminuem, pois pressupõe-se que uma rede neural com matrizes de menor peso leve a modelos mais simples. Nesse caso, também reduzirá o *overfitting* em grande parte.

O termo de regularização difere em L1 e L2. Em L2 temos:

$$\text{Função de Custo} = \text{Perda} + \frac{\lambda}{2m} \bullet \sum \|w\|^2$$

λ é o parâmetro de regularização. Ele é um hiperparâmetro cujo valor é otimizado para melhores resultados. A regularização de L2 também é conhecida como *decaimento de peso*, pois força os pesos a decair em direção a zero (mas não exatamente a zero).

Em L1 temos:

$$\text{Função de Custo} = \text{Perda} + \frac{\lambda}{2m} \bullet \sum \|w\|$$

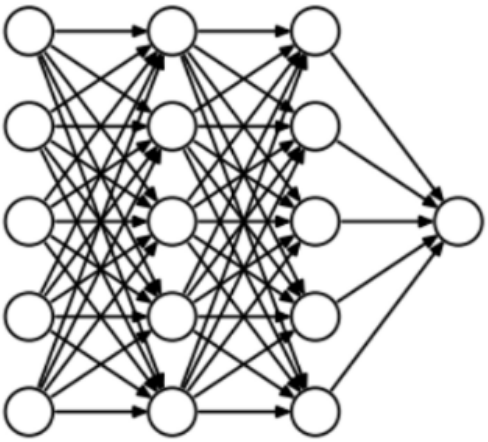
Em L1 há uma penalização do valor absoluto dos pesos. Ao contrário de L2, os pesos podem ser reduzidos a zero aqui. Por isso, é muito útil quando estamos tentando compactar nosso modelo. Caso contrário, geralmente iremos preferir a L2 sobre ele.

Dropout

Este é um dos tipos mais interessantes das técnicas de regularização. Produz também resultados muito bons e é, consequentemente, a técnica de regularização mais utilizada no campo do *deep learning*.

Para melhor compreender o *dropout*, observe a Figura 23 que apresenta como exemplo uma estrutura de rede neural.

Figura 23. Estrutura de uma rede neural.

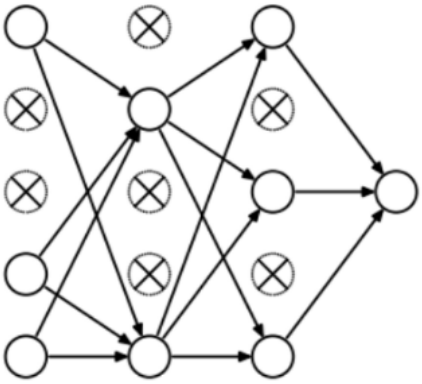


Fonte: Jain, 2018.

Então, como o *dropout* funciona nessa rede? A cada iteração, ele seleciona aleatoriamente alguns nós e remove junto todas as suas conexões de entrada e saída, conforme apresentado na Figura 24.

Portanto, a cada iteração, temos um conjunto diferente de nós e isso resulta num conjunto de diferentes saídas. Essa probabilidade de escolher quantos nós devem ser descartados quem decide é o hiperparâmetro da função *dropout*.

Figura 24. Funcionamento do *dropout*.



Fonte: Jain, 2018.

Dropout, portanto, é um método de regularização que aproxima a formação de um grande número de redes neurais com diferentes arquiteturas em paralelo. Durante o treinamento, algumas camadas de saída são aleatoriamente ignoradas ou até mesmo descartadas. Isso faz com que a camada pareça com uma camada com um número diferente de nós e conectividade com a camada anterior. De fato, cada atualização para uma camada durante o treinamento é realizada com uma visão diferente da camada configurada.

O *dropout* pode tornar o processo de treinamento barulhento, forçando os nós dentro de uma camada a assumirem mais ou menos entradas probabilisticamente. Esse conceito sugere que, talvez, situações de desistências em que camadas de rede se readaptem para corrigir erros de camadas anteriores tornem esse modelo mais robusto.

Dropout simula uma ativação esparsa de uma determinada camada, o que, curiosamente, por sua vez, incentiva a rede a realmente aprender uma representação esparsa como um efeito colateral. Como tal, pode ser utilizado como alternativa à regularização de atividades para estimular representações esparsas em modelos de codificadores automáticos. Como as saídas de uma camada sob o *dropout* são subamostras aleatórias, elas reduzem a capacidade ou diminuem a rede durante o treinamento.

Aumento de Dados

Quando o conjunto de treinamento é muito pequeno, a rede pode ficar muito especializada apenas neste conjunto, é o que chamamos de *overfitting*. A técnica de aumento de dados é uma das técnicas de regularização mais eficientes, pois ela busca resolver o problema de *overfitting* aumentando o conjunto de testes artificialmente. É possível aumentar a quantidade de dados de treinamento através da rotação das imagens, inversão horizontal ou vertical das imagens, aumento ou redução do tamanho, corte, inclinação, ajuste de contraste ou brilho.

Normalização em lotes

A normalização de lotes (*Batches*) permite a aplicação de um processamento de dados, semelhante à pontuação-padrão, nas camadas ocultas da rede. Essa técnica normaliza as saídas de cada camada oculta em pequenos lotes (minibatch), de modo que mantém seu valor médio de ativação próximo de zero e seu desvio-padrão próximo de um. Pode ser utilizada tanto em camadas convolucionais quanto em camadas totalmente conectadas. Uma rede que utiliza técnicas de normalização em lotes é treinada mais rapidamente e pode utilizar taxas de aprendizado mais altas.

Utilizando uma CNN para classificar imagens de objetos

Vamos voltar a trabalhar com o conjunto de dados CIFAR-10, e seguir outro exemplo apresentado por Vasilev *et al.* (2019), que aplicam uma Rede Neural Convolucional para obter uma melhor acurácia na classificação de imagens de objetos do que utilizando apenas uma rede totalmente conectada.

Inicialmente, vamos fazer as importações:

```
import keras
from keras.datasets import cifar10
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Dense, Dropout, Activation,
    Flatten, BatchNormalization
from keras.models import Sequential
from keras.preprocessing.image import ImageDataGenerator
```

O tamanho do pequeno lote (*minibach*) deve ser definido:

```
batch_size = 50
```

Em seguida, podemos importar o conjunto de dados CIFAR-10 e normalizar os dados dividindo-os por 255, que é o maior valor de intensidade de um *pixel*:

```
(X_train, Y_train), (X_test, Y_test) = cifar10.load_data()

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

Y_train = keras.utils.to_categorical(Y_train, 10)
Y_test = keras.utils.to_categorical(Y_test, 10)
```

Vamos baixar as imagens e definir as formas de aumento da quantidade. Para aumentar a quantidade, vamos precisar da classe ImageDataGenerator. Vamos permitir que essa classe gere imagens rotacionadas em até 90 graus, invertidas horizontalmente, e deslocadas horizontal e verticalmente. Os dados de treinamento serão padronizados (featurewise_center e featurewise_std_normalization). Como a média e o desvio-padrão são calculados em todo o conjunto de dados, é necessário chamar o método data_generator.fit(X_train). Para finalizar, é necessário aplicar a padronização de

treinamento no conjunto de teste, pois o método ImageDataGenerator gerará imagens modificadas durante o treinamento da rede:

```
data_generator = ImageDataGenerator(rotation_range=90,
    width_shift_range=0.1,
    height_shift_range=0.1,
    featurewise_center=True,
    featurewise_std_normalization=True,
    horizontal_flip=True)

data_generator.fit(X_train)

# Padronizando o conjunto de teste
for i in range(len(X_test)):
  X_test[i] = data_generator.standardize(X_test[i])
```

O próximo passo é definir a rede. Ela terá três blocos de duas camadas convolucionais (Filtros de tamanho 3x3) e uma camada de agrupamento do tipo max pooling; aplicará a normalização em lotes após cada camada convolucional. A função de ativação utilizada será a Unidade Linear Exponencial (ELU, do inglês *Exponential Linear Unit*); haverá apenas uma camada totalmente conectada após a última camada de agrupamento; será feito o preenchimento das saídas para que possuam as mesmas dimensões da entrada (padding='same'):

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
    input_shape=X_train.shape[1:]))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(Conv2D(32, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('elu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

model.add(Conv2D(128, (3, 3)))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(Conv2D(128, (3, 3)))
model.add(BatchNormalization())
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

O otimizador utilizado será Adam:

```
model.compile(loss='categorical_crossentropy',
    optimizer='adam', metrics=['accuracy'])
```

Agora, com o código pronto, é possível treinar a rede. Como existe a necessidade de aumentar a quantidade de dados, vamos utilizar o método model.fit_generator. O gerador de dados é ImageDataGenerator. O conjunto de teste será usado para validação através de validation_data, de modo a saber a performance em cada interação:

```
model.fit_generator(
  generator=data_generator.flow(x=X_train,
  y=Y_train,
  batch_size=batch_size),
  steps_per_epoch=len(X_train) // batch_size,
  epochs=100,
  validation_data=(X_test, Y_test),
  workers=4)
```

O valor da acurácia do modelo variará de acordo com a quantidade de interações (*epochs*) da seguinte forma:

- 3 interações: Acurácia de 47%.
- 5 interações: Acurácia de 59%.
- 100 interações: Acurácia de 80%.

CAPÍTULO 2

REDES NEURAIS RECORRENTES

De acordo com Goodfellow, Bengio e Courville (2016), as Redes Neurais Recorrentes (RNN, do inglês *Recurrent neural Networks*) são específicas para processar dados sequenciais de tamanho variável. Patterson e Gibson (2017) afirmam que as RNN são similares às redes neurais *Feedforward*, porém, o que as diferencia é a habilidade que as RNN têm de enviar informações através das camadas. Os autores ainda afirmam que antes era muito complexo treinar esse tipo de rede, porém, os avanços nas pesquisas têm tornado sua utilização mais fácil.

As Redes Neurais Recorrentes modelam individualmente cada vetor em uma sequência de vetores de entrada, o que permite que ela guarde seu estado, enquanto modela cada vetor na janela de entrada.

As redes neurais recorrentes recebem este nome, pois aplicam, de forma recorrente, a mesma função sobre uma sequência de valores.

$$s_t = f(s_{t-1}, x_t)$$

Onde f é uma função diferencial, s_t é um vetor de valores conhecido como estado interno da rede na interação t, e x_t é a entrada da rede na interação t. Ao contrário das redes regulares, nas RNNs o estado atual de uma interação depende da entrada atual e do estado da interação anterior s_{t-1}. Essa relação recorrente permite acompanhar a evolução dos estados da rede a cada interação sobre a sequência de dados.

Uma das características das RNNs é modelar a dimensão temporal. Ou seja, os dados de entrada ou saída de uma rede neural recorrente são vetores que possuem dependência temporal entre eles, ou seja, a sequência com que são apresentados à rede é relevante para a solução do problema. As RNNs podem ter *loops* nas conexões, permitindo modelar a acurácia do ganho de comportamento ao longo do tempo em séries temporais como conversas, áudio, texto etc. Nestes casos, os dados estão relacionados com a ordem e o contexto em que são apresentados, de modo que os valores apresentados por último dependem dos anteriores.

As redes neurais recorrentes possuem três conjuntos de parâmetros: I) U, transforma a entrada x_t para o estado s_t; II) W, transforma o estado prévio s_{t-1} para o estado atual s_t; V, mapeia o estado interno recém-computado s_t para a saída y_t. Os três parâmetros

aplicam transformações lineares sobre suas entradas. Dessa forma, o estado atual e a saída da rede podem ser definidos, respectivamente, como:

$$S_t = f(s_{t-1} * W + x_t * U)$$

$$y_t = s_t * V$$

Onde f é a função de ativação não linear (como tanh, sigmoid ou ReLU).

A função acima descreve a RNN para uma rede de uma camada apenas. Para transformá-la em multicamadas, basta empilhar camadas de RNNs. Assim, o estado da célula s_t^l de uma célula no nível l no tempo t receberá a saída y_t^{l-1} da célula no nível l – 1 e estado prévio s_{t-1}^l da célula de mesmo nível l como entrada:

$$s_t^l = f(s_{t-1}^l, y_t^{l-1})$$

De acordo com Vasilev *et al.* (2019), as redes RNNs não têm limite de tamanho para os dados de entrada a serem processados, o que aumenta as possibilidades do que pode ser computado por ela, por exemplo:

- Um para Um: processamento não sequencial, como redes neurais *Feedforward* ou redes neurais convolucionais.
- Um para Muitos: gera como saída uma sequência baseada em uma única entrada. Exemplo: Gerar legendas para imagens (VINYALS *et al.*, 2015).
- Muitos para Um: gera como saída um único resultado baseado em uma sequência de entrada. Exemplo: Classificação de um sentimento a partir de um texto.
- Muitos para Muitos (Indiretamente): uma sequência de entrada é codificada em um vetor de estados, em seguida, o vetor de estados é codificado em uma nova sequência. Exemplo: Tradução de texto (CHO *et al.*, 2014; SUTSKEVER; VINYALS; LE, 2014)
- Muitos para Muitos (Diretamente): gera uma saída para cada entrada. Exemplo: Reconhecimento de fala.

Implementação e treinamento

A fim de entender o funcionamento das Redes Neurais Recorrentes, vamos pegar o exemplo apresentado por Vasilev *et al.* (2019), que é contar a quantidade de vezes que o número 1 aparece em uma sequência de valores binários. É um exemplo de RNN "Muitos para Um". A implementação será feita utilizando Python e NumPy, não será necessária

a importação de bibliotecas de *Deep Learning*, visto que é um exemplo simples. Veja a sequência de entrada e a correspondente saída:

```
In: (0, 0, 0, 0, 1, 0, 1, 0, 1, 0) Out: 3
```

Essa rede terá apenas dois parâmetros, um peso de entrada U e um peso recorrente W. O peso de saída V será configurado como 1, para que a saída y seja o último estado.

Inicialmente, vamos importar o numpy e definir o conjunto de dados de treinamento x e o conjunto de rótulos y. x é bidimensional, visto que a primeira dimensão representa cada subconjunto da amostra. Como nosso conjunto de dados é pequeno, não há necessidade de fazer a divisão em subconjuntos menores:

```
import numpy as np


# A primeira dimensão representa o subconjunto
x = np.array([[0, 0, 0, 0, 1, 0, 1, 0, 1, 0]])


y = np.array([3])
```

A relação de recorrência desta rede é um modelo linear definido por $s_t = s_{t-1} * W + x_t * U$ que pode ser implementada como:

```
def step(s, x, U, W):
  return x * U + s * W
```

Os valores de estado, s_t, e pesos, W e U, são valores escalares simples, neste caso, basta somar as entradas no decorrer da sequência. Então, se definirmos U = 1, receberemos o valor total de cada entrada. Se definirmos W = 1, os valores acumulados não decairão. Sendo assim, para este exemplo, conseguiremos a saída 3, que é a desejada.

Backpropagation ao longo do tempo

Backpropagation ao longo do tempo é um algoritmo tipicamente utilizado para treinar redes recursivas. A diferença entre essa técnica e um *Backpropagation* normal é que uma RNN é desdobrada ao longo do tempo algumas vezes. Sendo assim, após o desdobramento de toda a rede, teremos um modelo similar a uma rede *Feedforward* com multicamadas. Ou seja, cada camada oculta em uma rede *Feedforward* pode ser comparada com uma interação da rede recursiva, porém, com várias entradas para cada camada, o estado prévio s_{t-1} e a entrada atual x_t, os parâmetros U e W serão comuns para todas as camadas ocultas.

O passo adiante (*forward pass*) desdobra a RNN ao longo da sequência e constrói uma pilha de estados a cada interação. A seguir, veja a implementação do *forward pass*, que retornará a ativação s para cada interação recorrente e cada valor do conjunto:

```
def forward(x, U, W):
  # Número de amostras do subconjunto
  number_of_samples = len(x)

  # Tamanho de cada amostra
  sequence_length = len(x[0])

  # Inicialize o estado de ativação para cada amostra da
  # sequência
  s = np.zeros((number_of_samples, sequence_length + 1))

  # Atualize os estados no decorrer da sequencia
  for t in range(0, sequence_length):
    s[:, t + 1] = step(s[:, t], x[:, t], U, W) # step
      Function

  return s
```

Após definir o passo adiante e a função de custo, é possível definir, também, o gradiente de propagação de volta (*backward*) utilizando *backpropagation*. Como os pesos W e U são constantes em todas as camadas, acumularemos o erro derivado de cada interação recorrente e, ao final, atualizaremos os pesos com o valor acumulado. É necessário obter o gradiente da saída com relação à função de custo, e então, propagá-lo pela pilha de atividades realizadas durante os passos adiante. Os passos para trás retiram as atividades da pilha para acumular seus erros derivados a cada interação. A seguir, veja a implementação do *backward pass*:

Inicialmente, os gradientes para U e W estão acumulados em gU e gW, respectivamente:

```
def backward(x, s, y, W):
  sequence_length = len(x[0])

  # A saída da rede é apenas a última ativação da sequência
  s_t = s[:, -1]
```

```
    # Calcule o gradiente de saída w.r.t. Função de custo
    # MSE do estado final.
    gS = 2 * (s_t - y)

    # Defina as acumulações gradientes em 0
    gU, gW = 0, 0

    # Retroceda acumulando os gradientes
    for k in range(sequence_length, 0, -1):
      # Calcule os gradientes dos parâmetros e acumule os
      # resultados
      gU += np.sum(gS * x[:, k - 1])
      gW += np.sum(gS * s[:, k - 1])

    # Calcule o gradiente de saída para a camada anterior
    gS = gS * W

    return gU, gW
```

A partir de agora, é possível utilizar o Gradiente Descendente para otimizar a rede. Vamos utilizar o erro quadrático médio:

```
def train(x, y, epochs, learning_rate=0.0005):
  """Train the network"""

  # Defina os parâmetros iniciais
  weights = (-2, 0) # (U, W)

  # Acumule as perdas e seus respectivos pesos
  losses = list()
  weights_u = list()
  weights_w = list()

  # Execute o gradiente descendente interativo
  for i in range(epochs):
    # Execute os passos adiante e para trás para oobter
    # os gradientes
```

```
        s = forward(x, weights[0], weights[1])

        # Calcule a função de custo MSE
        loss = (y[0] - s[-1, -1]) ** 2

        # Armazene os valores de perda e pesos para mostrar
        # depois
        losses.append(loss)

        weights_u.append(weights[0])
        weights_w.append(weights[1])

        gradients = backward(x, s, y, weights[1])
        # Atualize cada parâmetro 'p' para p = p-(gradiente
        # * taxa_aprendizado)
        # 'gp' é o gradiende do parâmetro 'p'
        weights = tuple((p - gp * learning_rate) for p, gp
          in zip(weights, gradients))

    print(weights)
    return np.array(losses), np.array(weights_u),
      np.array(weights_w)
```

Em seguida, implementaremos a função `plot_training`, que exibe os pesos e perda:

```
def plot_training(losses, weights_u, weights_w):
    import matplotlib.pyplot as plt

    # Remove os valores nan e inf
    losses = losses[~np.isnan(losses)][:-1]
    weights_u = weights_u[~np.isnan(weights_u)][:-1]
    weights_w = weights_w[~np.isnan(weights_w)][:-1]

    # Mostra os pesos U e W
    fig, ax1 = plt.subplots(figsize=(5, 3.4))

    ax1.set_ylim(-3, 2)
    ax1.set_xlabel('Ciclos')
```

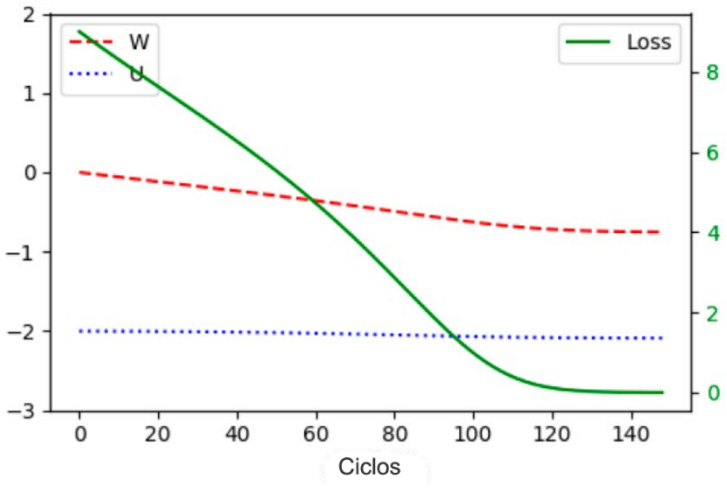
```
ax1.plot(weights_w, label='W', color='red',
  linestyle='-')
ax1.plot(weights_u, label='U', color='blue',
  linestyle=':')
ax1.legend(loc='upper left')

# Instancia um novo eixo que compartilha o mesmo eixo x
# Mostra a perda no Segundo eixo
ax2 = ax1.twinx()

# Retire os comentários para mostrar os gradientes
ax2.set_ylim(-1, 9)
ax2.plot(losses, label='Loss', color='green')
ax2.tick_params(axis='y', labelcolor='green')
ax2.legend(loc='upper right')

fig.tight_layout()

plt.show()
```

Agora, com o código pronto, é possível executá-lo:

```
losses, weights_u, weights_w = train(x, y, epochs=150)
plot_training(losses, weights_u, weights_w)
```

Este Código gerará como saída uma imagem como a Figura 25.

Figura 25. A linha verde representa a perda, a linha pontilhada vermelha representa o peso W e a linha pontilhada azul representa o peso U.



Fonte: Adaptado de Vasilev *et al.*, 2019, p. 207.

Você pode obter mais informações sobre a técnica de *Backpropagation* ao longo do tempo no artigo "*Backpropagation through time: what it does and how to do it*" (WERBOS, 1990).

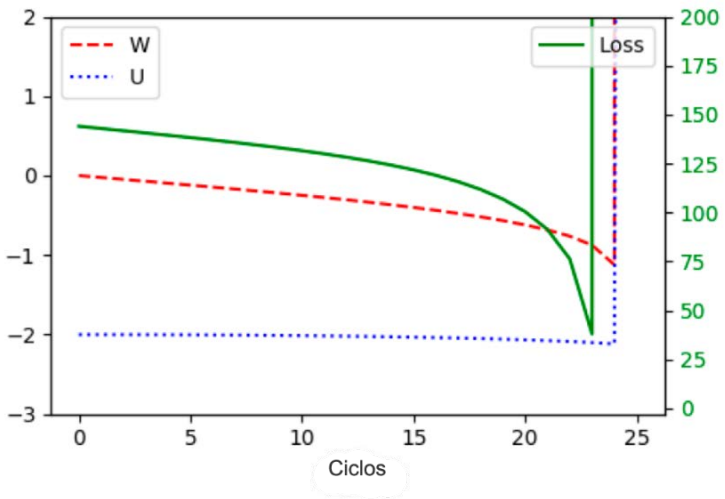
Problema da dissipação e explosão dos gradientes

O problema da dissipação ou explosão dos gradientes acontece quando os gradientes se tornam grandes ou pequenos demais, dificultando uma dependência de longo alcance (com 10 interações ou mais) na estrutura do conjunto de dados de entrada. Para visualizar o problema, vamos utilizar uma sequência de treinamento um pouco mais longa

```
x = np.array([[0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1,
  0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,
  1, 0, 1, 0, 1, 0]])

y = np.array([12])

losses, weights_u, weights_w = train(x, y, epochs=150)
plot_training(losses, weights_u, weights_w)
```

Você perceberá que sua IDE indicará diversos avisos (*RuntimeWarning*), e o gráfico da Figura 26 será mostrado. Os avisos acontecem, pois os valores finais dos parâmetros U e W não são números (NaN, *Not as Number*).

Figura 26. Cenário de explosão de gradiente.



Fonte: Adaptado de Vasilev *et al.*, 2019, p. 208.

No gráfico é possível perceber que os pesos se aproximam do ótimo e o custo cai até que eles dão um salto. Esse fenômeno é conhecido como explosão do gradiente. A dissipação

do gradiente é o fenômeno inverso, onde os valores decaem exponencialmente. Em ambos os casos, a rede neural recursiva perderá a capacidade de aprender as dependências da sequência a longo prazo. A melhor forma de resolver este problema é utilizando o LSTM (*Long Short-Term Memory*), que é uma rede neural recorrente modificada com suporte a DL4J.

Long short-term memory

As LSTMs podem resolver o problema causado pela dissipação ou explosão dos gradientes, ou seja, utilizando uma célula de memória especial, devolve à RNN a capacidade de aprender as dependências da sequência a longo prazo. As redes LSTM são tipos especiais de redes neurais recorrentes que possuem uma célula de estado cujo valor só pode ser inserido ou removido de forma explícita, mantendo o valor constante se não houver interferência externa. As células de estado só podem ser modificadas por portas específicas compostas por uma função sigmoid logística e multiplicação elemento a elemento. Como a função logística só gera valores entre 0 e 1, a multiplicação pode apenas reduzir o valor que passa pela porta.

Cada unidade LSTM possui dois tipos de conexão, uma com a interação (tempo) anterior e outra com a camada anterior. Uma rede LSTM é composta por três portas, uma de entrada, que protege a célula de estado de eventos de entrada irrelevantes, uma de esquecimento, ajuda a célula de estado a esquecer conteúdos prévios da memória, e uma de saída, que repassa o conteúdo da célula de memória para a saída da unidade LSTM. Nas redes LSTM, a função de ativação da camada é geralmente a tanh.

Modelagem de linguagem

Modelagem de Linguagem é a tarefa de computar a probabilidade de que uma sequência de palavras faça sentido. É uma tarefa essencial para diversas aplicações como reconhecimento de discurso, reconhecimento de imagens de caracteres, tradução pela máquina e correção da escrita. A modelagem de linguagem pode ser feita em nível das palavras ou em nível dos caracteres.

Modelos baseados em palavras

Define a distribuição probabilística para uma sequência de palavras. Dada uma sequência de palavras de tamanho m, este modelo define a probabilidade $P(w_1, w_2, \ldots, w_m)$ para toda a sequência de palavras. Pode ser utilizada para estimar a semelhança de diferentes frases em aplicações para processamento natural de palavras, ou como modelo generativo para criar um novo texto, visto que este modelo pode calcular a probabilidade de certa palavra ser a correta em uma sequência de palavras.

N-grams

A inferência da probabilidade de uma longa sequência de palavras é praticamente impossível. Então, para calcular a probabilidade de junção das palavras na sequência $P(w_1, w_2, \ldots, w_m)$, utiliza-se a seguinte regra:

$$P(w_1, w_2, \ldots, w_m) = P(w_1)P(w_1)P(w_1, w_2), \ldots, P(w_1, \ldots, w_{m-1})$$

A probabilidade da última palavra de acordo com as anteriores seria difícil de estimar a partir dos dados, então, a probabilidade da junção define que a iésima palavra é dependente apenas n – 1 palavras anteriores. Este modelo calcula a probabilidade da junção de n palavras sequenciais. O modelo n-gram distribui a junção das palavras em várias partes independentes.

Então, o modelo encontra os n-grams até um valor n, e conta a ocorrência de cada n-gram naquele conjunto. A partir dessa conta, é possível estimar a probabilidade da última palavra de cada n-gram de acordo com as n – 1 palavras.

1. gram (unigram): $P(\text{palavra}) = \dfrac{\text{contagem}(\text{palavras})}{\text{número total de palavras no conjunto}}$
2. gram (bigram): $P(w_{i-1}) = \dfrac{\text{contagem}(w_{i-1}, w_i)}{\text{contagem}(w_{i-1})}$
3. gram (trigram): $P(w_n, \ldots, w_{n+i-1}) = \dfrac{\text{contagem}(w_n, \ldots, w_{n+i-1}, w_{n+i})}{\text{contagem}(w_n, \ldots, w_{n+i-1})}$

Por exemplo, para a frase “Está fazendo sol hoje”, teríamos os seguintes n-grams:

1. gram: “Está”, “fazendo”, “sol” e “hoje”.
2. gram: “Está fazendo”, “fazendo sol” e “sol hoje”.
3. gram: “Está fazendo sol” e “fazendo sol hoje”.
4. gram: “Está fazendo sol hoje”.

A quantidade de n-grams depende do tamanho do vocabulário, que pode ser muito grande, dependendo do valor de n. Por exemplo, para um pequeno vocabulário contendo 100 palavras, a quantidade de 5-grams seria $10^5 = 10.000.000.000$ 5-grams diferentes. Tornando impossível o uso deste modelo para um número grande de n. Este problema é conhecido como “maldição da dimensionalidade”, ao aumentar o número de possíveis variáveis de entrada (palavras), aumenta-se exponencialmente a quantidade de combinações.

Modelos neurais de linguagem

Uma forma de superar a maldição da dimensionalidade é aprender uma representação de palavras de dimensão menor e distribuída. Essa representação distribuída é criada ao aprender uma função de incorporação que transforma o espaço de palavras em um espaço de menor dimensão. Sendo assim, um vocabulário de tamanho V é transformado em um vetor do tipo *one-hot encoded*, onde cada palavra é codificada de forma única. Então, a função de incorporação transforma este espaço V-dimensional em uma representação distribuída de tamanho D. A função de incorporação aprende informações semânticas das palavras e associa cada palavra do vocabulário com um vetor de representação contínua. Cada palavra corresponde a um ponto no espaço incorporado, e dimensões diferentes correspondem às propriedades gramaticais ou semânticas das palavras. A função de incorporação garante que palavras próximas no espaço incorporado possuem significado similar, então, essa característica pode ser explorada pelo modelo de linguagem.

Modelo de linguagem neural probabilístico

É possível aprender o modelo de linguagem e a função de incorporação através de uma rede neural *Feedforward* totalmente conectada. Dada uma sequência de n – 1 palavras $\left(w_{t-n+1},\ldots,w_{t-1}\right)$ a rede tenta inferir a distribuição probabilística da próxima palavra w_t. A camada de incorporação recebe a representação *one-hot* da palavra w_i e a transforma no vetor de incorporação através da multiplicação pela matriz de incorporação C. A matriz de incorporação C é a mesma para todas as palavras, de modo que todas utilizam a mesma função de incorporação. C pode ser representado por $V * D$, onde V é o tamanho do vocabulário e D é o tamanho do espaço incorporado. A incorporação resultante é concatenada e passada como entrada para a camada oculta, que utiliza a função de ativação Tanh. A camada de saída utiliza a função de ativação *softmax*, que mapeia a camada oculta para a distribuição probabilística da palavra no espaço.

Este modelo aprende, simultaneamente, a incorporação de todas as palavras do espaço (camada de incorporação) e o modelo da função probabilística para a sequência de palavras (camada de saída). O modelo ainda permite generalizar essa função probabilística para outros conjuntos de palavras que não estavam presentes no conjunto de treinamento, visto que a rede aprende as características incorporadas. Como a rede aprende baseada em um texto pré-existente, este é considerado uma tarefa de aprendizado não supervisionado.

Word2Vec

Para criar um vetor de incorporação com este modelo é necessária uma rede neural simples, que possui uma camada de entrada, uma camada oculta e uma camada de saída.

A camada de entrada é uma camada única de representação do tipo *one-hot encoded*. A camada de saída é uma camada única que utiliza *softmax* e prevê a palavra mais provável, de acordo com o contexto da palavra de entrada. A rede é treinada utilizando Gradiente Descendente e *Backpropagation*. O conjunto de treinamento consiste em pares de palavras, que aparecem próximas no texto, codificadas como *one-hot*.

O vetor de incorporação é representado pelos pesos de entrada para a camada oculta da rede, e é uma matriz $V * D$, onde V é o tamanho do vocabulário e D é o tamanho do vetor de incorporação, que é igual à quantidade de neurônios da camada oculta. Para cada amostra de entrada (palavra), apenas o vetor incorporado da própria palavra irá participar, ou seja, somente o neurônio relativo àquela palavra será ativado. Dependendo da forma como o modelo é treinado, ele pode ser classificado de diferentes formas:

- **Saco contínuo de Palavras (CBOW, do inglês *continuous bag of words*):** a rede é treinada para prever qual palavra encaixa em uma sequência de palavras onde uma palavra específica foi retirada intencionalmente.
- **Skip-gram:** é o oposto do CBOW, ou seja, dada uma palavra, a rede prevê quais outras palavras a cercam.

Existem outros modelos de incorporação, por exemplo:

GloVe: <https://nlp.stanford.edu/projects/glove/>.FastText: <https://fasttext.cc/>.

Palavras semanticamente correlacionadas estão próximas no espaço de incorporação. Sendo assim, o modelo de incorporação de palavras pode capturar analogias e diferenças entre palavras. Por exemplo, a rede pode perceber que a diferença entre as palavras incorporadas **homem** e **mulher** significam o gênero e que a diferença entre elas é a mesma que em outras palavras relativas ao gênero, como **rainha** e **rei**.

Modelos baseados em caracteres

Geralmente, a modelagem de linguagem é trabalhada em nível de palavras, em que a distribuição se dá de acordo com um vocabulário de palavras V predefinido. Este modelo é limitado, visto que o vocabulário pode se tornar muito grande, inviabilizando o modelo, o vocabulário pode possuir *strings* que não são palavras (como números), pode haver palavras que não fazem parte do vocabulário. Nestes casos, é recomendado o uso de modelos baseados em caracteres.

Os modelos baseados em caracteres modelam a distribuição de acordo com uma sequência de caracteres, e não de acordo com as palavras, o que permite calcular a probabilidade em

um vocabulário menor. Um vocabulário, neste modelo, compreende todos os caracteres presentes no corpo do texto.

O lado negativo deste modelo é que, ao modelar sequências de caracteres, e não de palavras, é necessário que haja uma sequência maior para capturar a mesma informação ao longo do tempo. Neste caso, é necessário utilizar um modelo de linguagem LSTM para capturar a dependência entre as letras. A LSTM modela a probabilidade do próximo caractere de acordo com os caracteres anteriores.

O texto completo é muito longo para utilizar *backpropagation* ao longo do tempo (BPTT), então, é necessário dividir o conjunto de treinamento em lotes com sequências de mesmo tamanho, e treinar a rede lote por lote, em uma variante do BPTT chamada *backpropagation* truncado ao longo do tempo. Como os lotes serão sequenciais, o último estado de um lote será o estado inicial do próximo lote.

Utilizando Tensorflow para geração de novo texto

Os autores Vasilev *et al.* (2019) utilizaram a obra ***War and Peace***, de Leo Tolstoy, para treinar uma rede LSTM utilizando *Tensorflow* na geração de texto. O código completo e as bases para treinamento podem ser verificados no link a seguir: https://github.com/ivan-vasilev/Python-Deep-Learning-SE/tree/master/ch07/language%20model.

O livro foi escolhido por possuir mais de 500.000 palavras, estar em domínio público e poder ser acessado gratuitamente no Projeto Gutenberg. Disponível em: http://www.gutenberg.org/. No pré-processamento, foram retiradas as partes pré-textuais, e as quebras de linhas entre as sentenças foi reduzida a duas (arquivo data_processing.py do github):

```
"""
  Processamento de arquivo de texto para treinamento de
  modelos de linguagem
"""
from __future__ import print_function, division
import codecs
import re

"""
  Baixe os arquivos war_and_peace.txt e wap.txt no link
  https://github.com/ivan-vasilev/Python-Deep-Learning-
  SE/tree/master/ch07/language%20model. E salve na pasta
```

```
  do projeto.
"""
filepath = 'war_and_peace.txt' # entrada
out_file = 'wap.txt' # alvos

# Regexes usados para limpar o texto
NEW_LINE_IN_PARAGRAPH_REGEX = re.compile(r'(\S)\n(\S)')
MULTIPLE_NEWLINES_REGEX = re.compile(r'(\n)(\n)+')

# Lendo o texto como string
with codecs.open(filepath, encoding='utf-8', mode='r') as
  f_input:
    book_str = f_input.read()

# Limpando
book_str = NEW_LINE_IN_PARAGRAPH_REGEX.sub('\g<1> \g<2>',
  book_str)
book_str = MULTIPLE_NEWLINES_REGEX.sub('\n\n', book_str)
book_str = re.sub(u'[\u201c\u201d]', '"', book_str)

# Escrevendo o texto processado em um arquivo
with codecs.open(out_file, encoding='utf-8', mode='w')as
  f_output:
    f_output.write(book_str)
```

Para alimentar a rede, é necessário converter os dados em formato numérico, dessa forma, cada caractere será associado a um número inteiro. No texto de exemplo, foi possível identificar 98 caracteres diferentes. É necessário, também, definir as entradas e os alvos, predizendo, para cada caractere de entrada, o próximo caractere. No exemplo, todos os lotes serão extraídos sequencialmente (arquivo data_reader.py do github):

```
from __future__ import print_function, division

import codecs

import numpy as np
from six.moves import range
```

```
class DataReader(object):
  """Leitor de dados usado para treinamento de modelos de
    linguagem """

  def __init__(self, filepath, batch_length, batch_size):
    self.batch_length = batch_length
    self.batch_size = batch_size
    # Lendo os dados para uma String
    with codecs.open(filepath, encoding='utf-8',
      mode='r') as f:
        self.data_str = f.read()
    self.data_length = len(self.data_str)
    print('data_length: ', self.data_length)
    # Criar uma lista de caracteres, os índices são os
    # mesmos das classes para softmax.
    char_set = set()
    for ch in self.data_str:
      char_set.add(ch)
    self.char_list = sorted(list(char_set))
    print('char_list: ', len(self.char_list),
      self.char_list)
    # Criar mapeamento reverse para verificar os
    # índices com base nos caracteres.
    self.char_dict = {val: idx for idx, val in
      enumerate(self.char_list)}
    print('char_dict: ', self.char_dict)
    # Inicializar índices iniciais aleatórios
    self.reset_indices()

  def reset_indices(self):
    self.start_idxs = np.random.random_integers(
      0, self.data_length, self.batch_size)

  def get_sample(self, start_idx, length):
```

```
        # Obtenha uma amostra e envolva a sequência de dados
        return [self.char_dict[self.data_str[i %
          self.data_length]]
          for i in range(start_idx, start_idx + length)]

    def get_input_target_sample(self, start_idx):
        sample = self.get_sample(start_idx,
          self.batch_length + 1)
        inpt = sample[0:self.batch_length]
        trgt = sample[1:self.batch_length + 1]
        return inpt, trgt

    def get_batch(self, start_idxs):
        input_batch = np.zeros((self.batch_size,
          self.batch_length), dtype=np.int32)
        target_batch = np.zeros((self.batch_size,
          self.batch_length), dtype=np.int32)
        for i, start_idx in enumerate(start_idxs):
          inpt, trgt =
            self.get_input_target_sample(start_idx)
          input_batch[i, :] = inpt
          target_batch[i, :] = trgt
        return input_batch, target_batch

    def __iter__(self):
        while True:
          input_batch, target_batch =
            self.get_batch(self.start_idxs)
          self.start_idxs = (self.start_idxs +
            self.batch_length) % self.data_length
          yield input_batch, target_batch

def main():
    filepath = './wap.txt'
    batch_length = 10
```

```
  batch_size = 2
  reader = DataReader(filepath, batch_length, batch_size)
  s = 'As in the question of astronomy then, so in the
    question of history now,'
  print([reader.char_dict[c] for c in s])


if __name__ == "__main__":
  main()
```

Após o pré-processamento, é possível criar a rede LSTM. Será treinada uma rede LSTM de duas camadas com 512 células cada camada (arquivo model.py do github):

```
from __future__ import print_function, division

import time

import data_reader
import numpy as np
import tensorflow as tf


class Model(object):
  """Modelo de RNN para Processamento de Linguagem"""

  def __init__(self, batch_size, sequence_length,
    lstm_sizes, dropout, labels, save_path):
      self.batch_size = batch_size
      self.sequence_length = sequence_length
      self.lstm_sizes = lstm_sizes
      self.labels = labels
      self.label_map = {val: idx for idx, val in
        enumerate(labels)}
      self.number_of_characters = len(labels)
      self.save_path = save_path
      self.dropout = dropout
```

Inicialmente, serão definidos os espaços reservados (*placeholder*), que são os *links* entre o modelo e o conjunto de treinamento, para as entradas e alvos. A rede pode ser alimentada com um lote único ao definirmos este valor no *placeholder*. A primeira dimensão, tanto da entrada quanto do alvo, é o tamanho do lote; a segunda dimensão é o tamanho da sequência no texto.

```
def init_graph(self):
  # Tamanho da sequência de variáveis
  self.inputs = tf.placeholder(
    tf.int32, [self.batch_size,
    self.sequence_length])
  self.targets = tf.placeholder(
    tf.int32, [self.batch_size,
    self.sequence_length])
  self.init_architecture()
  self.saver = tf.train.Saver(
    tf.trainable_variables())
```

Para apresentar os caracteres à rede, é necessário transformá-los em vetores *one-hot*. Em seguida, será definida a arquitetura LSTM multicamadas, serão definidas as células LSTM para cada camada (lstm_sizes é uma lista de tamanhos para cada camada, neste exemplo, 512,512), e depois, as células serão ajustadas em uma rede neural recursiva multicamadas.

```
def init_architecture(self):
  # Definir uma célula LSTM multicamadas
  self.one_hot_inputs = tf.one_hot(
    self.inputs, depth=self.number_of_characters)
  cell_list = [tf.nn.rnn_cell.LSTMCell(lstm_size)
    for lstm_size in self.lstm_sizes]
  self.multi_cell_lstm =
    tf.nn.rnn_cell.MultiRNNCell(cell_list)
```

A fim de armazenar os estados entre cada lote, é necessário saber o estado inicial da rede e armazená-lo em uma variável:

```
  # Estado inicial da memória LSTM.
  # Manter o estado na memória gráfica para usar entre
  # lotes.
```

```
self.initial_state =
  self.multi_cell_lstm.zero_state(
  self.batch_size, tf.float32)
# Converter em variáveis para que o estado possa
# ser armazenado entre lotes.
# Observe que o estado LSTM é uma tupla de tensores,
# Esta estrutura deve ser reorganizada de modo a
# usar o estado do LSTM.
self.state_variables =
  tf.contrib.framework.nest.pack_sequence_as(
  self.initial_state,
  [tf.Variable(var, trainable=False)
    for var in
    tf.contrib.framework.nest.flatten(
    self.initial_state)])
```

Com o estado inicial definido como uma variável, é possível desenrolar a rede no tempo. O método dynamic_rnn, do *Tensorflow* faz isso de forma dinâmica, de acordo com o comprimento da sequência de entrada:

```
# Definir a rede neural recursiva no tempo
lstm_output, final_state = tf.nn.dynamic_rnn(
  cell=self.multi_cell_lstm,
  inputs=self.one_hot_inputs,
  initial_state=self.state_variables)
```

O próximo passo é armazenar o estado final para ser utilizado como estado inicial pelo próximo lote. O método state_variable.assign será utilizado para armazenar cada estado final na variável de estado inicial correta. O método control_dependencies força a atualização do estado antes de retornar à saída do LSTM:

```
# Força o estado inicial a ser passado como novo
# estado para o próximo lote antes de retornar a
# saída
store_states = [
  state_variable.assign(new_state)
  for (state_variable, new_state) in zip(
    tf.contrib.framework.nest.flatten(
```

```
            self.state_variables),
        tf.contrib.framework.nest.flatten(
            final_state))]
with tf.control_dependencies(store_states):
  lstm_output = tf.identity(lstm_output)
```

A fim de obter a saída logit a partir da saída LSTM final, é necessário aplicar uma transformação linear à saída, para que essa passe a ter a dimensão tamanho Lote • tamanho Sequência • quantidade de símbolos:

```
# Reorganizar para aplicar a transformação linear
# a todas as saídas.
output_flat = tf.reshape(lstm_output, (-1,
  self.lstm_sizes[-1]))
```

Então, é possível aplicar a transformação linear com matriz de peso W e bias b para obter o logits, aplicar a função *softmax*, e remodelar isso para um tensor de tamanho Lote • tamanho Sequência • quantidade de símbolos:

```
# Define output layer
self.logit_weights = tf.Variable(
  tf.truncated_normal(
    (self.lstm_sizes[-1],
    self.number_of_characters), stddev=0.01),
    name='logit_weights')
self.logit_bias = tf.Variable(
  tf.zeros((self.number_of_characters)),
    name='logit_bias')
# Aplicar a transformação na última camada
self.logits_flat = tf.matmul(
  output_flat, self.logit_weights) +
    self.logit_bias
probabilities_flat = tf.nn.softmax(self.logits_flat)
self.probabilities = tf.reshape(
  probabilities_flat,
  (self.batch_size, -1,
  self.number_of_characters))
```

Após a definição da entrada, alvos e arquitetura de rede, é possível implementar o treinamento. Inicialmente, é necessário definir a função de perda, que será a função de entropia cruzada. Os dados devem ser remodelados em um vetor unidimensional para que sejam compatíveis com o logit de saída da rede. Em seguida, será definida a operação de treinamento do *Tensorflow*:

```
def init_train_op(self, optimizer):
  # Remodelando os alvos para que sejam compatíveis
  # com o formato dos Logits
  targets_flat = tf.reshape(self.targets, (-1,))
  # Aplicar a função de perda em todas as saídas
  loss =
    tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits=self.logits_flat, labels=targets_flat,
    name='x_entropy')
  self.loss = tf.reduce_mean(loss)
  trainable_variables = tf.trainable_variables()
  gradients = tf.gradients(loss, trainable_variables)
  gradients, _ = tf.clip_by_global_norm(gradients, 5)
  self.train_op =
    optimizer.apply_gradients(zip(gradients,
    trainable_variables))
```

Uma vez que o modelo já está treinado, é possível gerar um texto a partir de amostras das sequências. Vamos apresentar ao modelo uma *string* inicial (prime_string), a partir daí, é possível gerar o próximo caractere baseado na distribuição da *softmax*, então, este será passado para a rede e um novo caractere será gerado, e isso será feito várias vezes:

```
def sample(self, session, prime_string, sample_length):
  self.reset_state(session)
  # Estado inicial
  print('prime_string: ', prime_string)
  for character in prime_string:
    character_idx = self.label_map[character]
    out = session.run(
      self.probabilities,
      feed_dict={self.inputs:
      np.asarray([[character_idx]])})
```

```
        output_sample = prime_string
        print('start sampling')
        # Amostrar por sample_length passos
        for _ in range(sample_length):
          sample_label = np.random.choice(
            self.labels, size=(1), p=out[0, 0])[0]
          output_sample += sample_label
          sample_idx = self.label_map[sample_label]
          out = session.run(
            self.probabilities,
            feed_dict={self.inputs:
            np.asarray([[sample_idx]])})

        return output_sample
```

Com as funções de treinamento e amostragem definidas, vamos finalizar o código e rodar para gerar texto:

```
    def reset_state(self, session):
      for state in
        tf.contrib.framework.nest.flatten(
        self.state_variables):
          session.run(state.initializer)

    def save(self, sess):
      self.saver.save(sess, self.save_path)

    def restore(self, sess):
      self.saver.restore(sess, self.save_path)

def train_and_sample(minibatch_iterations, restore):
  tf.reset_default_graph()
  batch_size = 64
  lstm_sizes = [512, 512]
  batch_len = 100
  learning_rate = 2e-3

  filepath = './wap.txt'
```

```
data_feed = data_reader.DataReader(
  filepath, batch_len, batch_size)
labels = data_feed.char_list
print('labels: ', labels)

save_path = './model.tf'
model = Model(
  batch_size, batch_len, lstm_sizes, 0.8, labels,
  save_path)
model.init_graph()
optimizer = tf.train.AdamOptimizer(learning_rate)
model.init_train_op(optimizer)

init_op = tf.initialize_all_variables()
with tf.Session() as sess:
  sess.run(init_op)
  if restore:
    print('Restoring model')
    model.restore(sess)
  model.reset_state(sess)
  start_time = time.time()
  for i in range(minibatch_iterations):
    input_batch, target_batch =
      next(iter(data_feed))
    loss, _ = sess.run(
      [model.loss, model.train_op],
      feed_dict={model.inputs: input_batch,
          model.targets: target_batch})
    if i % 50 == 0 and i != 0:
      print('i: ', i)
      duration = time.time() - start_time
      print('loss: {} ({} sec.)'.format(loss,
          duration))
      start_time = time.time()
    if i % 1000 == 0 and i != 0:
      model.save(sess)
```

```
      if i % 100 == 0 and i != 0:
        print('Reset initial state')
        model.reset_state(sess)
      if i % 1000 == 0 and i != 0:
        print('Reset minibatch feeder')
        data_feed.reset_indices()
    model.save(sess)

print('\n sampling after {}
  iterations'.format(minibatch_iterations))
tf.reset_default_graph()
model = Model(
  1, None, lstm_sizes, 1.0, labels, save_path)
model.init_graph()
init_op = tf.initialize_all_variables()
with tf.Session() as sess:
  sess.run(init_op)
  model.restore(sess)
  print('\nSample 1:')
  sample = model.sample(
    sess, prime_string=u'\n\nThis feeling was ',
      sample_length=500)
  print(u'sample: \n{}'.format(sample))
  print('\nSample 2:')
  sample = model.sample(
    sess, prime_string=u'She was born in the year
      ', sample_length=500)
  print(u'sample: \n{}'.format(sample))
  print('\nSample 3:')
  sample = model.sample(
    sess, prime_string=u'The meaning of this all is
      ',
    sample_length=500)
  print(u'sample: \n{}'.format(sample))
  print('\nSample 4:')
  sample = model.sample(
    sess,
```

```
        prime_string=u'In the midst of a conversation
          on political matters Anna Pávlovna burst
          out:,',
        sample_length=500)
      print(u'sample: \n{}'.format(sample))
      print('\nSample 5:')
      sample = model.sample(
        sess, prime_string=u'\n\nCHAPTER X\n\n',
        sample_length=500)
      print(u'sample: \n{}'.format(sample))
      print('\nSample 6:')
      sample = model.sample(
        sess, prime_string=u'\"If only you knew,\"',
        sample_length=500)
      print(u'sample: \n{}'.format(sample))


def main():
  print("Generating new text with character-level
    TensorFlow LSTM")

  total_iterations = 500
  print('\n\n\nTrain for {}'.format(500))
  print('Total iters: {}'.format(total_iterations))
  train_and_sample(500, restore=False)
  for i in [500, 1000, 3000, 5000, 10000, 30000, 50000,
    100000, 300000]:
    total_iterations += i
    print('\n\n\nTrain for {}'.format(i))
    print('Total iters: {}'.format(total_iterations))
    train_and_sample(i, restore=True)


if __name__ == "__main__":
  main()
```

Após o código completo, é possível treinar a rede e mostrar o que foi aprendido a cada interação. A rede será inicializada com a frase "*she was born in the year*". Vejamos como ela se comporta:

- Após 500 lotes, a saída será: *She was born in the year sive but us eret tuke Toffhin* e *feale shoud pille saky doctonas laft the comssing hinder to gam the droved at ay vime*.
- Após 5.000 lotes, a saída será: *She was born in the year he had meaningly many of Seffer Zsites. Now in his crownchy-destruction, eccention, was formed a wolf of Veakov one also because he was congrary, that he suddenly had first did not reply*.
- Após 50.000 lotes, a saída será: *She was born in the year 1813. At last the sky may behave the Moscow house there was a splendid chance that had to be passed the Rostóvs', all the times: sat retiring, showed them to confure the sovereigns*.
- Após 500.000 lotes, a saída será: *She was born in the year 1806, when he entered his thought on the words of his name. The commune would not sacrifice him: "What is this?" asked Natásha. "Do you remember*?".

É possível perceber que a rede montou sentenças, apesar de não muito coerentes, mas, inclusive, aprendeu a usar pontuação e citação.

O código implementado contém outras frases de inicialização que irão gerar saídas diferentes.

Observe todas as saídas e veja a evolução da rede.

O exemplo apresentado por Vasilev *et al*. (2019) foi implementado com apenas duas camadas ocultas. E utilizou, como base para treinamento, um livro em inglês. Encorajamos que você:

- Aumente o tamanho das camadas LSTM e veja se a acurácia da rede é melhorada.
- Aumente a quantidade de camadas ocultas e veja se a acurácia da rede é melhorada.
- Aumente o tamanho das camadas LSTM e a quantidade de camadas ocultas e veja se a acurácia da rede é melhorada.
- Repita o experimento com algum texto de sua preferência, em português, e veja se a rede é capaz de montar frases coerentes em nosso idioma.

Lembre-se, quanto maior for o conjunto de treinamento, melhor a rede será em montar textos compreensíveis.

CAPÍTULO 3

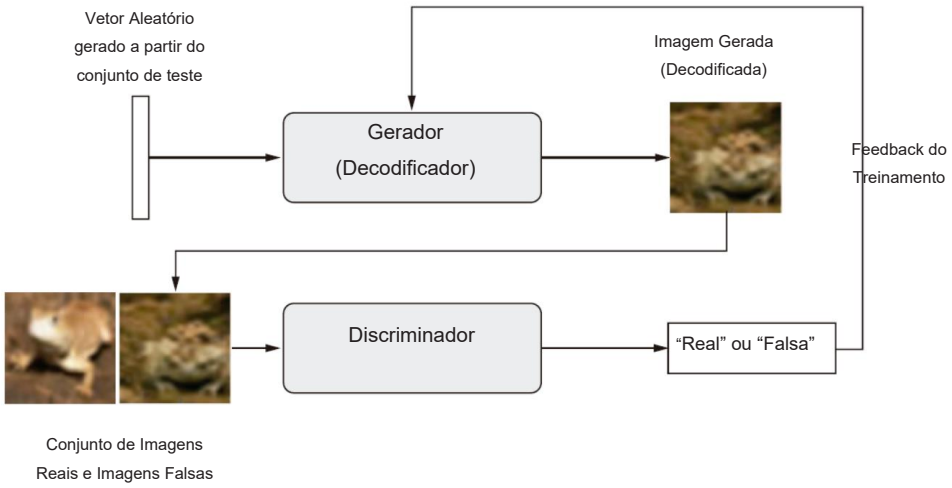
REDES GENERATIVAS ADVERSÁRIAS

As Redes Generativas Adversárias (GANs, do inglês, *Generative Adversarial Networks*) são um modelo de rede apresentado inicialmente por Goodfellow *et al.* (2014), que permitem a geração, principalmente, de imagens sintéticas realistas, pois forçam as imagens geradas a serem estatisticamente parecidas com as imagens reais. De acordo com Patterson e Gibson (2017), esse tipo de estrutura tem sido muito utilizado para sintetizar novas imagens a partir de dado conjunto de treinamento. Note que as GANs podem ser utilizadas para gerar qualquer tipo de dado, porém, são mais utilizadas na geração de imagens.

São redes que utilizam o aprendizado não supervisionado para treinar dois modelos em paralelo, uma rede que constrói uma imagem, e outra rede que verifica o quão parecida essa imagem é de uma real. Ambas as redes são treinadas simultaneamente, sempre buscando a melhoria uma da outra. Chollet (2018) explica que a rede geradora recebe como entrada um vetor aleatório, e o codifica em imagens sintéticas. Enquanto isso, a rede discriminadora, ou adversária, tem como entrada uma imagem (real ou sintética) e consegue prever se essa imagem foi gerada pela rede geradora, ou veio do conjunto de treinamento.

Este tipo de abordagem é chamado de redes adversárias, pois elas são treinadas para enganar uma a outra, ou seja, a rede geradora tenta criar uma imagem tão realística que a rede discriminadora entenderá como real, e a rede discriminadora é treinada para distinguir imagens reais das sintéticas. Na Figura 27 é possível perceber o fluxo de funcionamento das GANs, em que um vetor aleatório de valores presentes no conjunto de treinamento é apresentado ao gerador, que sintetiza uma imagem a partir dele, e apresenta essa imagem ao discriminador, juntamente com imagens advindas diretamente do conjunto de treinamento, então, o discriminador determina quais das imagens são reais e quais são falsas, e devolve um *feedback* para o gerador indicando pequenas alterações para que a imagem sintetizada pareça mais realística, permitindo ao gerador se aperfeiçoar.

Figura 27. Exemplo de Funcionamento das Redes Generativas Adversárias.



Fonte: Adaptado de Chollet, 2018, p. 306.

Esse tipo de rede é de difícil treinamento, pois não há um número fixo de ciclos para treinamento. As GANs são sistemas dinâmicos, nos quais a otimização procura sempre o equilíbrio entre as imagens sintetizadas e as reais, de modo que o discriminador não consiga distinguir entre elas. As redes geradoras aprendem como representar, de forma eficiente, as características presentes no conjunto de treinamento, como contornos, grupos de *pixel* similares, e outros padrões naquele grupo de imagens.

Em uma GAN, a rede discriminadora é uma rede convolucional típica, o que permite o treinamento paralelo dela e da rede geradora. Já a rede geradora utiliza uma camada especial, chamada de camada deconvolucional (*deconvolutional layer*). Durante o treinamento de uma GAN, é utilizado *backprpagation* para atualizar os parâmetros da rede geradora a fim de gerar saídas mais realísticas.

Redes Deconvolucionais

Tipo de rede desenvolvido por Matthew Zeiler e Rob Fergus que auxilia a verificação de quais características foram ativadas em uma camada, e qual sua relação com o conjunto de entrada. As camadas deconvolucionais dessa rede, ao contrário de uma camada convolucional, mapeiam as características em *pixels*, ao modelar imagens. Essa característica permite a geração de imagens como saídas de redes neurais, como as GANs.

As redes deconvolucionais são treinadas camada a camada, de forma não supervisionada. Elas possuem múltiplas camadas deconvolucionais empilhadas, onde cada camada é treinada na entrada da camada anterior. A ideia geral da informação passando pelas camadas é que a saída de cada camada é uma representação esparsa da entrada naquela camada.

Gerando novas imagens MNIST com GANs e keras

Vasilev *et al* (2019) apresentam o passo a passo para gerar novas imagens a partir da base de dados MNIST usando Keras. Para começar, vamos importar as bibliotecas necessárias:

```
import matplotlib.pyplot as plt
import numpy as np
from keras.datasets import mnist
from keras.layers import BatchNormalization, Input, Dense,
    Reshape, Flatten
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Sequential, Model
from keras.optimizers import Adam
```

Em seguida, vamos implementar a função build_generator, que é a rede geradora. No exemplo apresentado pelo autor, ele utiliza uma rede totalmente conectada como gerador:

```
def build_generator(latent_dim: int):
"""
  Build discriminator network
  :param latent_dim: latent vector size
"""

model = Sequential([
  Dense(128, input_dim=latent_dim),
  LeakyReLU(alpha=0.2),
  BatchNormalization(momentum=0.8),
  Dense(256),
  LeakyReLU(alpha=0.2),
  BatchNormalization(momentum=0.8),
  Dense(512),
  LeakyReLU(alpha=0.2),
  BatchNormalization(momentum=0.8),
  Dense(np.prod((28, 28, 1)), activation='tanh'),
  # Reajustar para o mesmo tamanho das imagens MNIST
  Reshape((28, 28, 1))
])
```

```
model.summary()

# O vetor latente de entrada z
z = Input(shape=(latent_dim,))
generated = model(z)

# Construir o modelo de entrada e saída
return Model(z, generated)
```

Com a rede geradora criada, é hora de construir a rede discriminadora. Novamente, essa será uma rede totalmente conectada simples:

```
def build_discriminator():
"""
  Build discriminator network
"""

model = Sequential([
  Flatten(input_shape=(28, 28, 1)),
  Dense(256),
  LeakyReLU(alpha=0.2),
  Dense(128),
  LeakyReLU(alpha=0.2),
  Dense(1, activation='sigmoid'),
], name='discriminator')

model.summary()

image = Input(shape=(28, 28, 1))
output = model(image)

return Model(image, output)
```

Após a implementação das redes geradora e discriminadora, é hora de implementar a função de treinamento:

```
def train(generator, discriminator, combined, steps,
  batch_size):
```

```
"""
  Train the GAN system
  :param generator: generator
  :param discriminator: discriminator
  :param combined: stacked generator and discriminator
  we'll use the combined network when we train the
    generator
  :param steps: number of alternating steps for training
  :param batch_size: size of the minibatch
"""

# Carregando a base de dados
(x_train, _), _ = mnist.load_data()

# Reajustar a escala para um intervalo entre [-1, 1]
x_train = (x_train.astype(np.float32) - 127.5) / 127.5
x_train = np.expand_dims(x_train, axis=-1)

# Verdades para o Discriminador
real = np.ones((batch_size, 1))
fake = np.zeros((batch_size, 1))

latent_dim = generator.input_shape[1]

for step in range(steps):
  # Treinando o discriminador

  # Selecione um conjunto aleatório de imagens reais
  real_images = x_train[np.random.randint(0,
    x_train.shape[0], batch_size)]

  # Pacote conjunto de valores (ruídos)
  noise = np.random.normal(0, 1, (batch_size, latent_dim))

  # Gerando um conjunto de novas imagens
```

```
    generated_images = generator.predict(noise)

    # Treinando o discriminador
    discriminator_real_loss =
      discriminator.train_on_batch(real_images, real)
    discriminator_fake_loss =
      discriminator.train_on_batch(generated_images,
        fake)
    discriminator_loss = 0.5 *
      np.add(discriminator_real_loss,
      discriminator_fake_loss)

    # Treinando o gerador
    # Vetor Z de valores latentes aleatórios
    noise = np.random.normal(0, 1, (batch_size, latent_dim))

    # Treinando o gerador
    # Observe que usamos os rótulos "válidos" para as imagens
    # geradas para tentar maximizar o custo do discriminador
    generator_loss = combined.train_on_batch(noise, real)

    # Mostrando o progresso
    print("%d [Discriminator loss: %.4f%%, acc.: %.2f%%]
      [Generator loss: %.4f%%]" %
      (step, discriminator_loss[0], 100 *
        discriminator_loss[1], generator_loss))
```

O próximo passo será gerar uma função comum para mostrar algumas das imagens geradas após a fase de treinamento, chamaremos essa função de plot_generated_images. Essa função será criada da seguinte forma: I) vamos criar um grid de tamanho nxn; II) vamos criar um vetor latente de valores aleatórios de tamanho nxn para cada imagem gerada; III) vamos gerar as imagens e colocá-las em cada célula do grid; IV) vamos mostrar o resultado:

```
def plot_generated_images(generator):
"""
  Display a nxn 2D manifold of digits
```

```
    :param generator: the generator
  """
    n = 10
    digit_size = 28

    # Grande matriz contendo todas as imagens
    figure = np.zeros((digit_size * n, digit_size * n))

    latent_dim = generator.input_shape[1]

    # Distribuição aleatória dos valores latentes n*n
    noise = np.random.normal(0, 1, (n * n, latent_dim))

    # Gerando as imagens
    generated_images = generator.predict(noise)

    # Preenchendo a grande matriz com as imagens
    for i in range(n):
      for j in range(n):
        slice_i = slice(i * digit_size, (i + 1) *
          digit_size)
        slice_j = slice(j * digit_size, (j + 1) *
          digit_size)
        figure[slice_i, slice_j] =
          np.reshape(generated_images[i * n + j],
          (28, 28))

    # Mostrando os resultados
    plt.figure(figsize=(6, 5))
    plt.axis('off')
    plt.imshow(figure, cmap='Greys_r')
    plt.show()
```

O último passo será montar os construtores do gerador, do discriminador, e agregar ambos na rede. Rodar o treinamento com 15.000 passos usando um otimizador ADAM, e mostrar os resultados ao final:

```
if __name__ == '__main__':
  latent_dim = 64
```

```
# Construir e compilar o discriminador
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
  optimizer=Adam(lr=0.0002, beta_1=0.5),
  metrics=['accuracy'])

# Construir o gerador
generator = build_generator(latent_dim)

# Entrada z do gerador
z = Input(shape=(latent_dim,))
generated_image = generator(z)

# Apenas treine o gerador para o modelo combinado
discriminator.trainable = False

# O gerador recebe as imagens geradas como entrada e
# determina sua validade
real_or_fake = discriminator(generated_image)

# Empilhe o gerador e o discriminador em um modelo
# combinado
# Treine o gerador para enganar o discriminador
combined = Model(z, real_or_fake)
combined.compile(loss='binary_crossentropy',
  optimizer=Adam(lr=0.0002, beta_1=0.5))

# Treine o Sistema GAN
train(generator=generator,
  discriminator=discriminator,
  combined=combined,
  steps=15000,
  batch_size=128)
```

```
# Mostre aleatoriamente algumas imagens geradas
plot_generated_images(generator)
```

O resultado apresentado deverá ser similar à Figura 28.

Figura 28. Apresentação aleatória de imagens geradas pelo sistema GAN.

Fonte: Vasilev *et al.*, 2019, p. 196.

Existem tipos especiais de GAN:

DCGAN (*Deep Convolutional Generative Adversarial Networks*): desenvolvida por Radford, Metz e Chintala (2015), tem uma boa performance na geração de imagens. Exemplos, códigos e maiores detalhes podem ser verificados no repositório dos autores. Disponível em: <https://github.com/Newmu/dcgan_code>.

GAN Condicional (*Conditional GAN*): apresentado por Mirza e Osindero (2014), esse tipo de GAN se utiliza, também, dos rótulos das classes, a fim de gerar imagens de uma classe específica.

LAPGAN: apresentado por Denton *et al.* (2015), mostra que uma série de GANs condicionais podem ser treinadas para gerar uma imagem inicial de baixa resolução, e então, adicionar detalhes a essa imagem de forma incremental. O nível de detalhes nas imagens geradas por essa rede chega a confundir não apenas as redes discriminadoras, mas também o ser humano.

REFERÊNCIAS

AREL, I.; ROSE, D. C.; KARNOWSKI, T. P. Deep machine learning – a new frontier in artificial intelligence research. **IEEE Computational intelligence magazine**, v. 5, n. 4, pp. 13-18, 2010.

BENGIO, Y. Learning deep architectures for AI. **Foundations and trends® in machine learning**, v. 2, n.1, p. 1-127, 2009.

BENGIO, Y.; COURVILLE, A.; VINCENT, P. representation learning: a review and new perspectives. **IEEE Transactions on pattern analysis and machine intelligence**, v. 35, n.8, pp. 1.798-1.828, 2013.

BRYNJOLFSSON, E.; MITCHELL, T. M. What can machine learning do? Workforce Implications. **Science**, v. 358, n. 6.370, pp. 1.530-1.534, 2017.

BUDUMA, N. **Fundamentals of deep learning:** designing next-generation machine intelligence algorithms. Sebastopol, CA: O'Reilly Media, 2017.

CHO, K.; MERRIËNBOER, B. van; GULCEHRE, C.; BAHDANAU, D.; BOUGARES, F.; SCHWENK, H.; BENGIO, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *In*: Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar. **Anais**... Doha, Qatar: 2014.

CHOLLET, F. **Deep learning with python**. New York: Manning Publications, 2018.

DATA SCIENCE ACADEMY. **Deep learning book**. [s.l.] Data Science Academy, 2017.

DECHTER, R. Learning while searching in constraint-satisfaction-problems. *In*: Conference Proceedings of the Association for the Advancement of Artificial Intelligence 1986, **Anais**...1986.

DENG, L. An overview of deep-structured learning for information processing. *In*: Proceedings of Asian-Pacific Signal & Information Processing Annual Summit and Conference (APSIPA-ASC), Xi'an. **Anais**... Xi'an: 2011. Disponível em: https://www.microsoft.com/en-us/research/publication/an-overview-of-deep-structured-learning-for-information-processing/. Acesso em: 27/05/2019.

DENG, L. *et al*. Recent advances in deep learning for speech research at Microsoft. *In*: ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing – Proceedings, Vancouver, BC, Canadá. **Anais**... Vancouver, BC, Canadá: 2013.

DENG, L.; YU, D. Deep Learning: Methods and Applications. **Foundations and trends® in signal processing**, v. 7, n. 3-4, pp. 197-387, 2013.

DENTON, E.; CHINTALA, S.; SZLAM, A.; FERGUS, R. Deep generative image models using a laplacian pyramid of adversarial networks. *In*: Neural Information Processing Systems (NIPS), Montreal, Canadá. **Anais**... Montreal, Canadá: 2015. Disponível em: http://arxiv.org/abs/1506.05751.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. [s.l.] MIT Press, 2016.

GOODFELLOW, I.; POUGET-ABADIE, J.; MIRZA, M.; XU, B.; WARDE-FARLEY, D.; OZAIR, S.; COURVILLE, A.; BENGIO, Y. Generative adversarial nets. In: Advances in Neural Information Processing Systems 27 (NIPS 2014), **Anais**...2014. Disponível em: https://papers.nips.cc/paper/5423-generative-adversarial-nets. Acesso em: 27/5/2019.

HINTON, G. E. Learning Multiple Layers of Representation. **Trends in cognitive sciences**, v. 11, n. 10, pp. 428-434, 2007.

HINTON, G. E. *et al*. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. **IEEE Signal Processing Magazine**, v. 29, n. 6, pp. 82-97, 2012.

HINTON, G. E.; OSINDERO, S.; TEH, Y. W. A fast learning algorithm for deep belief nets. **Neural computation**, v. 18, n. 7, pp. 1.527–1.554, 2006.

JAIN, S. **An Overview of Regularization Techniques in Deep Learning (with Python code)**. 2018. Disponível em: https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/. Acesso em: 27/05/2019.

JOSHI, P. **Artificial intelligence with python:** build real-world artificial intelligence applications with python to intelligently interact with the world around you. Birmingham, UK: Packt Publishing, 2017.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Convolutional neural networks imagenet classification with deep convolutional neural network. **Communications of the Acm**, v. 60, n. 6, pp. 84-90, 2017.

LECUN, Y. **Generalization and network design strategiestechnical report CRG-TR-89-4**. [s.l: s.n.].

LEE, H.; GROSSE, R.; RANGANATH, R.; NG, A. Y. Unsupervised learning of hierarchical representations with convolutional deep belief networks. **Communications of the ACM**, v. 54, n. 10, pp. 95-103, 2011.

LIMA, I.; PINHEIRO, C. A. M.; SANTOS, F. A. O. **Inteligência artificial**. Rio de Janeiro: Elsevier, 2014.

MCCULLOCH, W. S.; PITTS, W. A Logical Calculus of the Ideas Immanent in Nervous Activity. **The bulletin of mathematical biophysics**, v. 5, n. 4, pp. 115-133, 1943.

MIRZA, M.; OSINDERO, S. **Conditional generative adversarial nets**. [s.l: s.n.]. Disponível em: http://arxiv.org/abs/1411.1784. Acesso em: 27/05/2019.

MITCHELL, T. M. **Machine learning**. [s.l: s.n.]

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. **Foundations of machine learning**. Cambridge, MA: The MIT Press, 2012.

OHLSSON, S. **Deep learning:** how the mind overrides experience. Cambridge: Cambridge University Press, 2011.

PATTERSON, J.; GIBSON, A. **Deep Learning:** A practitioner's approach. Sebastopol, CA: O'Reilly Media, 2017.

PÉREZ CASTAÑO, A. **Practical artificial intelligence:** machine learning, bots, and agent solutions using C#. New York, NY: Apress, 2018.

RADFORD, A.; METZ, L.; CHINTALA, S. **Unsupervised representation learning with deep convolutional generative adversarial networks**. [s.l: s.n.]. Disponível em: http://arxiv.org/abs/1511.06434. Acesso em: 27/05/2019.

RUSSELL, S.; NORVIG, P. **Artificial intelligence:** a modern approach. 3ª ed. New Jersey: Pearson Education, 2010.

SIMEONE, O. **A brief introduction to machine learning for engineers**. London, England: King's College London, 2018.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to Sequence Learning with Neural Networks. *In*: NIPS'14 Proceedings of the 27th International Conference on Neural Information Processing Systems, Montreal, Canadá. **Anais**... Montreal, Canadá: 2014. Disponível em: http://arxiv.org/abs/1409.3215. Acesso em: 27/05/2019.

THE ASIMOV INSTITUTE. **The neural network zoo**. Disponível em: http://www.asimovinstitute.org/neural-network-zoo/. Acesso em: 27/5/2019.

THRUN, S. *et al*. Stanley: The robot that won the DARPA grand challenge. **Journal of Field Robotics**, v. 23, n. 9, pp. 661-692, 2006.

TURING, A. M. Computing Machinery and Intelligence. **MIND**, v. 59, n. 236, pp. 433-460, 1950.

VASILEV, I.; SLATER, D.; SPACAGNA, G.; ROELANTS, P.; ZOCCA, V. **Python deep learning:** exploring deep learning techniques and neural network architectures with pytorch, keras and tensorflow. 2ª ed. [s.l.] Packt Publishing, 2019.

VINYALS, O.; TOSHEV, A.; BENGIO, S.; ERHAN, D. **Show and tell:** a neural image caption generator. *In*: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA. **Anais**... Boston, MA: 2015.

WERBOS, P. J. **Backpropagation Through time:** what it does and how to do it. *In*: Proceedings of IEEE, **Anais**...1990.

YU, D.; DENG, L. Deep Learning and its applications to signal and information processing. **IEEE Signal Processing Magazine**, v. 28, n. 1, pp. 145-150, 2011.