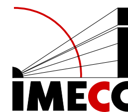
UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO CIENTÍFICA
DEPARTAMENTO DE MATEMÁTICA APLICADA



Carolina Rodrigues Menezes

Uma Introdução às Redes Neurais Profundas: Redes Convolucionais na Classificação de Imagens de Linfoblastos

Campinas
22/11/2024

Carolina Rodrigues Menezes

Uma Introdução às Redes Neurais Profundas: Redes Convolucionais na Classificação de Imagens de Linfoblastos*

Monografia apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas como parte dos requisitos para obtenção de créditos na disciplina Projeto Supervisionado, sob a orientação do(a) Prof. Dr. Marcos Eduardo Ribeiro do Valle Mesquita.

*Este trabalho foi financiado Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) através do Programa Institucional de Bolsas de Iniciação Científica (PIBIC) da Universidade Estadual de Campinas (Unicamp), quota 2024-2025.

Resumo

Neste projeto, estudamos conceitos fundamentais e relevantes de Aprendizado de Máquinas Profundo, dentro do contexto de Redes Neurais Convolucionais para Problemas de Classificação de Imagens. A base de dados escolhida é usada para a classificação de imagens de linfoblastos, que são células maléficas ligadas ao diagnóstico de leucemia linfóide aguda. Para completar nosso objetivo, exploramos diversos conceitos, como: Redes Neurais Artificiais; treinamento de redes neurais para classificação de imagens com Entropia Cruzada como função perda; métodos de otimização como Gradiente Descendente e o Adam; técnicas de pré-processamento como *One-Hot Encoding* e *Data Augmentation*; Camadas Convolucionais, de *Pooling* e *Dropout*. Usando Python e suas bibliotecas, incluindo Scikit-Learn e Keras, implementamos a leitura do banco de dados, a construção e o treinamento de um modelo de Rede Neural Convolucional, além da visualização de métricas de validação e teste do modelo e seus gráficos correspondentes. Por fim, com a arquitetura escolhida para a rede neural e os valores adequados de hiperparâmetros, obtivemos uma acurácia nos dados de teste de 98% e função perda na ordem de grandeza de 10^{-2}, o que configura um modelo de classificação de imagens viável e satisfatório para a classificação de imagens de linfoblastos.

Abstract

In this project, we study fundamental and relevant concepts of Deep Machine Learning within the context of Convolutional Neural Networks applied to Image Classification Tasks. The chosen dataset is used in the image classification of lymphoblasts, which are unhealthy cells linked to the diagnosis of acute lymphoblastic leukemia. In order to fulfill our objective, we explored several concepts, such as: Artificial Neural Networks; training of neural networks for classification tasks with Cross-Entropy as a cost function; optimization methods such as Gradient Descent and Adam; pre-processing techniques such as One-Hot Encoding and Data Augmentation; Convolutional, Pooling and Dropout layers. Using Python and its libraries, including Scikit-Learn and Keras, we implemented a code to read and load the dataset, build and train a Convolutional Neural Network model, as well as visualize the model's validation and testing metrics and the corresponding graphs. Finally, with the chosen neural network architecture and adequate values for its hyperparameters, we achieved in the testing data an accuracy of 98% and a loss function near the magnitude of 10^{-2}, which characterizes an image classification model that is viable and satisfactory for the image classification of lymphoblasts.

Conteúdo

1 Introdução **6**

2 Conceitos Fundamentais de Aprendizado Profundo **6**

2.1 Redes Neurais em Problemas de Classificação 6

2.2 Particionando o banco de dados 8

2.3 *One-Hot Encoding* 9

2.4 Função Custo: *Cross Entropy* 9

2.5 Métodos de Otimização 10

2.5.1 Gradiente Descendente 10

2.5.2 Gradiente Descendente em Lotes e Estocástico 11

2.5.3 Adam 11

2.6 Modelo de Rede Neural Sequencial no Keras 12

2.7 Camada *Dropout* 14

2.8 Camadas Convolucionais 15

2.9 Camadas de *Max-Pooling* 18

2.10 *Data Augmentation*: Aumentando o conjunto de dados 19

3 Resultados **20**

4 Conclusão **28**

1 Introdução

Redes Neurais Profundas são modelos de Aprendizado de Máquinas que buscam realizar abstrações de alto nível de dados, utilizando diversas camadas de processamento. Em particular, Redes Neurais Convolucionais são especialmente boas em problemas de classificação de imagens, pois se inspiram no entendimento biológico do cortéx visual do cérebro.

Neste trabalho, detalharemos e implementaremos diversos conceitos necessários e úteis para a construção e uso de um modelo de rede neural convolucional capaz de diferenciar imagens de células saudáveis e linfoblastos — assim, auxiliando no diagnóstico de leucemia linfoblástica aguda.

2 Conceitos Fundamentais de Aprendizado Profundo

Nesta seção, detalharemos todos os conceitos de Aprendizado de Máquinas utilizados na construção do projeto, bem como suas implementações no Scikit-Learn, uma biblioteca em Python para Aprendizado de Máquinas, e no Keras, uma biblioteca em Python para Redes Neurais (Pedregosa et al., 2011; Chollet et al., 2015).

2.1 Redes Neurais em Problemas de Classificação

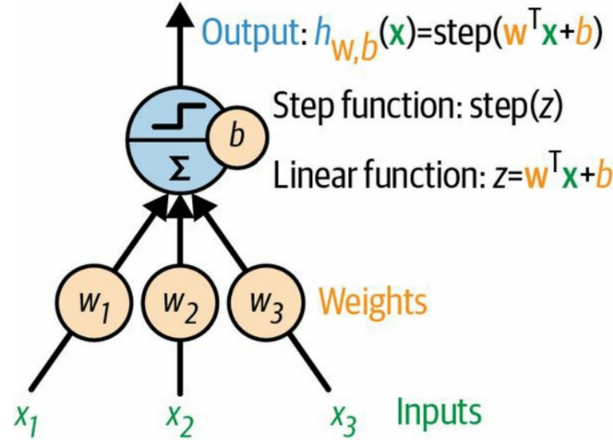
Redes Neurais Artificiais (em inglês, *Artificial Neural Networks*: ANNs) são estruturas computacionais formadas por uma rede com diversos neurônios artificiais conectados entre si, com lógica inspirada nas sinapses de neurônios biológicos. Em uma rede neural artificial, a cada neurônio está associado ao menos uma entrada e uma saída.

Os neurônios são modelados de acordo com o chamado Perceptron, ilustrado na Figura 1, cujas entradas e saídas são números, e a cada conexão sináptica está associada um peso w_i. Assim, dado um neurônio com n entradas, primeiro é computada uma combinação linear destas

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n + b, \tag{1}$$

onde x_i são os valores de entrada para cada parâmetro modelado pela rede e b é um termo

Figura 1: Diagrama do Perceptron.



Fonte: Géron (2022)

de viés. Depois, aplica-se uma função de ativação ϕ a essa entrada, gerando uma saída da forma:

$$h_{w,b}(x) = \phi(w^T x + b). \tag{2}$$

Quando uma rede neural apresenta outras camadas entre a de entrada e a de saída, denominamos estas como camadas ocultas, e a rede neural totalmente conectada é chamada Perceptron Multicamadas (MLP), conforme ilustrado na Figura 2.

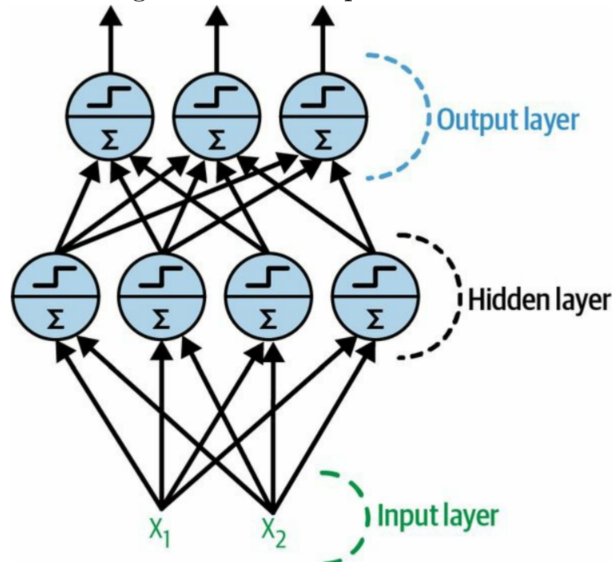
As redes MLPs podem ser usadas tanto para problemas de regressão (modelos que preveem valores), quanto para problemas de classificação (modelos que preveem classes). Neste estudo, estaremos interessados em problemas de classificação binária, ou seja, problemas em que o conjunto de imagens com o qual trabalharemos cairá em uma de duas classes exclusivas, denominadas 0 e 1.

Para isso, na camada de saída da rede neural, usaremos a função de ativação softmax, definida por:

$$\hat{p}_k = \frac{exp(s_k(x))}{\sum_{j=1}^{K} exp(s_j(x))}, \quad k = 1, ..., K, \tag{3}$$

onde $\hat{p}_k$ é a probabilidade que a amostra x terá de pertencer à classe k, $s_k(x)$ é o *score* da amostra x na classe k e K é o número de classes. Para obter, finalmente, a classe que o modelo prevê (ou seja, que o modelo calcula como tendo a maior probabilidade) para

Figura 2: Diagrama do Perceptron Multicamadas.



Fonte: Géron (2022)

uma amostra x, usamos a equação:

$$\hat{y} = \arg\max_k \hat{p}_k \equiv \arg\max_k s_k(x). \tag{4}$$

2.2 Particionando o banco de dados

Para construir, treinar e avaliar um modelo de aprendizado de máquinas, antes precisamos dividir o nosso banco de dados em ao menos três subconjuntos: treino, validação e teste.

O conjunto de treinamento e o conjunto de validação serão usados durante o treinamento do modelo: o primeiro servirá para o treino em si, enquanto o segundo será usado para avaliar o modelo durante esta etapa. O conjunto de teste deve conter exclusivamente dados que não foram vistos em nenhuma etapa do treinamento, de forma que seja possível testar como o modelo já treinado se comporta com novas amostras.

Usando o Scikit-Learn, podemos particionar o banco de dados da seguinte forma:

```
Xtrf, Xte, ytrf, yte = train_test_split(Xr_rgb, y,
                           test_size=0.2, random_state=42)
Xtr, Xval, ytr, yval = train_test_split(Xtrf, ytrf,
                           test_size = 0.2, random_state = 42)
```

onde o primeiro comando divide o conjunto completo de dados (Xr_rgb, y) no conjunto de treino completo (Xtrf, ytrf) com 80% dos dados e no conjunto de teste (Xte, yte) com 20% dos dados. O segundo comando é similar, dividindo o conjunto de treino completo no conjunto de treino (Xtr, ytr) com 80% dos dados e no conjunto de validação (Xval, yval) com 20% dos dados. O argumento `random_state = 42` é usado para assegurar a reprodutibilidade dos resultados.

2.3 *One-Hot Encoding*

Nesta técnica, transformamos um valor que representa a classe de uma amostra em um vetor *one-hot*. Em resumo, buscamos escrever para a classificação binária:

$$0 \equiv \begin{bmatrix} 1 & 0 \end{bmatrix} \text{ e } 1 \equiv \begin{bmatrix} 0 & 1 \end{bmatrix}. \tag{5}$$

Obtemos este processamento com o comando `keras.utils.to_categorical(,)` no Keras, onde o primeiro argumento é o vetor ou matriz a ser processado e o segundo argumento é o número de classes (nesse caso, duas) a serem codificadas. Este processo é importante para as manipulações matemáticas que serão feitas na implementação do modelo.

2.4 Função Custo: *Cross Entropy*

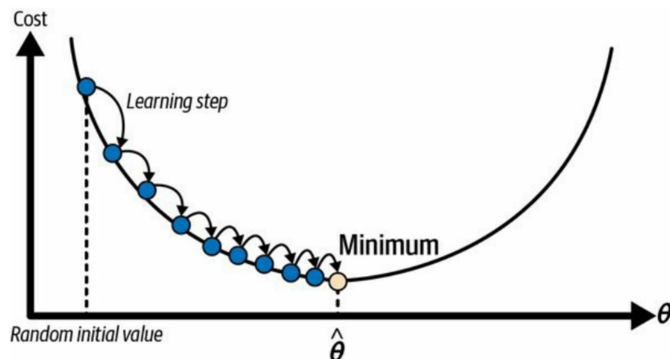
No treinamento de uma rede neural para classificação de imagens, dado um conjunto de treino $\tau = \{(x_i, y_i) : i = 1, ..., m\}$, queremos otimizar um problema do tipo:

$$\underset{\Theta}{\text{minimize}}\ J(\Theta) = \underset{\Theta}{\text{minimize}}\ -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(\hat{p}_k^{(i)}), \tag{6}$$

onde $J(\Theta)$ é a função custo que chamamos de *Cross Entropy* (entropia cruzada), $\hat{p}_k^{(i)}$ é a probabilidade calculada (Equação 3) de que a amostra i pertença à classe k, $y_k^{(i)}$ é o valor real da classe da amostra, K é o número de classes e Θ é a matriz de parâmetros. O objetivo do treinamento de uma rede neural de classificação é obter a matriz de parâmetros Θ, que é a junção de todos os parâmetros da rede neural (vieses e pesos sinápticos).

Podemos calcular o gradiente de $J(\Theta)$ usando o algoritmo *backpropagation* (Géron, 2022) ou usando alguma derivação automática disponível em softwares de apren-

Figura 3: Evolução do Gradiente Descendente segundo a atualização de θ.



Fonte: Géron (2022)

dizado profundo, para em seguida usar algum algoritmo de otimização para achar a matriz de parâmetros Θ que minimiza esta função custo, como o Gradiente Descendente ou Adam.

2.5 Métodos de Otimização

2.5.1 Gradiente Descendente

O Gradiente Descendente é um algoritmo de otimização bastante comum, capaz de solucionar uma vasta gama de problemas. A ideia principal do algoritmo é ajustar os parâmetros iterativamente de forma a minimizar a função custo. Para cada iteração, calcula-se o gradiente da função em relação ao parâmetro θ e é feito um avanço na direção contrária. Como o gradiente de uma função é justamente o vetor que indica a direção na qual o deslocamento aumenta o valor da função, a direção contrária o diminuirá. Uma vez que o gradiente calculado na iteração for zero, encontramos um ponto crítico da função, que pode ser um mínimo local.

O método começa escolhendo valores aleatórios para preencher o vetor de parâmetros θ (o que chamamos de *random initialization*, ou seja, inicialização aleatória), e é atualizado segundo a Equação 7, com um passo que devemos definir como um hiperparâmetro (que diferentemente dos parâmetros do modelo, devem ser definidos externamente a ele) que chamamos *learning rate* η, ou passo de aprendizagem:

$$\theta_{i+1} = \theta_i - \eta \nabla_{\theta_i} J(\Theta). \tag{7}$$

Se este passo for pequeno demais, pode levar muito tempo para chegar próximo a um mínimo local, correndo o risco do método atingir o limite de iterações ou de tempo de execução antes de convergir; alternativamente, se for grande demais, corre o risco de "pular" o mínimo local de uma iteração para a próxima, fazendo com que o algoritmo acabe divergindo— na Figura 3, podemos ver o comportamento da convergência quando um passo η apropriado é utilizado. É também importante reescalar os valores dos parâmetros para que o algoritmo convirja mais rapidamente.

2.5.2 Gradiente Descendente em Lotes e Estocástico

Uma alternativa comum e mais rápida por iteração ao Método do Gradiente Descendente é utilizar o Método do Gradiente Descendente em Lotes, que a cada iteração, ao invés de usar todo o conjunto de treinamento para calcular o gradiente, usa um subconjunto menor aleatório. Define-se, então, o número de lotes (*batches*) a serem usados em cada época e seus tamanhos.

Outra alternativa comum é o Método do Gradiente Descendente Estocástico, que escolhe uma amostra aleatória do conjunto de treino a cada iteração e computa os gradientes com base apenas nesta amostra. É mais rápido que os dois outros métodos até agora apresentados e é menos susceptível a estagnação em mínimos locais, mas costuma ser mais irregular e corre maior risco de não convergir.

2.5.3 Adam

O otimizador Adam (*adaptive moment estimation)* combina diversas técnicas de otimização, sendo um método de Gradiente Descendente Estocástico em Lotes baseado na adaptação de acordo com o histórico de gradientes anteriores para acelerar a convergência, com o uso de técnicas para ajuste de escala e de taxa de aprendizagem. É um otimizador robusto, descrito pelo seguinte algoritmo:

1. $m \leftarrow \beta_1 m - (1 - \beta_1)\nabla_\theta J(\theta)$;

2. $s \leftarrow \beta_2 s + (1 - \beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$;

3. $\hat{m} \leftarrow \frac{m}{1-\beta_1^i}$;

4. $\hat{s} \leftarrow \frac{s}{1-\beta_2^i}$;

5. $\theta \leftarrow \theta + \eta\hat{m} \oslash \sqrt{\hat{s} + \epsilon}$;

onde i representa o número da iteração, m é o vetor momento inicializado nulo, que atualiza os pesos, fazendo com que o gradiente atue como "aceleração", não como "velocidade", s é o vetor que acumula a raiz dos gradientes também inicializado nulo, β_1 é um hiperparâmetro de momento com valor típico de 0.9, β_2 é o *decay rate* (taxa de decaimento) com valor típico de 0.999, ϵ é um parâmetro de suavização para evitar divisões por zero com valor típico de 10^{-7}, η é a taxa de aprendizado com valor típico de 0.001 e os símbolos $\otimes$ e $\oslash$ representam, respectivamente, multiplicação e divisão elemento a elemento.

Por ser mais robusto e frequentemente mais fácil de usar que os métodos de Gradiente Descendente inicialmente apresentados (por precisar de menos ajuste dos hiperparâmetros), esse será o otimizador utilizado em nosso modelo.

2.6 Modelo de Rede Neural Sequencial no Keras

Uma rede neural sequencial é o mais simples modelo a ser construído na biblioteca Keras, que conecta as camadas do modelo de forma sequencial, ou seja, as saídas de uma camada são diretamente conectados às entradas da camada seguinte. Para criar uma ANN sequencial no Keras, podemos passar as camadas dentro de um objeto `keras.Sequential`:

```
model = keras.Sequential([
    keras.Input(shape = [28,28]),
    keras.layers.Flatten(),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Na primeira camada, inserimos uma camada de entrada, que recebe o tamanho das entradas a serem passadas para determinar o tamanho da matriz de pesos de conexão da primeira camada oculta.

A segunda camada é uma camada *Flatten*, que não ajusta nenhum parâmetro,

Figura 4: Tabela com detalhes do modelo do exemplo.

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)            Output Shape          Param #
=================================================================
flatten (Flatten)       (None, 784)           0

dense (Dense)           (None, 300)           235500

dense_1 (Dense)         (None, 100)           30100

dense_2 (Dense)         (None, 10)            1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Fonte: Géron (2022)

apenas converte uma imagem de entrada em um vetor 1D: por exemplo, se receber um *mini-batch* de tamanho [32, 28, 28], sua saída será do tamanho [32, 28*28].

A terceira camada é uma camada densa. Camadas densas têm todos os seus neurônios conectados a todos os neurônios da camada anterior. Esta usa a função de ativação ReLU, dada por:

$$f(x) = \begin{cases} x, & \text{se } x > 0 \\ 0, & \text{se } x \leq 0 \end{cases} \tag{8}$$

Cada camada densa tem sua prória matriz de pesos de conexão entre seus neurônios e entradas, além do vetor de viés (um por neurônio). Ao receber dados de entrada, computa a Equação 2.

A quarta camada, que é uma camada densa de saída, possui 10 neurônios (um para cada classe) e usa a função de ativação softmax, como na Equação 3.

Usando o comando `model.summary()`, temos um resumo da rede, como mostra a Figura 4, que contém todas as camadas, o tamanho de suas saídas e o número de parâmetros. Aqui, *None* significa que o tamanho da *batch* pode ser qualquer um, a ser definido posteriormente. Neste exemplo, o número de parâmetros é bem elevado, o que dá ao modelo bastante flexibilidade para ajustar os dados de treinamento, mas arrisca também o *overfitting*.

Para compilar o modelo, usamos o comando:

```python
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Aqui, definimos a função custo a ser utilizada, o método de otimização dela e uma métrica para avaliar o modelo durante o treinamento e a avaliação.

Para de fato treinar, usamos:

```python
history = model.fit(X_train, y_train, batch_size = 32, epochs=30,
                    validation_data=(X_valid, y_valid))
```

Aqui, especificamos o conjunto de dados a ser usado durante o treinamento, o tamanho dos lotes, o número de épocas e o conjunto de dados a ser usado para validação. Ao fim de cada época, será computada a função custo e quaisquer outras métricas que forem especificadas para ambos conjuntos de treino e validação.

Para enfim avaliar o modelo no conjunto de teste, usamos:

```python
score = model.evaluate(Xte, yte, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

Podemos também fazer gráficos das curvas das métricas que calcularmos ao longo das épocas. Para fazer novas previsões (no conjunto de teste, por exemplo), usamos o comando `model.predict()`, passando como argumento as amostras do conjunto que queremos prever.

2.7 Camada *Dropout*

O *Dropout* é uma técnica de regularização popular para redes neurais profundas com algoritmo simples: a cada passo do treinamento, todos os neurônios tem uma probabilidade p de serem temporariamente "deixados de lado", ou seja, durante este passo do treinamento, serão completamente ignorados, mas podem voltar a serem ativos no passo seguinte. O hiperparâmetro p é chamado taxa de *dropout*, com valor típico de 50%. Após o treino, os neurônios funcionam normalmente.

Por mais simples que seja a ideia, este algoritmo força os neurônios com ele treinados a não dependerem excessivamente de neurônios vizinhos ou de poucos neurônios, ou seja, são levados a serem tão úteis sozinhos quanto possível e a prestar atenção a cada

uma de suas entradas. Assim, geramos uma rede mais robusta que generaliza melhor, sendo menos sensível a pequenas mudanças de entrada.

No Keras, o *Dropout* é implementado com `keras.layers.Dropout()`, com o argumento sendo a taxa de *dropout* p. Esta regularização será aplicada aos neurônios da camada seguinte à camada de *dropout*. Se o modelo estiver *overfitting* o conjunto de treino, pode ser útil aumentar p; assim como pode ser útil diminuir p se estiver *underfitting*. O seu uso pode retardar a convergência, mas geralmente resulta em um modelo de bem maior qualidade, o que justifica o tempo extra utilizado.

2.8 Camadas Convolucionais

Redes Neurais Convolucionais (do inglês, Convolutional Neural Networks: CNNs) são inspiradas no estudo do cortéx visual do cérebro, e têm sido utilizadas em reconhecimento computacional de imagens desde a década de 80 (Géron, 2022). Muitos neurônios no córtex visual têm um pequeno campo receptor local, ou seja, só reagem à estímulos localizados em uma região limitada do campo de visão. Os campos receptores de diferentes neurônios se sobrepõem, compondo em conjunto o campo de visão inteiro. Essa observação biológica originou a ideia de neurônios de camadas maiores serem conectados apenas a neurônios na vizinhança da camada anterior.

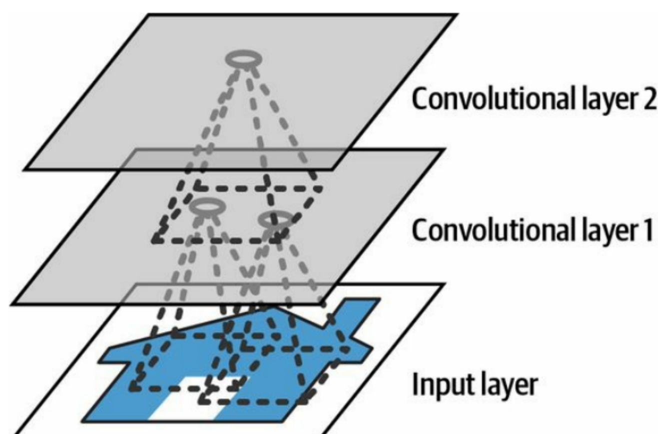
Assim, em camadas convolucionais, os neurônios não estão conectados a todos os pixels da imagem que recebe como entrada, mas apenas aos que estão em seus campos receptores, como mostra a Figura 5.

Essa arquitetura permite que a CNN se concentre em características mais simples, de baixo nível, nas primeiras camadas e congregá-las em características mais complexas, de alto nível, nas camadas seguintes. Um neurônio localizado numa linha i, coluna j de uma dada camada está conectado às saídas de neurônios da camada anterior localizados nas linhas i até $i + f_h - 1$, colunas j até $j + f_w - 1$, onde f_h e f_w são a altura e largura do campo receptor, como ilustrado na Figura 6.

O "*zero padding*" é a técnica de adicionar zeros ao redor das entradas, caso seja desejável que a próxima camada tenha mesmo tamanho que a anterior.

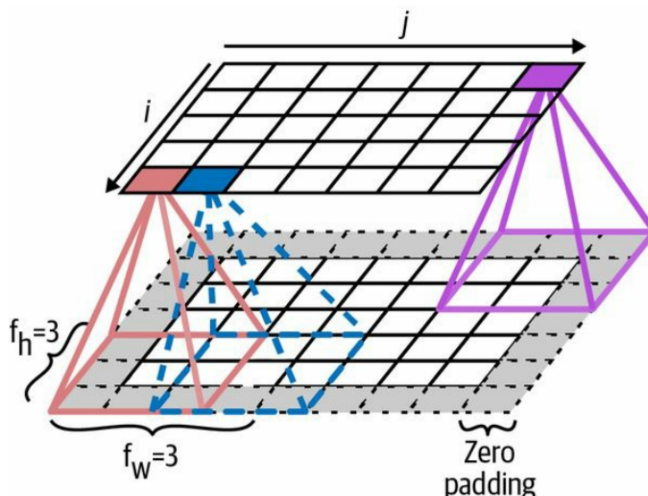
O conjunto de pesos de um neurônios em uma camada convolucional é chamado filtro (ou kernel de convolução). Tudo o que estiver fora do campo receptor descrito pelo

Figura 5: Conexão de neurônios em campos receptores de camadas convolucionais.



Fonte: Géron (2022)

Figura 6: Diagrama do campo receptor.



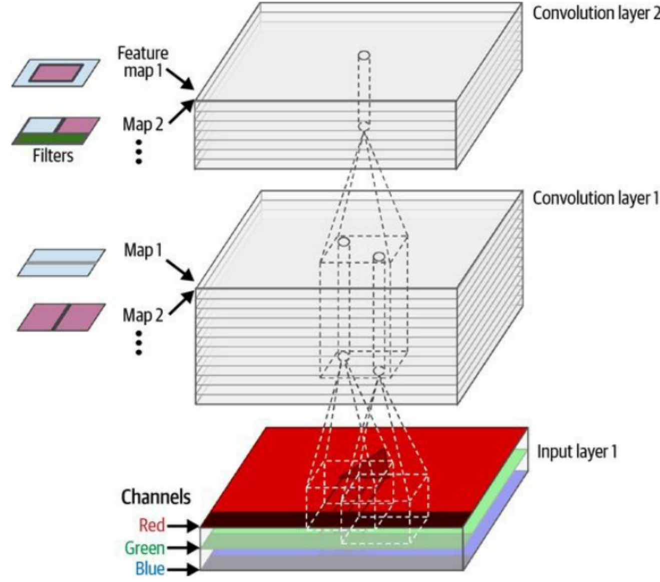
Fonte: Géron (2022)

filtro será ignorado: para um filtro que é uma matriz quadrada preenchida por zeros, com a coluna central preenchida por 1, todas as entradas fora da linha central vertical serão ignoradas pelo neurônios.

Na prática, camadas convolucionais têm tantos filtros quanto quisermos e geram um mapa de características por filtro, então são melhor representados como objetos 3D conforme a Figura 7. Cada imagem de entrada é composta também de subcamadas, com uma cor por canal. Para imagens coloridas em RGB, por exemplo, são três canais, logo três subcamadas.

Em suma, uma camada convolucional aplica simultaneamente múltiplos filtros

Figura 7: Visualização 3D de camadas convolucionais.



Fonte: Géron (2022)

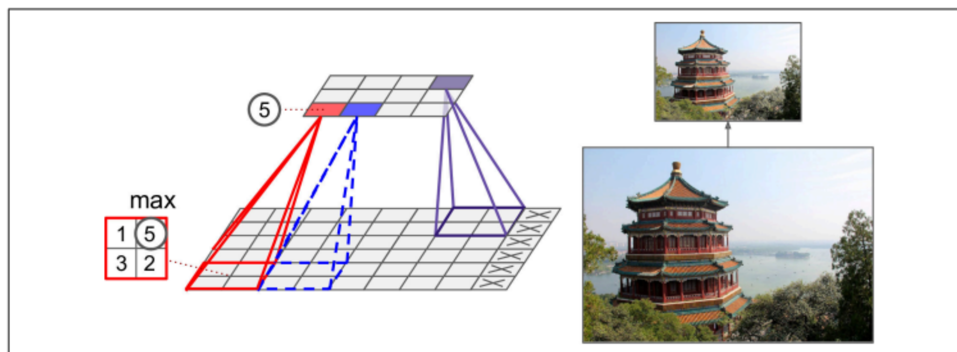
treináveis às suas entradas, o que o torna capaz de detectar múltiplos parâmetros em qualquer lugar de suas entradas. Na equação 9, podemos ver como é computada a saída de um dado neurônio numa camada convolucional:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k'}, \quad \text{com} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}, \tag{9}$$

onde $\phi(z_{i,j,k})$ é a saída do neurônio na linha i, coluna j no mapa de características k da camada convolucional, s_h e s_w são os *strides*, f_h e f_w são a altura e largura do campo receptor, $f_{n'}$ é o número de mapas de características da camada anterior, $x_{i',j',k'}$ é a saída do neurônio da camada anterior, localizado na linha i', coluna j' e mapa de características (ou canal, se a camada anterior é a de entrada) k', b_k é o termo de viés para o mapa de características k na camada atual, $w_{u,v,k'}$ é o peso de conexão entre qualquer neurônio no mapa de características k da camada atual e sua entrada localizada na linha u, coluna v (em relação ao campo receptor do neurônio) e mapa de características k' e $\phi : \mathbb{R} \to \mathbb{R}$ é a função de ativação do neurônio.

Implementamos a camada convolucional utilizando `keras.layers.Conv2D(,)` no Keras, onde o primeiro argumento deve ser o número de filtros da camada, enquanto

Figura 8: Funcionamento de uma camada de *max-pooling*.



Fonte: Géron (2022)

o segundo argumento é um vetor com a altura e largura do kernel, além de outros parâmetros.

2.9 Camadas de *Max-Pooling*

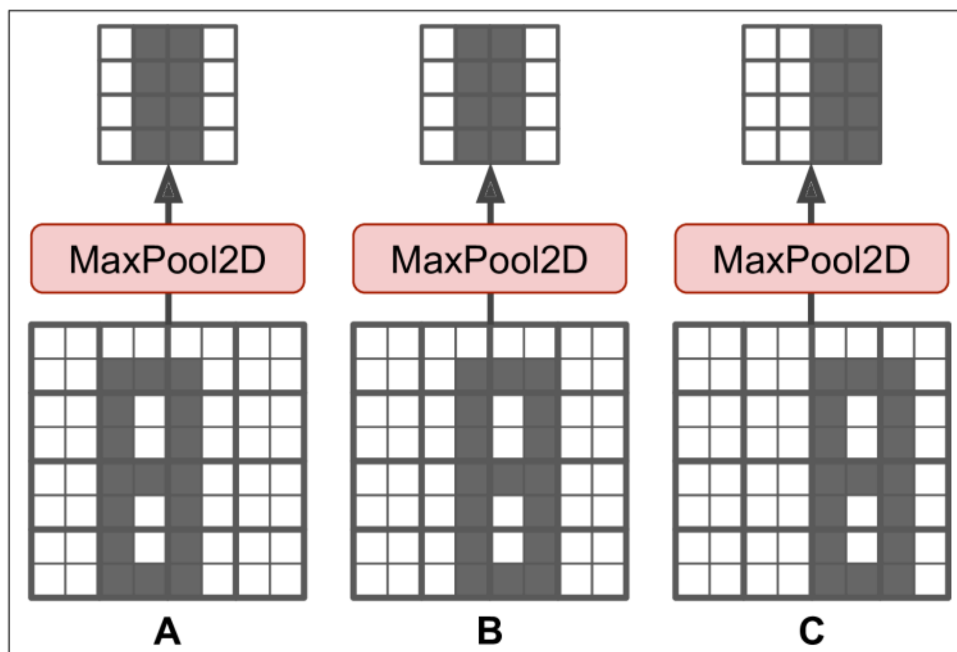
Camadas de *Pooling* tem como objetivo fazer uma sub-amostragem da imagem de entrada, de forma a reduzir sua dimensão — consequentemente reduzindo também o custo computacional, uso de memória e o número de parâmetros (reduzindo, assim, o risco de *overfitting*). Assim como as camadas convolucionais, cada neurônio da camada de *pooling* está conectado à saída de um número limitado de neurônios da camada anterior, localizados dentro de um pequeno campo receptor retangular de dimensão fixa. Porém, o neurônio de *pooling* não possui parâmetros treináveis ou pesos, ele apenas agrupa as entradas usando alguma função de máximo ou média. Aqui, usaremos apenas a camada *max-pooling*, que propaga a imagem de acordo com a Figura 8.

Além de diminuir custo computacional, uso de memória e número de parâmetros, a camada de *max-pooling* introduz um pouco de invariância a pequenas translações e rotações, tal como mostrado na Figura 9. Dessa forma, essa camada pode ser útil em problemas que não deveriam depender destes detalhes, como em problemas de classificação.

Tipicamente, numa arquitetura de rede neural convolucional, empilhamos camadas convolucionais e de *max-pooling* de forma alternada, com uma camada de saída final que fará as previsões das probabilidades de cada classe.

Implementamos as camadas de *max-pooling* no Keras utilizando o comando `keras.layers.MaxPool2D()`, onde o argumento a ser passado deve ser um vetor repre-

Figura 9: Demonstração da invariância introduzida pela camada *max-pooling*.



Fonte: Géron (2022)

sentando o tamanho do kernel da camada.

2.10 *Data Augmentation*: Aumentando o conjunto de dados

Data Augmentation, ou aumento do conjunto de dados, é uma técnica para gerar novos dados de maneira artificial a partir dos dados já existentes, aplicando transformações de forma aleatória às imagens. Aqui, usaremos as camadas de espelhamento, rotação, zoom e translação:

- A camada `keras.layers.RandomFlip` decide aleatoriamente se espelha a imagem de entrada na horizontal ou na vertical;
- A camada `keras.layers.RandomRotation()` decide aleatoriamente a que grau rotaciona a imagem de entrada, com o argumento sendo uma porcentagem de 2π a ser usada como limite para o ângulo de rotação positivo e negativo;
- A camada `keras.layers.RandomZoom()` decide aleatoriamente qual a porcentagem do zoom feito na imagem de entrada, com o argumento sendo usado como limite para a aproximação ou afastamento da imagem;

- A camada `keras.layers.RandomTranslation(,)` decide aleatoriamente qual a porcentagem da imagem transladar, com o primeiro argumento sendo o limite positivo e negativo para translação na vertical, enquanto o segundo argumento é para a translação na horizontal;

Assim, podemos complementar o treinamento do modelo com pequenas variações nos dados de entrada, melhorando a precisão de suas previsões e tornando-o mais capaz de fazer uma boa generalização e atuar bem em dados nunca antes vistos por ele.

3 Resultados

Com os conceitos descritos na seção anterior, podemos implementar a construção, o treinamento, e a avaliação de um modelo de rede neural convolucional para a classificação de imagens em um ambiente do *Google Colab*.

O banco de dados que usaremos é o "ALL-IDB: *Acute Lymphoblastic Leukemia Image Database for Image Processing*" de Fabio Scotti e colaboradores, do *Department of Information Technology - Università degli Studi di Milano*. Neste banco de dados, temos 260 imagens, das quais 130 são classificadas como células saudáveis (classificação 0) e 130 são linfoblastos (classificação 1). Todos os arquivos de imagem são nomeados na notação ImXXX_Y.jpg, onde XXX é um número inteiro de 3 dígitos progressivo e Y é um dígito booleano igual a 0 ou 1 (Labati et al., 2011).

O objetivo final do nosso modelo é conseguir prever corretamente qual a classificação — célula saudável ou linfoblasto — da imagem apresentada.

Na primeira seção do código, preparamos o acesso à pasta do *Google Drive* onde se encontra o banco de dados do problema e importamos as bibliotecas necessárias, como Matplotlib, Numpy, Scikit-Learn, Keras, entre outros:

```
1 from google.colab import drive
2 drive.mount('/content/gdrive')
3 import os
4 import glob
5 import cv2
6 import time
7 import matplotlib.pyplot as plt
```

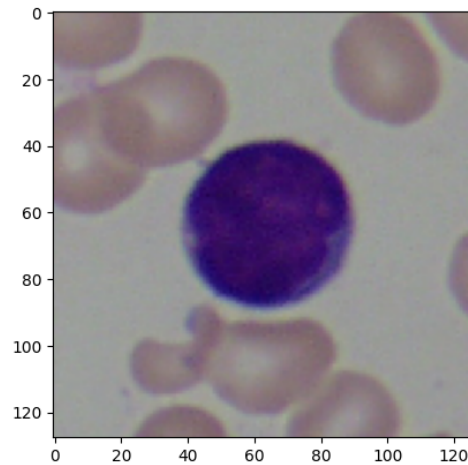
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import keras
from keras import layers
```

Na segunda seção, carregamos o banco de dados e preparamos a matriz Xr_rgb para receber as imagens e o vetor y para receber a classificação destas imagens. Em seguida, redimensionamos as imagens para o tamanho de 128 pixels x 128 pixels e reescalamos os valores dos canais de cor RGB, dividindo-os por 255, obtendo assim para cada pixel três números do tipo *float* entre 0 e 1. Como saída, podemos checar que o número total de imagens no banco original é de 260, dos quais 130 são linfoblastos.

```
image_size = (128,128)
dataset_path = "/content/gdrive/MyDrive/IC/ALL_IDB2/img/*.tif"

imgs_list = [f for f in glob.glob(dataset_path)]
N = len(imgs_list)
print("Number of images:",N)

Xr_rgb = np.zeros((N,image_size[0],image_size[1],3))
y = np.zeros((N,))

z = np.zeros((image_size[0],image_size[1],1))
for i,img_path in enumerate(imgs_list):
    img = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)
    img_resized = cv2.resize(img, image_size)

    # RGB images
    Xr_rgb[i,:,:,:] = img_resized/255

    ind = img_path.find(".tif")
    y[i] = int(img_path[ind-1])

print("Number of lymphoblasts:",int(np.sum(y)))
```

Em seguida, podemos visualizar na Figura 10 a primeira amostra do banco de dados com o comando `plt.imshow(Xr_rgb[0,:,:,:])`.

Figura 10: Imagem da primeira amostra do banco de dados completo.



Fonte: Elaboração própria.

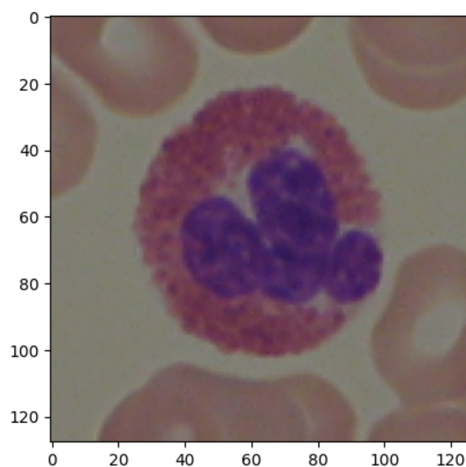
Na próxima seção do código, realizamos a repartição do banco de dados, descrita na Subseção 2.2:

```
Xtrf, Xte, ytrf, yte = train_test_split(Xr_rgb, y, test_size=0.2,
    random_state=42)
Xtr, Xval, ytr, yval = train_test_split(Xtrf, ytrf, test_size = 0.2,
    random_state = 42)
```

Em seguida, implementamos e visualizamos a ação da técnica da Subseção 2.10 de aumento de dados.

```
data_augmentation_layers = [
    layers.RandomFlip(), #aleatoriamente decide se espelha a imagem
    layers.RandomRotation(0.3), #aleatoriamente decide se rotaciona a
    imagem em ate +-30%*2pi
    layers.RandomZoom(0.2), #aleatoriamente decide se da zoom de ate
    +-20%
    layers.RandomTranslation(0.1, 0.1) #aleatoriamente decide se
    translada a imagem (vertical e horizontal) em ate 10%
]

def data_augmentation(images):
  for layer in data_augmentation_layers:
    images = layer(images)
  return images

```

Figura 11: Visualização da terceira amostra do conjunto de treino.



Fonte: Elaboração própria.

```
13 plt.figure(figsize=(5, 5))
14 for i in range(9):
15   augmented_images = data_augmentation(Xtr[:5])
16   ax = plt.subplot(3, 3, i + 1)
17   plt.imshow(np.array(augmented_images[2]))
18   plt.axis("off")
```

Aqui, o aumento de dados é aplicado à terceira amostra do conjunto de treino, ilustrada na Figura 11, cujo resultado após 9 repetições é visto na Figura 12.

Depois, preparamos alguns pré-processamentos, definindo o número de classes para o problema e o tamanho da entrada do modelo:

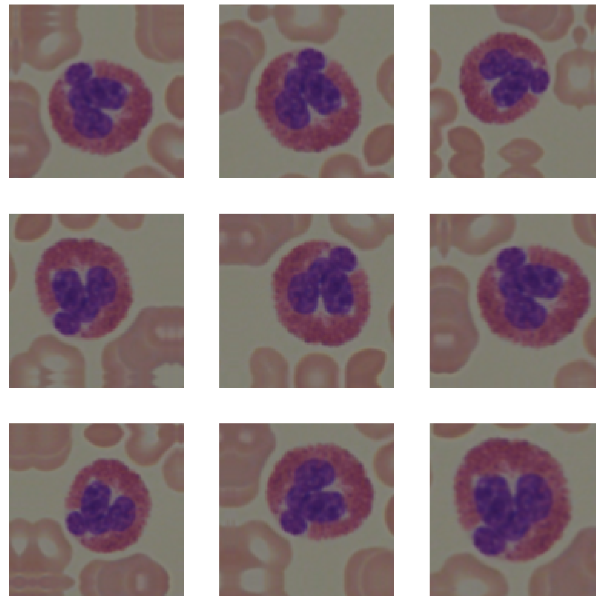
```
1 #parametros
2 num_classes = 2 #linfoblasto (1) ou saudavel (0)
3 input_shape = (128, 128, 3) #n de pixels, n de pixels, n de canais rgb
```

Então, fazemos o processamento do *one-hot encoding* descrito na Subseção 2.3.

```
1 #one hot encoding: transforma cada valor do vetor de classificacoes em
    uma matriz (cada linha composta pela probabilidade de 0 e 1)
2 ytr = keras.utils.to_categorical(ytr, num_classes)
3 yte = keras.utils.to_categorical(yte, num_classes)
4 yval = keras.utils.to_categorical(yval, num_classes)
```

Na sexta seção do código, construímos, enfim, o nosso modelo da rede neural convolucional, descrevendo cada uma de suas camadas. Podemos visualizar detalhes ao

Figura 12: Resultado de nove repetições da aplicação do aumento de dados construído no código.



Fonte: Elaboração própria.

fim em uma tabela com o `model.summary()` na Figura 13.

```
model = keras.Sequential([
    keras.Input(shape = input_shape),

    layers.RandomFlip(),
    layers.RandomRotation(0.3),
    layers.RandomZoom(0.2),
    layers.RandomTranslation(0.1, 0.1),

    layers.Conv2D(32, kernel_size = (3,3), activation = 'relu'),
    layers.MaxPooling2D(pool_size = (2,2)),
    layers.Conv2D(64, kernel_size = (3,3), activation = 'relu'),
    layers.MaxPooling2D(pool_size = (2,2)),
    layers.Conv2D(128, kernel_size = (3,3), activation = 'relu'),
    layers.MaxPooling2D(pool_size = (2,2)),
    layers.Conv2D(128, kernel_size = (3,3), activation = 'relu'),
    layers.MaxPooling2D(pool_size = (2,2)),

    layers.Flatten(),
    layers.Dropout(0.5),
```

Figura 13: Tabela descritiva do modelo construído.

Layer (type)	Output Shape	Param #
random_flip_1 (RandomFlip)	(None, 128, 128, 3)	0
random_rotation_1 (RandomRotation)	(None, 128, 128, 3)	0
random_zoom_1 (RandomZoom)	(None, 128, 128, 3)	0
random_translation_1 (RandomTranslation)	(None, 128, 128, 3)	0
conv2d (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 128)	0
conv2d_3 (Conv2D)	(None, 12, 12, 128)	147,584
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 128)	0
flatten (Flatten)	(None, 4608)	0
dropout (Dropout)	(None, 4608)	0
dense (Dense)	(None, 2)	9,218

Total params: 250,050 (976.76 KB)
Trainable params: 250,050 (976.76 KB)
Non-trainable params: 0 (0.00 B)

Fonte: Elaboração própria.

```
20
21     layers.Dense(num_classes, activation = 'softmax')
22 ])
23 model.summary()
```

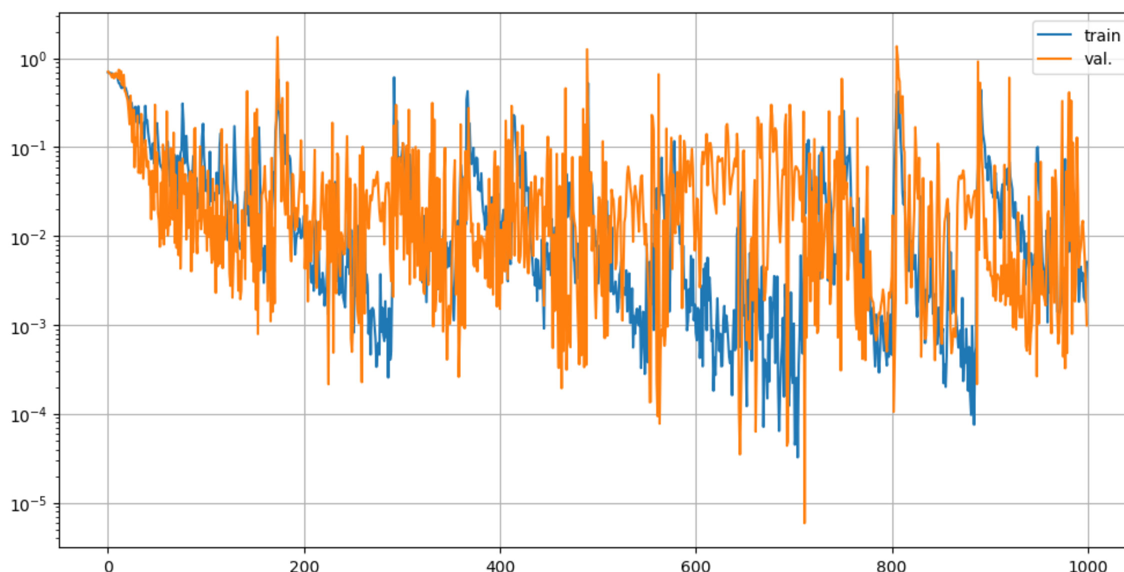
A sétima seção do código mostra o treinamento do modelo, onde definimos o tamanho dos *batches* como 32, que deverão ser treinados ao longo de 1000 épocas. Neste modelo, percebemos que estes são bons números para garantir uma acurácia final do modelo bastante elevada com frequência. Na saída da célula, é calculado para cada época os valores da função custo e da acurácia para o conjunto de treino e o de validação.

```
1 batch_size = 32
2 epochs = 1000
3 model.compile(loss = 'categorical_crossentropy', optimizer = 'adam',
    metrics = ['accuracy'])
4 hist=model.fit(Xtr, ytr, batch_size=batch_size, epochs=epochs,
    validation_data = (Xval, yval))
```

Na penúltima seção, temos a avaliação do modelo, onde visualizamos os gráficos da função custo e da acurácia dos conjuntos de treino e validação ao longo das épocas durante o treinamento, mostrados nas Figuras 14 e 15, respectivamente:

```
1 #para o grafico da funcao perda
```

Figura 14: Imagem gerada pelo código para as curvas da função perda durante o treinamento nos conjuntos de treino (em azul) e validação (em laranja)



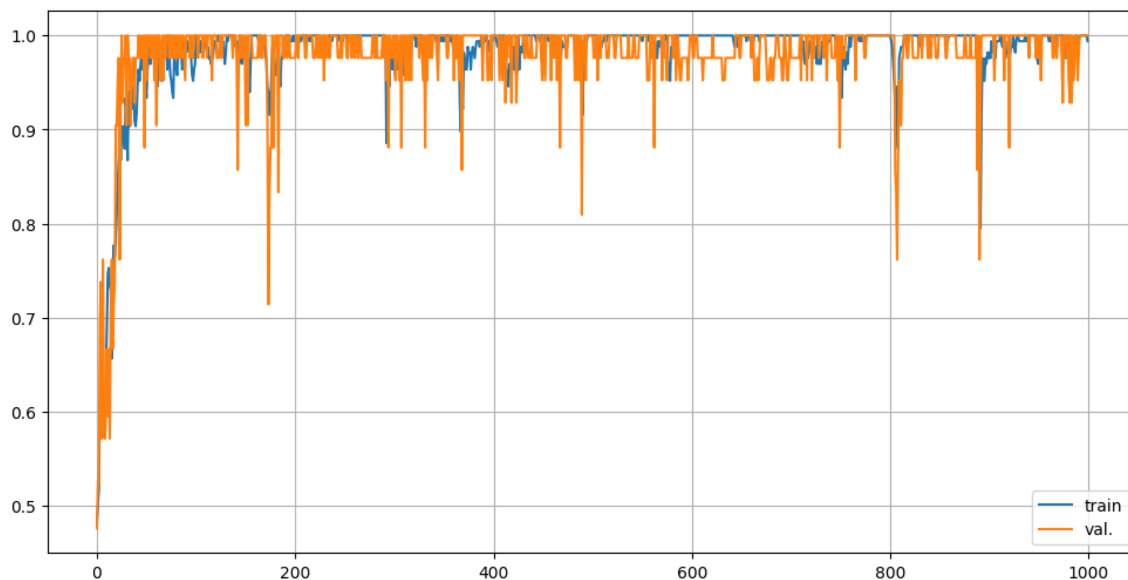
Fonte: Elaboração própria.

```
2 plt.figure(figsize=(12,6))
3 plt.semilogy(hist.history["loss"],label="train")
4 plt.semilogy(hist.history["val_loss"],label="val.")
5 plt.grid()
6 plt.legend()
7 
8 #para o grafico da acuracia
9 plt.figure(figsize=(12,6))
10 plt.plot(hist.history["accuracy"],label="train")
11 plt.plot(hist.history["val_accuracy"],label="val.")
12 plt.grid()
13 plt.legend()
```

Por fim, na nona e última seção do código, testamos o modelo — agora treinado — no conjunto de teste. Podemos visualizar os valores da função custo e acurácia, bem como a previsão das três primeiras amostras do conjunto de teste e compará-las com suas classificações reais.

```
1 # Calculando a funcao perda (custo) e a acuracia para todo o conjunto de
     teste
2 score = model.evaluate(Xte, yte, verbose=0)
3 print("Test loss:", score[0])
```

Figura 15: Imagem gerada pelo código para as curvas da acurácia durante o treinamento nos conjuntos de treino (em azul) e validação (em laranja)



Fonte: Elaboração própria.

```
4 print("Test accuracy:", score[1])
5
6 # Selecionando as tres primeiras amostras do conjunto de teste e
    calculando a previsao do modelo para elas
7 xnew = Xte[:3]
8 y_proba = model.predict(xnew)
9 y_proba
10
11 # Classificacao final do modelo (com base na maior probabilidade
    calculada entre as duas classes para cada amostra)
12 y_pred = y_proba.argmax(axis = -1)
13 y_pred
14
15 # Selecionando a classificacao real das tres primeiras amostras do
    conjunto de teste
16 y_r = yte[:3]
17
18 # Classificacao real das amostras selecionadas
19 y_real = y_r.argmax(axis = -1)
20 y_real
```

Aqui, as saídas foram $y_pred = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$ e $y_real = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$, que são os vetores que contém, respectivamente, as classificações previstas pelo modelo e retiradas diretamente do banco de dados original para as três primeiras amostras do conjunto de teste. Podemos ver que, no caso destas amostras, o modelo previu corretamente suas classes.

Mais importante, os valores da função custo e acurácia gerados em relação a todo o conjunto de teste foram $loss_teste = 0.0981$ e $acc_teste = 0.98$.

4 Conclusão

Ao analisar as Figuras 14 e 15, vemos que apesar de crescer bem rapidamente na acurácia de ambos treino e teste (em por volta de 50 épocas), o modelo consegue ao longo de mais épocas escapar dos pontos de mínimo iniciais, atingindo pontos em que a função custo é ainda menor — ou seja, mínimos ainda mais eficientes.

O dado mais importante para comprovar a viabilidade deste modelo é a acurácia do conjunto de teste, que nos diz que, de todos dados completamente novos que o modelo se propôs a classificar, 98% deles foram corretamente classificados. No nosso banco de dados, isso quer dizer que temos de 50 a 51 amostras (de um total de 52 testadas) que foram corretamente classificadas.

Referências

Francois Chollet et al. Keras, 2015. URL `https://github.com/fchollet/keras`.

Aurélien Géron. *Hands-on machine learning with scikit-learn keras and tensorflow*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA, United States of America, 3rd edition, 2022.

Ruggero Donida Labati, Vincenzo Piuri, and Fabio Scotti. All-idb: The acute lymphoblastic leukemia image database for image processing. In *2011 18th IEEE International Conference on Image Processing*, pages 2045–2048, 2011. doi: 10.1109/ICIP.2011.6115881.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.