

# APLICAÇÕES COM INTELIGÊNCIA ARTIFICIAL

Trilha de Visão Computacional

Aurélio Freire de Aquino - Engenheiro III

**Autor da apostila**

Aurélio Freire de Aquino - Engenheiro III

**Instrutor do curso**

Luana Dias Pena Forte – Assistente de Suporte Pedagógico

**Revisão da apostila**

**Fit Instituto de Tecnologia**

Manaus, Novembro de 2023

## Autor



**Aurélio Aquino** é formado em Engenharia da Computação pela UFAM – Universidade Federal do Amazonas e possui pós-graduação *lato sensu* em *Lean Manufacturing* e MBA em Gestão de Projetos Ágeis pelo IDAAM. Também foi aluno na California State University de Sacramento – Califórnia por um ano. Trabalha a mais de 5 anos na área de automação industrial, atuando nas áreas de robótica, visão computacional e desenvolvimento de *software*. Atualmente é instrutor de visão computacional no FIT – Instituto de Tecnologia da Amazônia.

## APRESENTAÇÃO

A presente apostila é um instrumento teórico que complementa o curso de capacitação de **Aplicações com Inteligência Artificial**, promovido pelo Fit - Instituto de Tecnologia da Amazônia, para colaboradores do Fit Instituto de Tecnologia, Flextronics e comunidade. Nela, teremos uma visão mais aprofundada dos conceitos, equipamentos e soluções usadas na indústria utilizando visão computacional.

Serão 30 horas de treinamento no qual apresentaremos desde os conceitos de aprendizado de máquina até o desenvolvimento de soluções completas de visão computacional, utilizando bibliotecas como OpenCV, Tensorflow e YOLO como base para o estudo e desenvolvimento. É extremamente recomendável ao aluno que, ao final da leitura de cada seção, realize os exercícios propostos, e acesse os materiais indicados nas referências bibliográficas, para aprofundar a leitura desse material e complementar o que foi lido aqui.

A apostila está dividida em quatro seções, iniciando com a Seção 1 com fundamentos de aprendizado de máquina com a biblioteca Scikit Learn. A Seção 2 abordará as redes neurais convolucionais com Tensorflow. A Seção 3 apresenta o algoritmo YOLO, um dos algoritmos mais avançados no campo de visão computacional. Para finalizar, a Seção 4 propõe um projeto final utilizando todo o conhecimento da trilha de Visão Computacional.

Desejo a você, prezado aluno, que tenha um excelente curso!!

***Boa Leitura !!***

## Sumário

1	Introdução ao Aprendizado de Máquina	5
1.1	Aprendizado Supervisionado e Não Supervisionado	5
1.2	Classificador K-NN	7
1.3	Arquivos	13
1.4	Support Vector Machines	14
1.5	Introdução a Redes Neurais	19
1.6	Exercício	23
1.7	Redes Neurais com Scikit Learn	23
1.8	Exercício	25
2	Redes Neurais Convolucionais	26
2.1	Vantagens das CNNs em relação às MLPs	26
2.2	Arquitetura das CNNs	27
2.3	Tensorflow e Keras	32
2.4	Treinando e Salvando CNNs	37
2.5	Abrindo e Avaliando a CNN	39
2.6	Exercício	41
3	Deteção de Objetos com YOLO	42
3.1	Modelos YOLOv8	44
3.2	Inferências com modelos	45
3.3	Validação no YOLOv8	48
3.4	Treinando com outras imagens	50
4	Projeto Final	54
	Conclusão	55
	Referências	56
	<b>CONTROLE DE REVISÃO DO DOCUMENTO / DOCUMENT REVISION</b>	
	<b>CONTROL</b>	<b>57</b>

## 1 Introdução ao Aprendizado de Máquina

Nesta seção, vamos explorar aspectos mais práticos e avançados em detecção de objetos e classificação de imagens.

### 1.1 *Aprendizado Supervisionado e Não Supervisionado*

Nos sistemas baseados em Visão Computacional, as técnicas de reconhecimento de padrões são essenciais, pois possibilitam a classificação automática de um objeto de interesse. Em outras palavras, essas técnicas permitem que computadores enxerguem o mundo à nossa volta, reconhecendo placas, peças, caracteres, faces humanas e outros objetos.

A participação humana é fundamental não apenas para definir o algoritmo de classificação, mas também, para definir as classes e as características consideradas no processo. Para que os classificadores consigam designar objetos em classes, sem que tenham sido explicitamente programados, eles precisam aprender alguns critérios para executar essa tarefa. Existem dois métodos de aprendizagem de máquina bastante usados para ensiná-los: a aprendizagem supervisionada e a não supervisionada.

No aprendizado supervisionado, o classificador é treinado para reconhecer padrões a partir de objetos já conhecidos. Um conjunto de características de objetos e suas respectivas classes são apresentados ao classificador. A partir desses dados, ele é treinado para identificar automaticamente padrões nessas informações, tornando-se capaz de classificar novos objetos.

Justamente por necessitar de um agente externo, responsável por apresentar ao classificador dados para o treinamento, esse método é conhecido como supervisionado. Geralmente, os dados usados na etapa de treinamento são definidos previamente.

A figura a seguir ilustra o processo de aprendizagem supervisionada. Observe que um conjunto de objetos está representado em um espaço de características bidimensional. Todos os objetos coloridos possuem classes

definidas e indicadas pela legenda da imagem; eles representam a base de conhecimento do classificador.

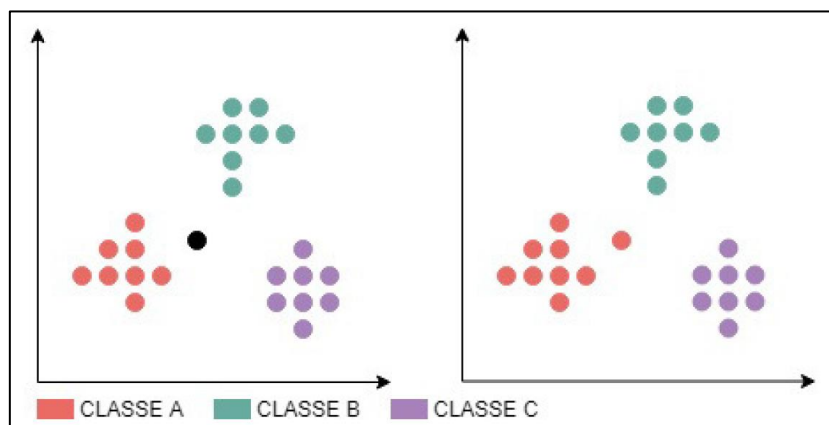


Figura 1. Aprendizagem Supervisionada. Fonte: Livro “Introdução à Visão Computacional”.

No entanto, a imagem à esquerda exibe um elemento de classe desconhecida, representado na cor preta. Os classificadores, baseados nesse tipo de aprendizagem, são capazes de classificar esse elemento a partir das informações sobre as características dos elementos conhecidos (já classificados). A imagem à direita apresenta esse mesmo objeto, após o processo de classificação, definido em uma das classes.

Assim como na aprendizagem supervisionada, a aprendizagem não-supervisionada também necessita de uma lista de características de diversos objetos para o treinamento. A diferença é que as classificações desses objetos não precisam ser apresentadas, descartando a necessidade de um supervisor.

No nosso curso, iremos focar em técnicas de aprendizado supervisionado.

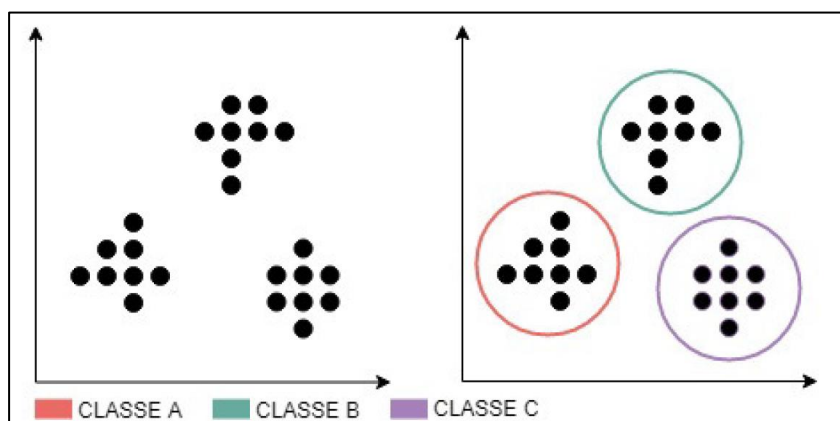


Figura 2. Aprendizagem não-supervisionada. Fonte: Livro “Introdução à Visão Computacional”.

## 1.2 Classificador K-NN

O algoritmo K-Nearest Neighbors (K-NN), também conhecido como K Vizinhos Mais Próximos, é um dos algoritmos de classificação mais simples de ser compreendido. Por ser de fácil implementação e capaz de solucionar rapidamente problemas que envolvem reconhecimento de padrões, ele é amplamente utilizado.

Considere a classificação de objetos por sua massa e área:

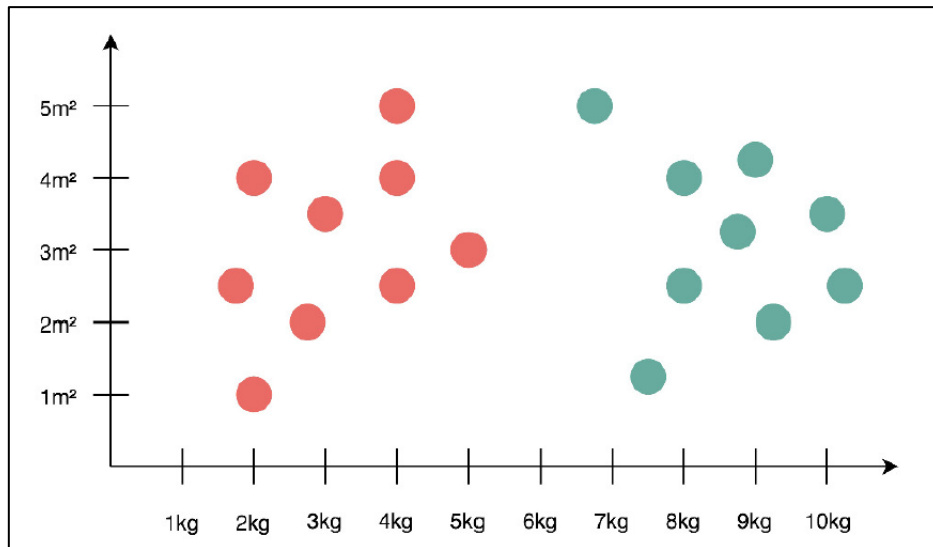


Figura 3. Classificando objetos por massa e área. Fonte: Livro “Introdução à Visão Computacional”.

No esquema ilustrado, observe que as características de massa e área, de cada objeto apresentado, estão definidas em um espaço euclidiano bidimensional. Os objetos definidos à esquerda no espaço, na cor vermelha, indicam uma classe; os definidos à direita, na cor verde, indicam uma outra classe.

A figura a seguir ilustra a etapa de classificação. Perceba que um novo objeto, representado na cor cinza e cuja classificação é desconhecida, foi inserido ao conjunto.



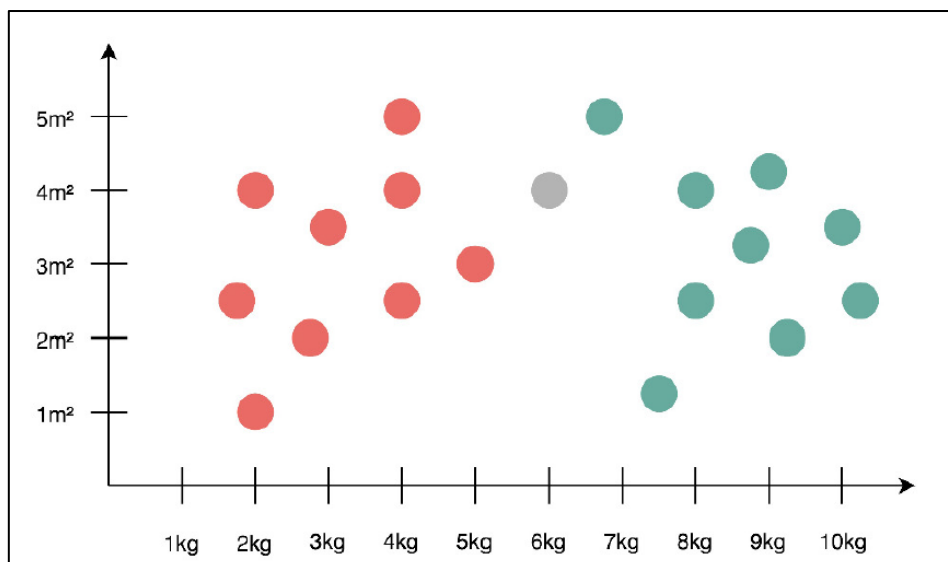


Figura 4. Novo elemento em cinza. Fonte: Livro “Introdução à Visão Computacional”.

Observe que esse novo objeto possui valores de massa e área que podem nos confundir ao tentar classificá-lo como pertencente a uma das duas classes. Entretanto, esta questão pode ser rapidamente solucionada por meio do classificador K-NN.

Para classificá-lo, o K-NN calcula a distância euclidiana entre o objeto e os seus K vizinhos mais próximos. Considerando K igual a 5, a distância entre os 5 vizinhos mais próximos será verificada, como mostra a figura a seguir.

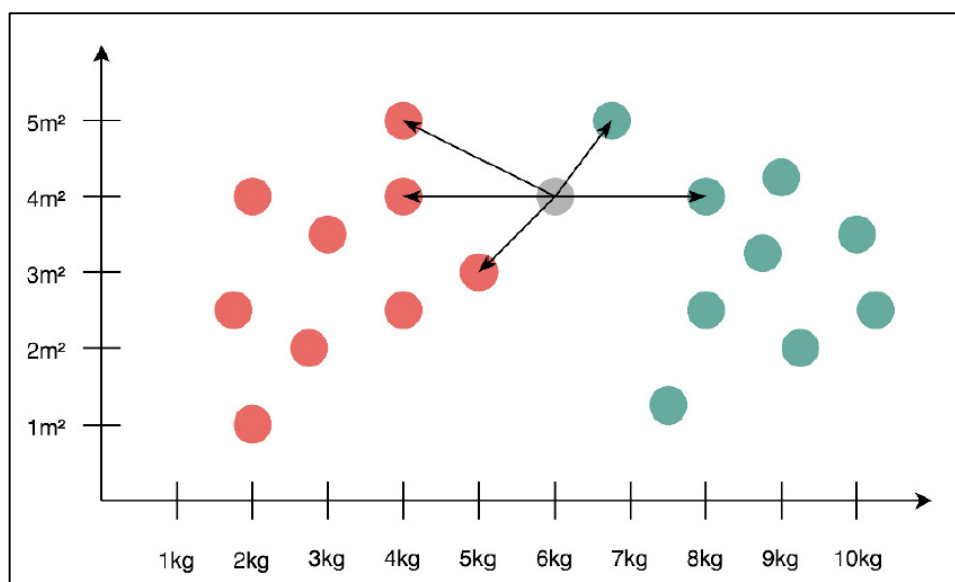


Figura 5. Classificação do novo elemento. Fonte: Livro “Introdução à Visão Computacional”.

Ao comparar 5 vizinhos, observamos 3 da classe vermelha e 2 da verde. A partir dessa informação, o classificador determina que o novo elemento pertence à classe vermelha.

Neste exemplo, foi utilizado K igual a 5, entretanto, K pode ser qualquer valor inteiro positivo, mas em geral, sempre se utiliza valores ímpares com o intuito de não gerar empates entre classes. Além disso, o exemplo mostra uma classificação com duas características: massa e área. Porém, o número de características pode ser bem maior.

Para utilizar K-NN em processamento de imagens, vamos utilizar a biblioteca chamada “SciKit Learn”, uma ampla biblioteca com diversas ferramentas para aprendizado de máquina. A instalação é feita da seguinte forma:

```
pip install -U scikit-learn
```

Vamos classificar objetos baseados nas classes A e B exemplificados a seguir.

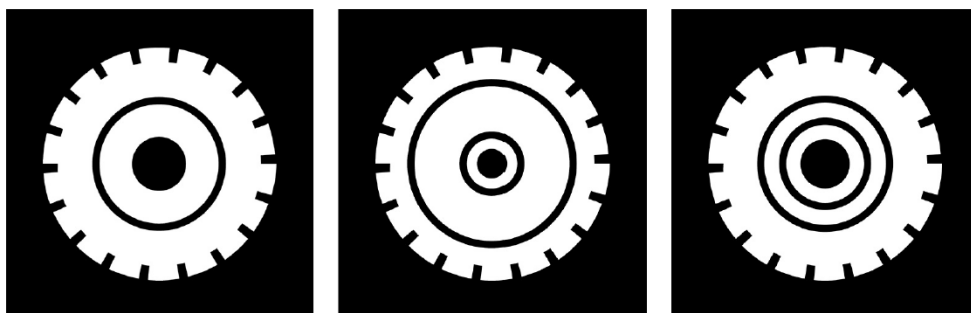


Figura 6. Objetos da classe A. Fonte: Livro “Introdução à Visão Computacional”.

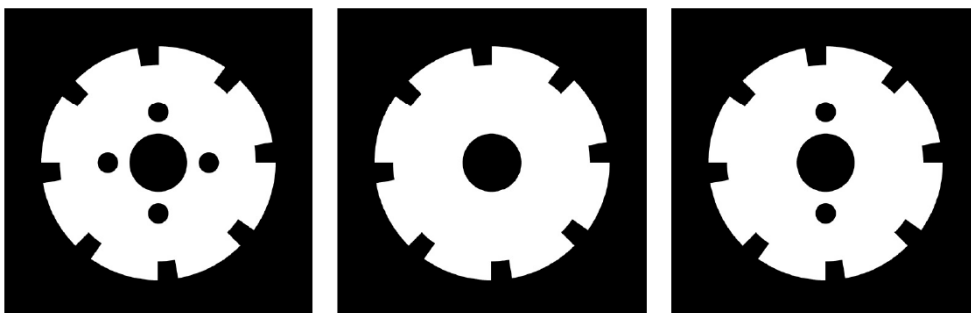


Figura 7. Objetos da classe B. Fonte: Livro “Introdução à Visão Computacional”.

Inúmeras características desses objetos poderiam ser usadas para classificá-los. Uma delas é a quantidade de dentes que compõe as engrenagens de cada classe. Observe que as engrenagens da classe A possuem 17 dentes, diferente das da classe B, que possuem 7 dentes.

Porém, encontrar a quantidade de dentes das engrenagens pode ser uma tarefa de alta complexidade. Podemos então utilizar os momentos invariantes de Hu de cada objeto.

Objeto	M1	M2	M3	M4	M5	M6	M7
A1.jpeg	3.10	11.98	14.68	16.86	-32.64	22.85	33.25
A2.jpeg	3.11	12.36	14.63	17.06	32.99	23.34	33.16
A3.jpeg	3.07	12.26	14.62	15.32	31.4	-22.69	-30.30
A4.jpeg	3.10	11.98	14.68	16.86	-32.64	22.85	33.25
A5.jpeg	3.11	12.36	14.63	17.06	32.99	23.34	33.16
A6.jpeg	3.07	12.26	14.62	15.32	31.4	-22.69	-30.30
B1.jpeg	3.12	11.72	14.76	15.33	32.08	21.40	30.37
B2.jpeg	3.14	11.78	14.83	15.59	-34.29	21.71	30.81
B3.jpeg	3.13	10.43	14.77	15.44	31.55	20.75	30.54
B4.jpeg	3.12	11.72	14.76	15.33	32.08	21.40	30.37
B5.jpeg	3.14	11.78	14.83	15.59	-34.29	21.71	30.81
B6.jpeg	3.13	10.43	14.77	15.44	31.55	20.75	30.54

Figura 8. Momentos Invariantes de Hu para diferentes imagens. Fonte: Livro “Introdução à Visão Computacional”.

Vamos importar a classe *KNeighborsClassifier* da biblioteca *SciKit-Learn* e criar um vetor de características de cada imagem:

```
from sklearn.neighbors import KNeighborsClassifier
```

```
caracteristicas = [
    [3.10, 11.98, 14.68, 16.86, -32.64, 22.85, 33.25],
    [3.11, 12.36, 14.63, 17.06, 32.99, 23.34, 33.16],
    [3.07, 12.26, 14.62, 15.32, 31.40, -22.69, -30.30],
    [3.10, 11.98, 14.68, 16.86, -32.64, 22.85, 33.25],
    [3.11, 12.36, 14.63, 17.06, 32.99, 23.34, 33.16],
    [3.07, 12.26, 14.62, 15.32, 31.40, -22.69, -30.30],
    [3.12, 11.72, 14.76, 15.33, 32.08, 21.40, 30.37],
```

```
[3.14, 11.78, 14.83, 15.59, -34.29, 21.71, 30.81],  
[3.13, 10.43, 14.77, 15.44, 31.55, 20.75, 30.54],  
[3.12, 11.72, 14.76, 15.33, 32.08, 21.40, 30.37],  
[3.14, 11.78, 14.83, 15.59, -34.29, 21.71, 30.81],  
[3.13, 10.43, 14.77, 15.44, 31.55, 20.75, 30.54]  
]
```

Para determinar as classes de cada característica, cria-se também um vetor de classificações sendo 0 o valor que representa a classe A, e 1 o valor que representa a classe B.

```
classificacoes = [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
```

O próximo passo é criar o classificador com a classe importada anteriormente. Neste exemplo, o valor de K será 3, pois o conjunto de dados é pequeno. Não é recomendado utilizar K com valores grandes, quando há poucos dados, pois a comparação com os vizinhos mais próximos poderá considerar quase que todos os dados presentes.

```
knn = KNeighborsClassifier(3)
```

O treinamento do classificador K-NN é realizado pelo método *fit*, que requer dois parâmetros: a lista de características e a lista de classificações.

```
knn.fit(caracteristicas, classificacoes)
```

Para garantir o bom funcionamento do classificador é necessário fazer uma validação para garantir uma boa separação das classificações. Utiliza-se a função *score*, recebendo dois parâmetros iguais à função *fit*: uma lista de características e sua lista de classificações.

```
pontos = knn.score(caracteristicas, classificacoes)
```

O resultado é um valor entre 0 e 1, sendo que o valor 0 representa 0% de acertos na classificação e o valor 1 representa 100% de acertos.

E por fim, a biblioteca fornece a função *predict* para classificar um novo elemento. A função possui um parâmetro: o objeto a ser classificado. No código abaixo, um novo objeto desconhecido é criado e classificado.

```
objetoDesconhecido = [  
3.17, 11.84, 14.91, 16.22, -33.21, 22.38, 31.78  
]  
classe = knn.predict(objetoDesconhecido)
```

A variável “classe” será 0 se o classificador considerar o objeto pertencente à classe A, ou será 1 se o classificador considerar o objeto pertencente à classe B.

A função pode aceitar uma lista de objetos para serem classificados. No exemplo a seguir, dois objetos são classificados.

```
objetos = [  
[3.17, 11.84, 14.91, 16.22, -33.21, 22.38, 31.78],  
[3.17, 11.82, 14.91, 16.21, -33.38, 22.33, 31.77]  
]  
print(knn.predict(objetos))
```

A saída é:

```
[1 1]
```

### 1.3 Arquivos

A linguagem *Python* permite manipular arquivos, consumindo o conteúdo e criando novos arquivos. A função central na manipulação de arquivos em *Python* é a função *open*. Essa função recebe dois parâmetros: nome do arquivo e modo.

```
arquivo = open('mensagem.txt', 'r')
```

Os modos disponíveis são:

“r” (read) - Valor padrão. Abre o arquivo em modo leitura. Gera um erro se o arquivo não existir.

“a” (append) - Abre o arquivo para anexar mais conteúdo. Cria o arquivo se não existir.

“w” (write) - Abre um arquivo em modo escrita. Cria o arquivo se não existir.

“x” - Cria um arquivo. Gera um erro se o arquivo já existe.

Em combinação com os 4 modos listados, podemos adicionar o modo de tratamento do arquivo:

“t” - Valor padrão. Modo de texto.

“b” - Modo binário.

Ou seja, “rb” determina abrir um arquivo em modo leitura e em formato binário por exemplo.

Para ler o arquivo, utilize a função *read*:

```
texto = arquivo.read()
```

Na função *read* podemos também passar quantos caracteres desejamos ler:

```
algumasLetras = arquivo.read(5)
```

Ou podemos ler uma linha do texto com a função *readline*:

```
linhaDeTexto = arquivo.readline()
```

## 1.4 Support Vector Machines

SVMs (Support Vector Machines, ou Máquinas de Vetores de Suporte em português) são um tipo de algoritmo de aprendizado de máquina supervisionado, usado para classificação e regressão. Eles funcionam criando uma linha ou hiperplano de separação entre diferentes classes de dados em um espaço dimensional, de modo que a distância entre a linha e os pontos mais próximos de cada classe seja a maior possível. Isso é chamado de "margem máxima".

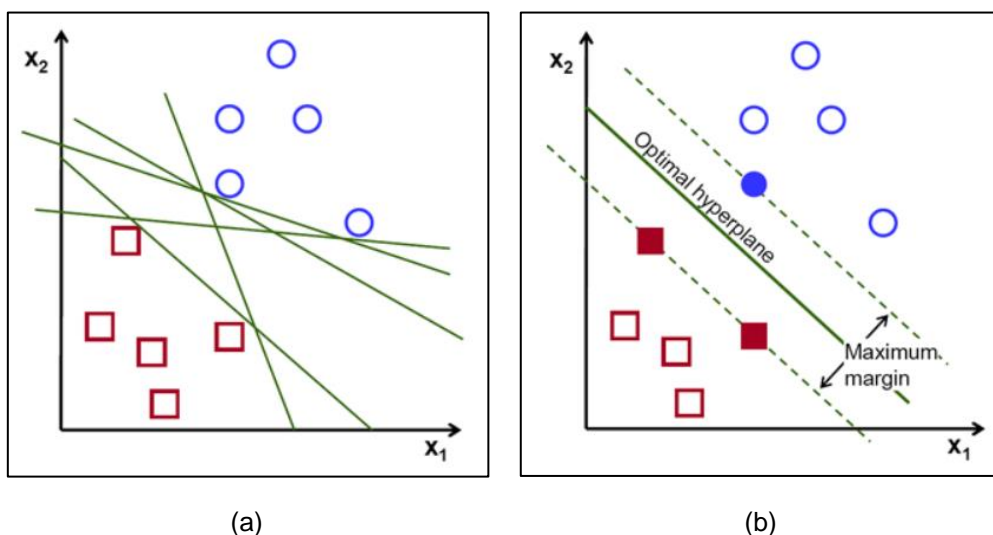


Figura 9. a) Possíveis retas separando duas classes; b) Melhor hiperplano encontrado e margem máxima. Fonte: OpenCV Docs.

SVMs são considerados um dos algoritmos mais poderosos e eficientes de aprendizado de máquina, especialmente para problemas de classificação binária (onde há apenas duas classes). Eles são amplamente utilizados em aplicações em que é necessário realizar classificações precisas com base em dados complexos, como reconhecimento de padrões em imagens ou texto.

No exemplo da figura 9, os dados possuem dois valores de informações,  $x_1$  e  $x_2$ . Logo, a variável "dados" teria o seguinte formato:

```
[  
  [v10, v20],  
  [v11, v21],
```

```
...,  
[v1n, v2n]  
]
```

No caso das imagens, devemos fazer um processamento que retorne 2 valores para representar uma imagem, ou podemos utilizar  $N$  valores para cada imagem, sendo  $N > 2$ . Ao invés de encontrar uma reta, a *SVM* encontrará um hiperplano para a quantidade de valores dos dados.

Utilizando a função *flatten* em uma imagem de escala de cinza, transforma-se a imagem do formato  $N \times M$  para  $1 \times N \times M$ , tendo uma lista uniforme com os valores dos pixels:

```
pixels = imgCinza.flatten()
```

Com isso, podemos criar uma lista de pixels para um banco de imagens, para gerar a variável “dados”. No código abaixo, são abertas inúmeras imagens de inúmeros diretórios, utilizando a função *listdir* da biblioteca *os* (*Operating System*):

```
import os  
nomesDasImagens = os.listdir('diretorioDelImagens/')
```

```
7  diretorios = ['classes/cachorro/', 'classes/gato/']  
8  
9  dados = None  
10 classes = np.array([])  
11 num_diretorios = len(diretorios)  
12 for i in range(num_diretorios):  
13     imagens = os.listdir(diretorios[i])  
14     for imagem in imagens:  
15         gray = cv.imread(diretorios[i]+imagem, cv.IMREAD_GRAYSCALE)  
16         if gray is None:  
17             continue  
18         gray = cv.resize(gray, (400, 400))  
19         if dados is not None:  
20             dados = np.append(dados, [gray.flatten()], axis=0)  
21         else:  
22             dados = np.array([gray.flatten()])  
23     classes = np.append(classes, i)
```

Figura 10. Código de exemplo. Fonte: Autoria própria



1. Na linha 7, cria-se uma lista de diretórios;
2. Linhas 9 e 10, são inicializadas as listas de dados e suas classes, onde serão colocados os dados processados das imagens e a classe de cada imagem;
3. Na linha 11, obtém-se o número de diretórios listados, no caso do exemplo, 2 diretórios;
4. A linha 12 é o laço de repetição para acessar cada diretório, sendo “i” o índice;
5. A linha 13 utiliza a biblioteca “os” (Operating System) para listar os arquivos de um diretório com a função *listdir*;
6. A linha 14 inicia o laço de repetição para processar cada imagem;
7. A linha 15 lê a imagem em escala de cinza, e na linha 16 e 17 é verificado se houve algum erro com a imagem, caso a imagem esteja vazia. O comando *continue* encerra o processamento e passa para a próxima iteração do laço da linha 14;
8. A linha 18 coloca a imagem sempre em tamanho fixo 400x400, para compararmos sempre a mesma quantidade de pixels, além de suprir a necessidade de dados com o mesmo tamanho;
9. Das linhas 19 até 22, adiciona-se a imagem processada na lista de dados. Caso a lista não esteja vazia (19 e 20), utiliza-se a função *append* da biblioteca *numpy* para adicionar a imagem à lista de dados. Caso a lista esteja vazia (21 e 22), um *array* é criado com a primeira imagem;
10. Por fim, a classe da imagem é adicionada na lista de classes.

*Nota: a função **append** tem como parâmetros: a lista para adicionar o elemento, o elemento a ser adicionado e um parâmetro opcional axis para indicar em qual dimensão do array o elemento será adicionado. Se axis fosse ignorado, a lista teria o formato 1xM, mas a ideia é que a lista seja (400\*400)xN, ou seja, 160000xN.*

Para utilizar SMVs em processamento de imagens, também vamos utilizar a biblioteca *SciKit Learn*. Vamos começar com a criação de um objeto com classe *SVC* (*Support Vector Classification*) contida no módulo *svm* da biblioteca.

```
from sklearn import svm
model = svm.SVC()
```

Esse objeto será o modelo a ser treinado com os dados. Os dados nada mais são do que um *array* de valores com suas respectivas classificações em outro *array*. Para “treinar” o modelo e obter a melhor curva de separação de classes, usamos o método *fit* passando os dados de entrada e suas classes.

```
model.fit(dados, classes)
```

O tempo de treino para modelos do tipo *SVC* pode chegar a ser quadrático, ou seja, quanto mais dados a serem treinados, mais tempo será preciso para treinar a *SVM*. Se for o caso de muitos dados e suas classes são linearmente separáveis, considere utilizar outra classe chamada *LinearSVC*, que pode ser treinada da mesma forma e gera resultados tão bons ou até melhores que a classe *SVC*.

```
model = svm.LinearSVC()
model.fit(dados, classes)
```

Com o modelo treinado, podemos utilizá-lo para predizer se uma ou mais imagens pertence à classe A ou B com a função *predict*, passando como parâmetro a lista de imagens (lembre-se que deve estar no mesmo formato processado anteriormente):

```
previsoes = model.predict(dados)
```

A saída será uma lista de previsões para cada imagem da lista de dados:

```
[0, 0, 0, 1, ..., 1]
```

Para salvar seu modelo e usá-lo em outro momento, utilize a biblioteca *pickle* com as funções *dump* para salvar o modelo, e *load* para carregá-lo. No código a seguir o modelo é salvo no arquivo “modelo.sav”. Observe que *dump* e

*load* recebem um arquivo aberto com a função *open* para criar e ler o arquivo em modo binário:

```
import pickle
...
modelo = ...
...
pickle.dump(modelo, open('modelo.sav', 'wb'))
modelo = pickle.load(open('modelo.sav', 'rb'))
```

Um dos principais benefícios das SVMs é sua capacidade de lidar bem com conjuntos de dados "não-linearmente separáveis". Isso significa que eles podem encontrar uma linha de separação, que não é uma simples reta, mas sim uma curva ou outra forma complexa. Isso é possível graças ao uso de funções de kernel, que transformam os dados em um espaço dimensional mais alto, onde eles podem ser linearmente separáveis. Nesse caso, uma SVM do tipo linear (*LinearSVC* utiliza uma *SMV* com kernel linear) não será capaz de separar as classes como mostra a imagem a seguir.

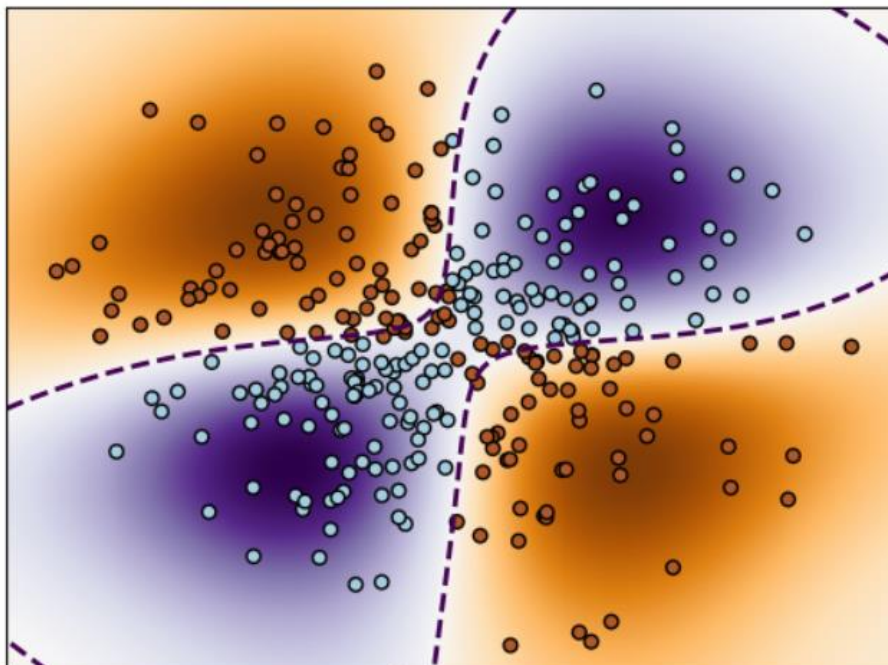


Figura 11. Classes não linearmente separáveis. Fonte: Scikit Learn Docs.

Ao criar um modelo com a classe *SVC*, utilize o parâmetro opcional “*kernel*” para selecionar o tipo de kernel para a classificação. Alguns tipos disponíveis são: *linear*, *poly*, *rbf* (valor padrão) e *sigmoid*. No exemplo abaixo, utiliza-se o kernel linear, o que não seria adequado para classificar os dados demonstrados na figura 11.

```
modelo = svm.SVC(kernel="linear")
```

## **1.5 Introdução a Redes Neurais**

Em seu núcleo, as redes neurais são apenas mais uma ferramenta no conjunto de algoritmos de aprendizado de máquina. Elas consistem em um conjunto de "neurônios", que nada mais são do que somas ponderadas por “pesos” e passados por uma função de ativação para produzir um valor final. Esse processo se repete ao longo das "camadas", que a rede neural tem para produzir uma saída. Na figura abaixo é possível observar a semelhança entre o neurônio biológico e o neurônio artificial.

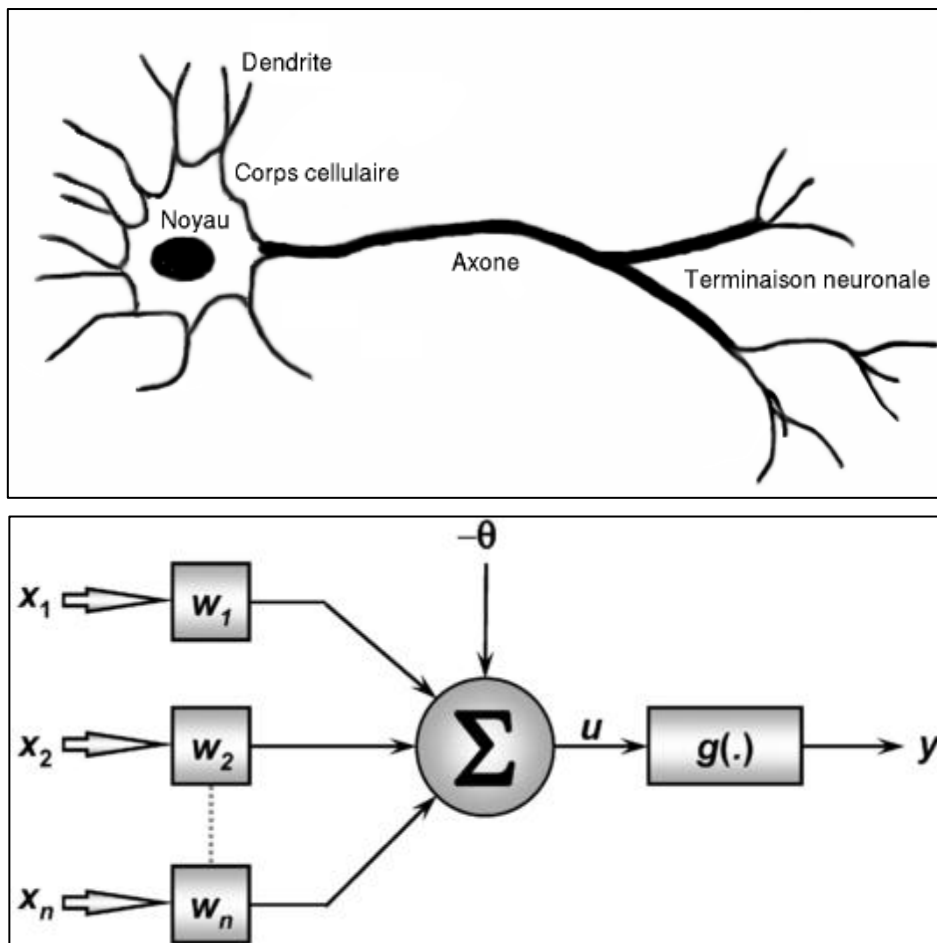


Figura 12. a) Neurônio biológico; b) Neurônio artificial. Fonte: Wikipedia.

Um neurônio artificial como o da figura anterior pode ser representado com a seguinte fórmula matemática:

$$u = x_1w_1 + x_2w_2 + x_3w_3 + \dots + x_nw_n + b$$

$$y = G(u)$$

Sendo “x” os valores de entrada, “w” os pesos, “b” um valor constante (bias) e “G” a função de ativação.

A função de ativação tem o propósito de determinar o quanto o neurônio está sendo “ativado”, através do valor final obtido na soma ponderada, realizada pelo neurônio. Algumas funções de ativação comuns são: *Sigmoid*, *ReLU* (*Rectified Linear Unit*) e Tangente Hiperbólica.

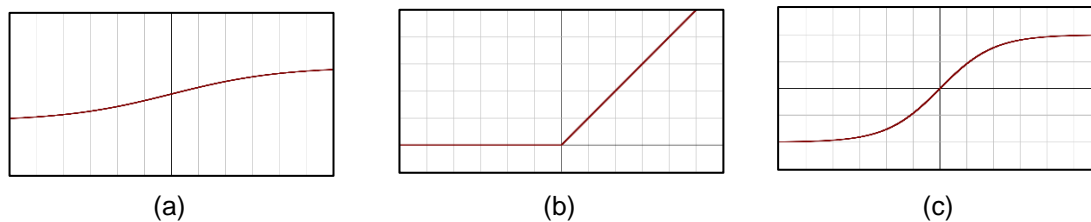


Figura 13. a) Sigmoid; b) ReLU; c) Tangente Hiperbólica. Fonte: Wikipedia.

A função *Sigmoid* converte valores positivos para valores entre 0.5 e 1, e converte valores negativos para valores entre 0 e 0.5, sendo formalmente descrita pela fórmula abaixo.

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

A função *ReLU* (*Rectified Linear Unit*) é a função mais utilizada em redes neurais e, em geral, obtém performances melhores nas etapas de treinamento. Ela converte valores negativos em 0 e mantém os valores positivos, sendo formalmente descrita pela fórmula abaixo.

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

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

A função Tangente Hiperbólica converte os valores positivos em valores entre 0 e 1, enquanto valores negativos são convertidos em valores entre -1 e 0. Ela é formalmente descrita pela fórmula abaixo.

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

Uma rede neural será composta por vários neurônios em camadas, como na figura a seguir. Em verde, estão os neurônios da camada de **entrada**, em azul estão os neurônios da camada **oculta** e em amarelo o neurônio da camada de **saída**. As redes neurais podem ter diversas camadas ocultas, porém, apenas uma camada de entrada e uma camada de saída.

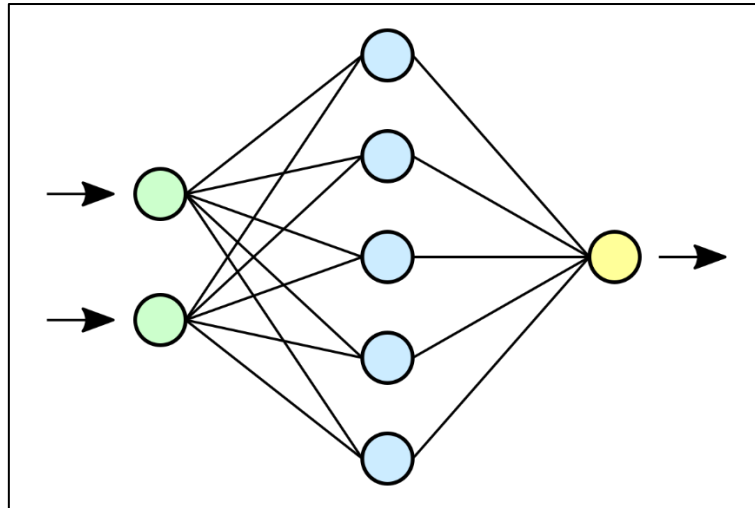


Figura 14. Rede Neural Artificial. Fonte: Wikipedia.

O Google disponibiliza uma ferramenta de testes de redes neurais com outra biblioteca chamada *Tensorflow*, disponível no site <https://playground.tensorflow.org/>. Nesse site, podemos simular diferentes arquiteturas de redes neurais para diferentes dados.

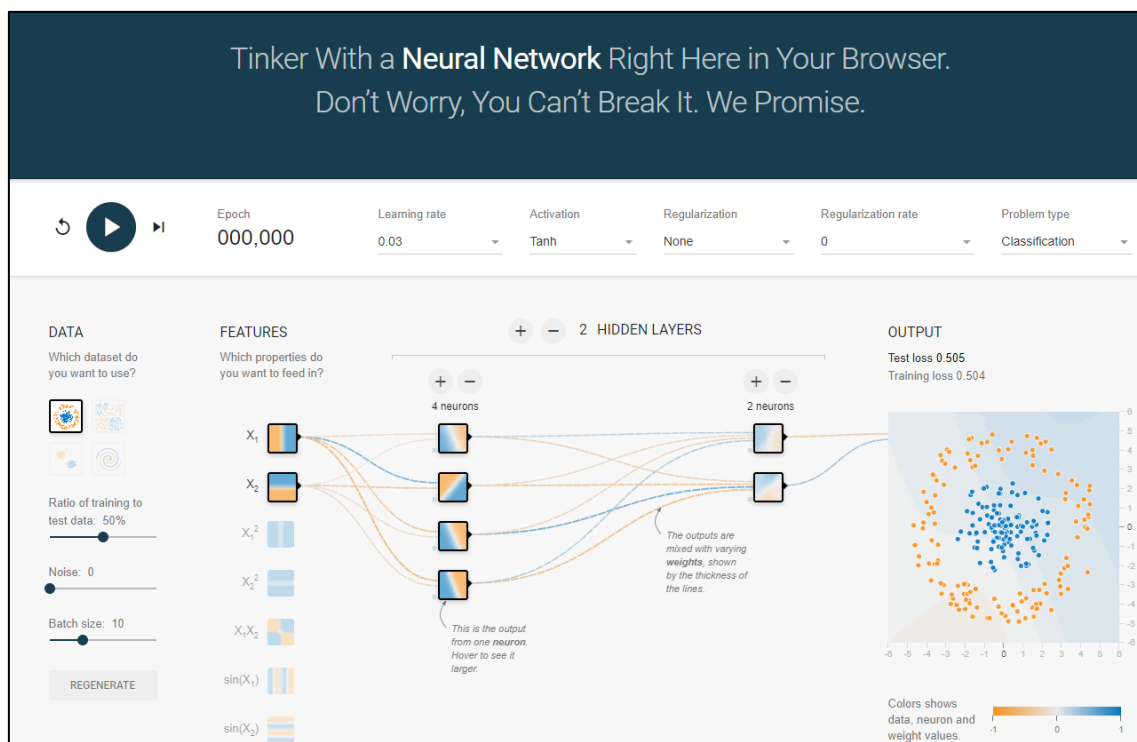


Figura 15. Exemplo de redes neurais com Tensorflow. Fonte: Site Tensorflow Playground.

## 1.6 Exercício

1. Utilize o Playground Tensorflow para classificar os dados em espiral. Atinja um *training loss* igual à 0 com a menor rede neural possível.

Após a execução da atividade prática, tire um print *do site*, e suba a imagem na plataforma do fiteducação (edX).

## 1.7 Redes Neurais com Scikit Learn

Vamos utilizar mais uma vez a biblioteca *Scikit Learn*. Vamos começar a montar uma rede neural para classificar números escritos à mão do banco de imagens chamado *MNIST*, um grande banco de imagens comumente utilizado para treinar sistemas de processamento de imagens.



Figura 16. Dígitos do banco de imagens MNIST. Fonte: Wikipedia.

Para obter os dados do banco de imagens *MNIST*, utilize a função *fetch\_openml* de *sklearn.datasets*. Os parâmetros são: o conjunto de dados, versão dos dados (opcional) e se o *download* deve ser separado em: dados (*X*) e classes (*y*).

```
from sklearn.datasets import fetch_openml
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
```



A normalização ajuda a treinar a rede neural com velocidade, obtendo uma convergência mais rápida. Para normalizar os dados, dividimos os valores dos dados por 255, pois os dados obtidos não estavam com valores entre 0 e 1, comum nos dados do mundo real. No caso das imagens, os valores dos pixels podem ser divididos por 255 para convertê-los para essa escala.

$$X = X / 255$$

Para validar a rede neural (ou qualquer outra técnica de aprendizagem de máquina), os testes devem ser feitos em dados nunca vistos no treinamento. Por isso, os dados serão separados em dois conjuntos: dados de treino e dados de testes:

```
X_treino, X_teste = X[:60000], X[60000:]  
y_treino, y_teste = y[:60000], y[60000:]
```

Finalmente, a rede neural pode ser instanciada com a classe *MLPClassifier* de *sklearn.neural\_network* e treinada com o método *fit*:

```
from sklearn.neural_network import MLPClassifier  
redeNeural = MLPClassifier()  
redeNeural.fit(X_treino, y_treino)
```

Após treinar a rede neural, utilize a função *score* para testar sua rede neural com os dados treino e principalmente com os dados de teste separados previamente:

```
print('Treino:', redeNeural.score(X_treino, y_treino))  
print('Teste:', redeNeural.score(X_teste, y_teste))
```

A saída será semelhante ao texto abaixo. No caso, 100% de acerto nos dados de treino e 97,9% de acertos nos dados de teste:

```
Treino: 1.0  
Teste: 0.979
```

## 1.8 Exercício

1. Faça um programa que classifique objetos quaisquer na webcam. Escolha duas técnicas e compare os resultados da classificação. Como saída, seu programa deve detectar e escrever na imagem o objeto detectado. Recomenda-se fazer scripts separados para cada etapa:
  - 1) capturar as imagens
  - 2) treinar o modelo de aprendizado de máquina
  - 3) utilizar o modelo de aprendizado de máquina para classificação.

Após a execução da atividade prática, faça upload dos *arquivos* “.py” em formato “.zip”, e suba na plataforma do fiteducação (edX).

## 2 Redes Neurais Convolucionais

Uma Rede Neural Convolucional (ConvNet / Convolutional Neural Network / CNN) é um algoritmo de Aprendizado Profundo que pode captar uma imagem de entrada, atribuir importância (pesos e vieses que podem ser aprendidos) a vários aspectos / objetos da imagem e ser capaz de diferenciar um do outro. O pré-processamento exigido em uma ConvNet é muito menor em comparação com outros algoritmos de classificação. Enquanto nos métodos primitivos os filtros são feitos à mão, com treinamento suficiente, as ConvNets têm a capacidade de aprender esses filtros / características.

### 2.1 Vantagens das CNNs em relação às MLPs

Um problema ao utilizar MLPs (Redes Neurais Multi Camadas) para processar dados de imagens é que eles não são invariáveis à transladação. Isso significa que a rede reage de maneira diferente se o conteúdo principal da imagem for deslocado. Como MLPs respondem de forma distinta a imagens deslocadas, os exemplos a seguir ilustram como tais imagens complicam o problema de classificação e produzem resultados não confiáveis.

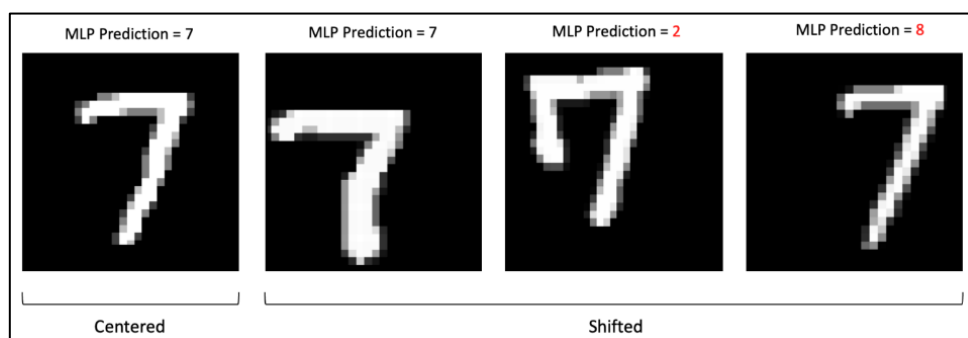


Figura 17. Inferências incorretas em objeto não centralizado. Fonte: LearnOpenCV.

Outra questão ao lidar com dados de imagem usando MLPs é que essas redes usam um único neurônio para cada pixel de entrada na imagem. Isso resulta em um número rapidamente incontrolável de pesos na rede, especialmente quando tratamos de imagens grandes com vários canais. Por exemplo, considere uma imagem colorida com dimensões de (224x224x3). A

camada de entrada de uma MLP teria 150.528 neurônios. Se adicionarmos apenas três camadas ocultas de tamanho moderado, com 128 neurônios cada, além da camada de entrada, o número de parâmetros treináveis na rede ultrapassaria os 300 bilhões! Isso não apenas tornaria o tempo de treinamento excessivamente longo para uma rede desse porte, mas também aumentaria significativamente o risco de superajuste aos dados de treinamento, devido ao grande número de parâmetros treináveis.

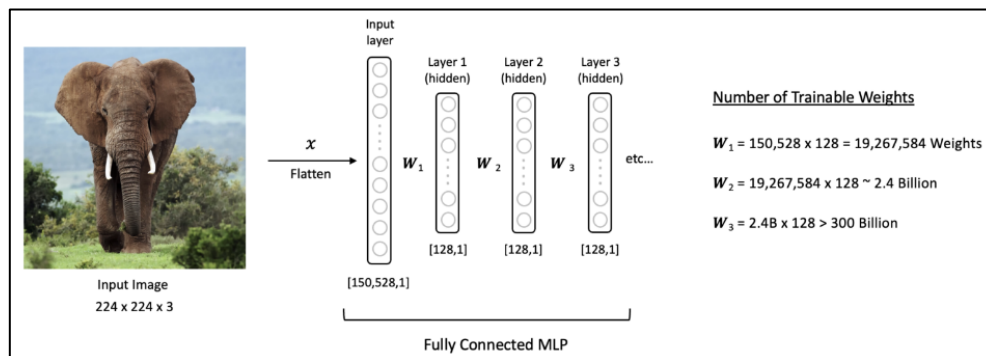


Figura 18. Tamanho de uma MLP para imagens coloridas. Fonte: LearnOpenCV.

## 2.2 Arquitetura das CNNs

A arquitetura de uma ConvNet é análoga àquela do padrão de conectividade de neurônios no cérebro humano e foi inspirada na organização do Visual Cortex. Os neurônios individuais respondem a estímulos apenas em uma região restrita do campo visual conhecida como Campo Receptivo. Uma coleção desses campos se sobrepõe para cobrir toda a área visual.

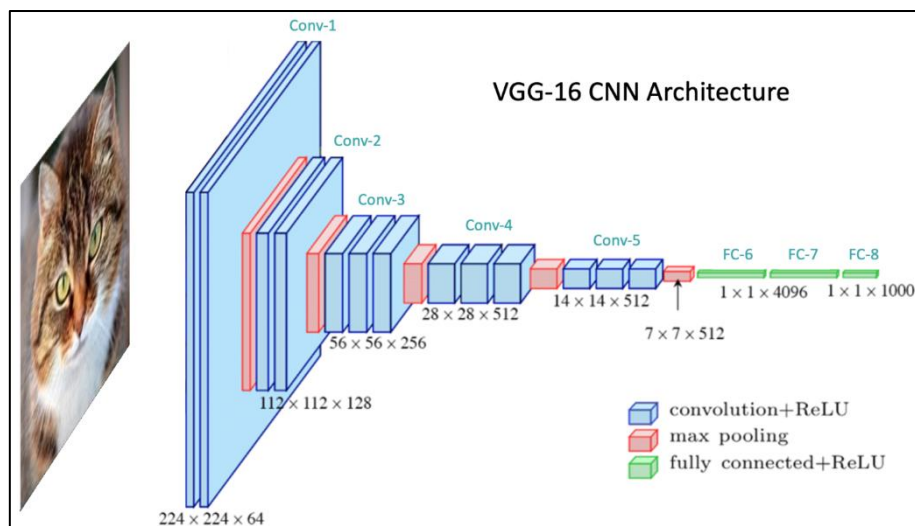


Figura 19. Exemplo de arquitetura de CNNs. Fonte: LearnOpenCV.

As CNNs são divididas em duas partes: uma parte projetada para extrair características e outra parte projetada para ser o classificador. A segunda parte é igual às redes neurais que vimos anteriormente, com todos os neurônios conectados para gerar uma saída final. Já a primeira parte, extratora de características, é onde as redes neurais convolucionais se diferenciam das redes neurais comuns.

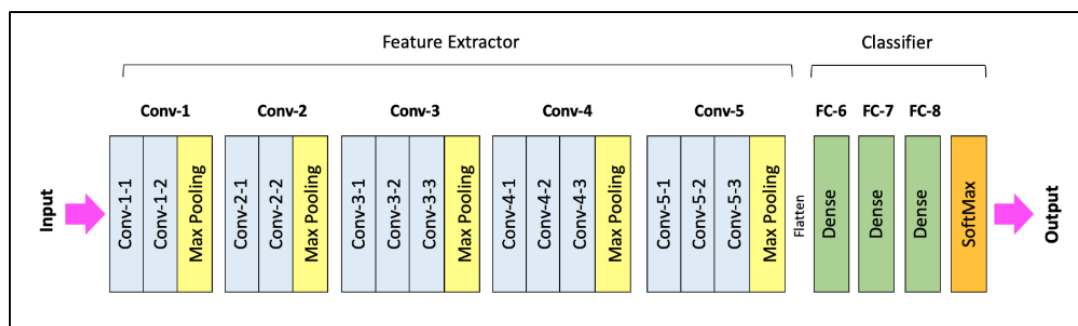


Figura 20. Partes de uma CNN. Fonte: LearnOpenCV.

As camadas convolucionais (convolutional layers) são uma parte fundamental das Redes Neurais Convolucionais (Convolutional Neural Networks - CNNs), que são uma classe especializada de redes neurais profundas projetadas especialmente para o processamento de dados de imagem e reconhecimento de padrões em imagens. As camadas convolucionais desempenham um papel crucial na extração de características das imagens, sendo formados processos que seguem alguns passos:

1. **Convolução:** A operação de convolução é o coração das camadas convolucionais. Ela envolve uma pequena janela chamada "filtro" ou "kernel" que desliza pela imagem de entrada, multiplicando seus valores pelos valores correspondentes na imagem e somando os resultados para produzir um único valor em uma nova matriz chamada "mapa de características" ou "feature map".
2. **Camadas de Pooling:** Após a convolução, as CNNs frequentemente incluem camadas de pooling (também chamadas de camadas de subamostragem), como a camada de Max Pooling. Essas camadas reduzem a dimensionalidade do mapa de características, mantendo as características mais importantes. Isso ajuda a tornar a rede mais eficiente e a reduzir o risco de overfitting.
3. **Camadas de Ativação:** Após a convolução e a camada de pooling, é comum aplicar uma função de ativação, como a função ReLU (Rectified Linear Unit), para introduzir não linearidade nas saídas das camadas convolucionais. Isso permite que a rede aprenda a representar relações complexas nos dados.

Há também a técnica de empilhamento de camadas, na qual as CNNs frequentemente consistem em várias camadas convolucionais dispostas em sequência. À medida que você avança na rede, cada camada convolucional extrai características progressivamente mais complexas. A camada inicial pode detectar características simples, como bordas e cores, enquanto as camadas subsequentes são capazes de identificar padrões mais elaborados e características mais intrincadas.

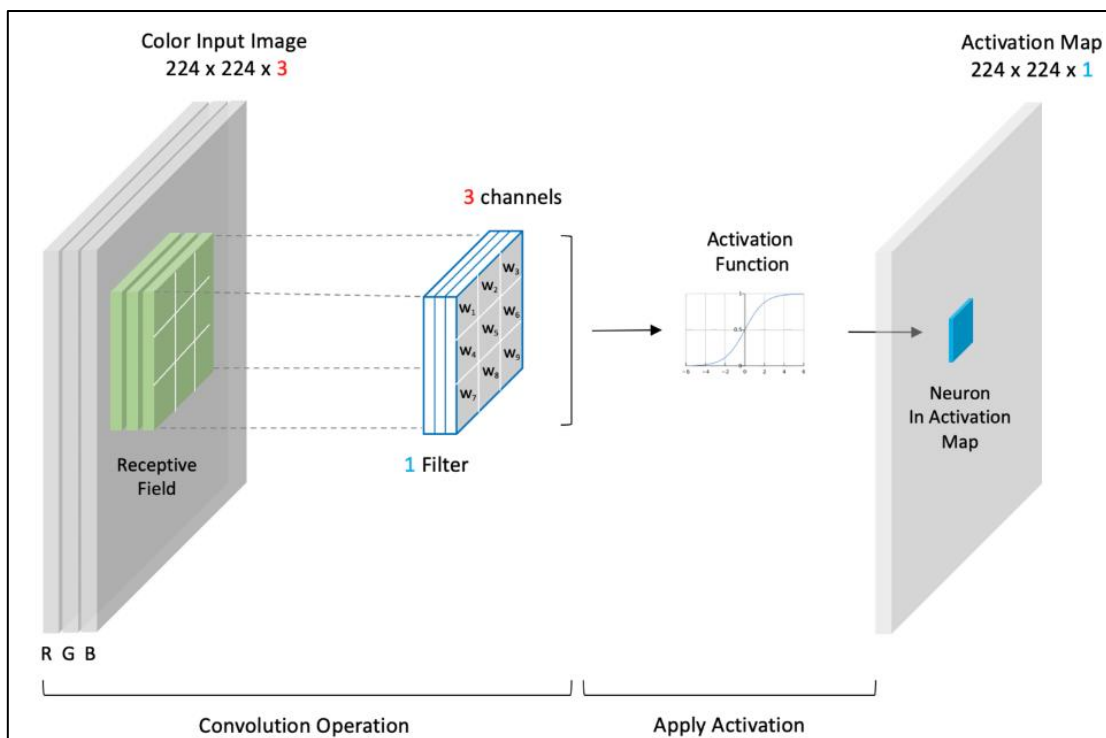


Figura 21. Etapa de Convolução. Fonte: LearnOpenCV.

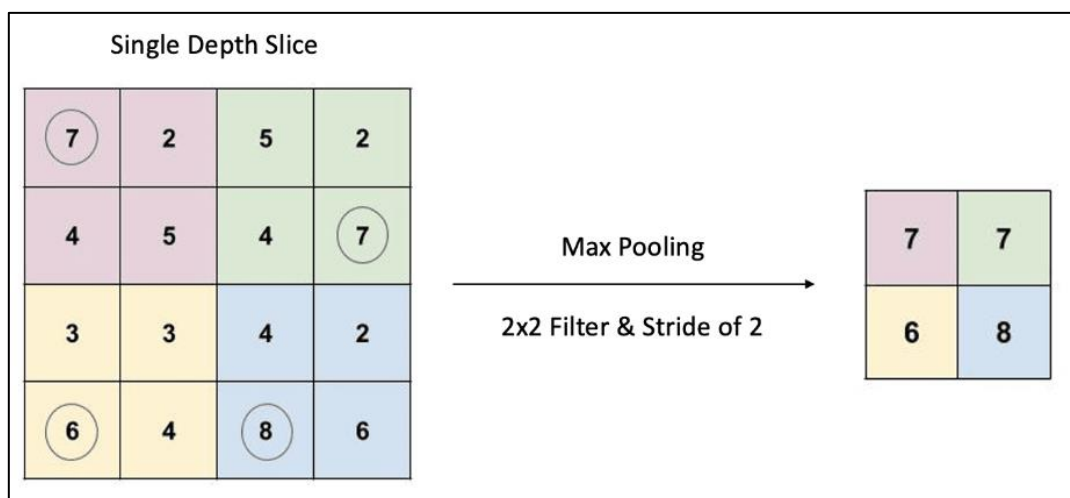


Figura 22. Etapa de Pooling. Fonte: OpenCV.

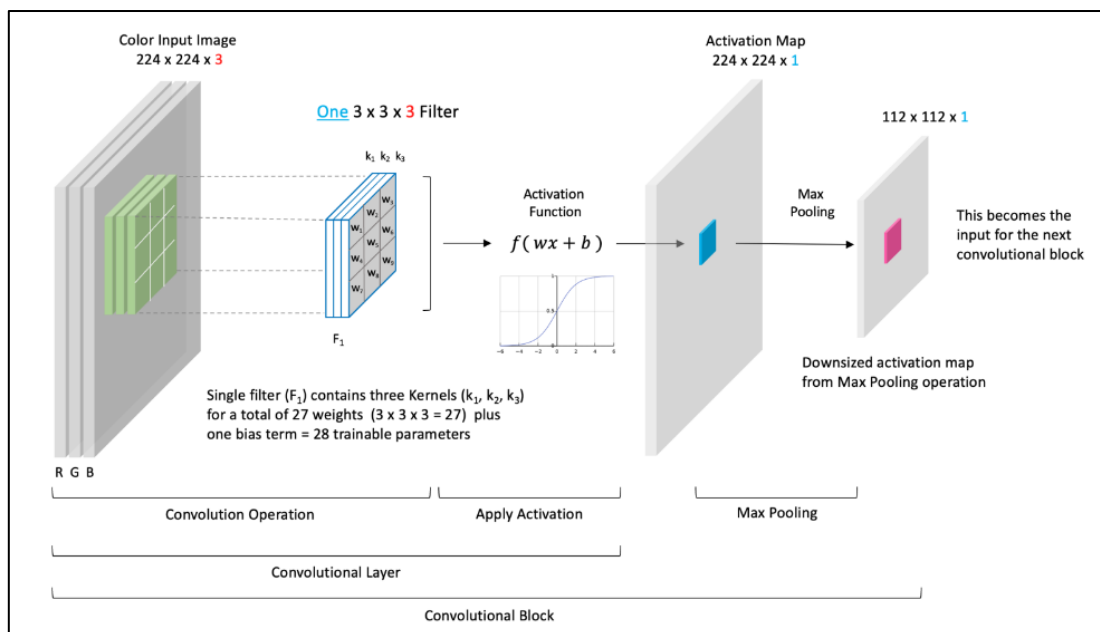


Figura 23. Camada convolucional completa. Fonte: LearnOpenCV.

Após o processamento pelas camadas convolucionais de uma Convolutional Neural Network (CNN), o formato dos dados deve estar em uma dimensão para servir de entrada nas camadas de conexão densas. Isso é feito através de camadas de "flatten", semelhante à função flatten do numpy.

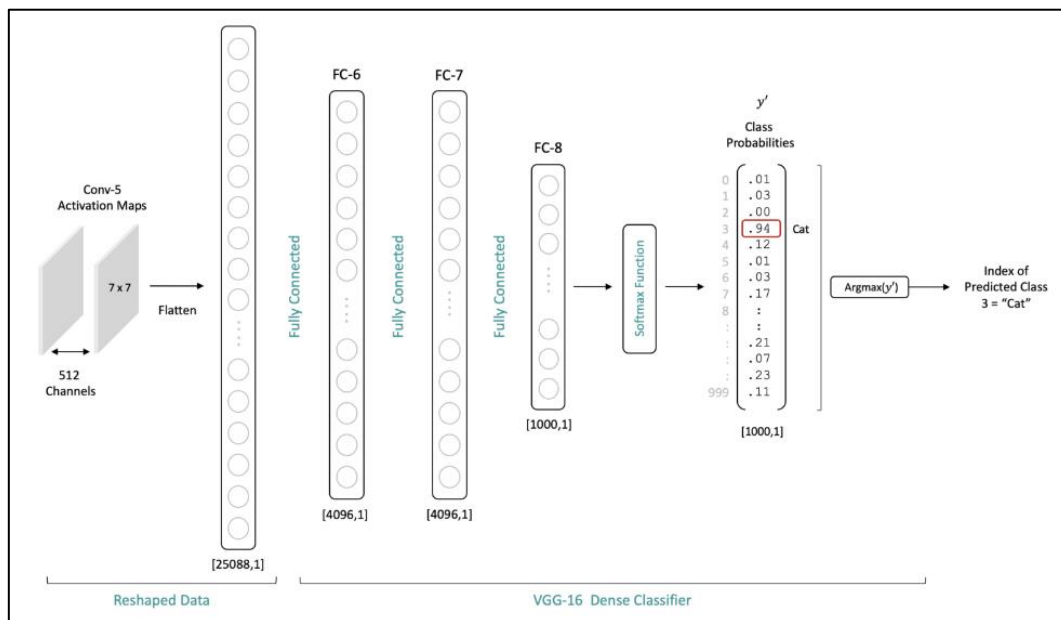


Figura 24. Formatação "flatten" e camadas densas. Fonte: LearnOpenCV.



## 2.3 *Tensorflow e Keras*

Para implementar uma CNN, vamos utilizar duas ferramentas chamadas Tensorflow e Keras. TensorFlow e Keras são duas das bibliotecas de código aberto mais populares para desenvolver e treinar modelos de aprendizado profundo, incluindo redes neurais artificiais. Eles são amplamente utilizados em projetos de aprendizado de máquina e aprendizado profundo devido à sua eficiência, flexibilidade e recursos poderosos.

O Tensorflow é uma biblioteca de código aberto desenvolvida pela equipe do Google Brain para implementar e treinar modelos de aprendizado de máquina e aprendizado profundo, fornecendo uma estrutura para criar grafos computacionais, onde as operações matemáticas são representadas como nós e as tensores (matrizes multidimensionais) fluem entre eles. Isso permite otimizações eficientes de cálculos em CPUs e GPUs.



Figura 25. Logo do Tensorflow. Fonte: Github.

Já o Keras é uma interface de alto nível para construir, treinar e avaliar modelos de aprendizado de máquina e aprendizado profundo. Por ser altamente modular, o Keras é conhecido por sua simplicidade e facilidade de uso. Ele fornece uma API intuitiva e orientada a objetos que permite aos desenvolvedores criar modelos de forma rápida e eficiente, com uma sintaxe mais amigável e menos complexa do que a API nativa do TensorFlow.



Figura 26. Logo do Keras. Fonte: Keras.

Para usar essas ferramentas, utilizaremos a biblioteca tensorflow importando os módulos do keras.

```
import tensorflow as tf
from keras import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten
```

Primeiro importamos o tensorflow com apelido “tf” para facilitar a codificação, depois importamos a classe "Sequential" do Keras. O modelo "Sequential" é uma forma de criar modelos de aprendizado profundo em que as camadas são empilhadas sequencialmente, uma após a outra. Isso é útil para a construção de redes neurais feedforward, onde os dados fluem de uma camada para a próxima.

Do módulo “layers”, importamos as camadas comumente usadas em redes neurais construídas com o Keras:

- **Dense:** Esta camada é uma camada totalmente conectada em que cada neurônio está conectado a todos os neurônios na camada anterior. É frequentemente usada nas camadas densas do modelo.
- **Conv2D:** Esta camada é usada para convoluções bidimensionais, comumente usadas em tarefas de visão computacional.
- **MaxPooling2D:** Esta camada é usada para operações de max pooling bidimensionais, que reduzem a dimensionalidade dos dados.
- **Dropout:** Esta camada é usada para a regularização do modelo, ajudando a prevenir o overfitting, desligando aleatoriamente um percentual de neurônios durante o treinamento.

- **Flatten:** Esta camada é usada para achatar dados multidimensionais em um vetor unidimensional, geralmente antes de passá-los para camadas densas.

O módulo do Keras contém bancos de dados que podem ser importados para realizarmos treinamentos de redes neurais. Seguiremos com o banco de imagens CIFAR-10, um conjunto de dados de 60 mil imagens com 10 classes, 6 mil imagens para cada classe. Dessas imagens 50 mil são para treino e 10 mil são para testes.

```
from tensorflow.keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print(X_train.shape)
```

Caso necessário, o módulo irá fazer o download dos dados automaticamente. As imagens têm o formato (32,32,3). De posse dos dados, faremos um pré-processamento para normalizá-los e transformá-los para a codificação "one-hot".

A codificação "one-hot" cria um vetor binário onde apenas um dos elementos é "quente" (1) e todos os outros são "frios" (0). Suponha que temos um conjunto de dados com três categorias de frutas: maçã, banana e laranja. Para representar essas categorias em uma codificação "one-hot", criaríamos vetores binários da seguinte maneira:

- Maçã: [1, 0, 0]
- Banana: [0, 1, 0]
- Laranja: [0, 0, 1]

Para o banco de dados CIFAR-10, as categorias são: Avião, Automóvel, Pássaro, Gato, Cervo, Cachorro, Sapo, Cavalo, Navio, Caminhão. Cada classe é representada por um número de 0 a 9, mas importaremos do módulo "utils" do Keras a função "to\_categorical" para codificar as classes em "one-hot".

```
from keras.utils import to_categorical
```

```
X_train = X_train.astype("float32") / 255
X_test = X_test.astype("float32") / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Agora vamos finalmente montar nossa arquitetura de rede neural convolucional. Abaixo temos o uso das camadas que importamos do Keras através do método “add”:

```
modelo = Sequential()
modelo.add(Conv2D(50, (3, 3), activation='relu', input_shape=(32, 32, 3)))
modelo.add(MaxPooling2D((2, 2)))
modelo.add(Dropout(0.2))
modelo.add(Flatten())
modelo.add(Dense(200, activation='relu'))
modelo.add(Dropout(0.3))
modelo.add(Dense(10, activation='softmax'))
```

Primeiro inicializamos uma variável para ser nosso modelo sequencial, pois as camadas serão empilhadas uma após a outra. Adicionamos a camada convolucional com 50 filtros com formatos 3x3, ativação “ReLU” e formato de entrada igual ao formato das imagens. Faz-se a operação de “pooling” máximo em formato 2x2 e “dropout” de 20% das conexões dos neurônios.

Entre a seção de extração de características e a seção de classificação fazemos o achatamento de dados com uma camada “Flatten”. Uma camada densa de 200 neurônios é adicionada, um “dropout” de 30% e por último, uma camada densa de 10 neurônios com ativação “softmax”, entregando o formato “one-hot” para 10 classes na saída.

Podemos observar o formato final da arquitetura criada com o método “summary”:

```
modelo.summary()
```

A saída fica da seguinte forma:

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	(None, 30, 30, 50)	1400
max_pooling2d (MaxPooling2D)	(None, 15, 15, 50)	0
dropout (Dropout)	(None, 15, 15, 50)	0
flatten (Flatten)	(None, 11250)	0
dense (Dense)	(None, 200)	2250200
dropout_1 (Dropout)	(None, 200)	0
dense_1 (Dense)	(None, 10)	2010
=====		
Total params: 2,253,610		
Trainable params: 2,253,610		
Non-trainable params: 0		

O próximo passo é compilar nosso modelo. Antes de prosseguirmos, nosso modelo necessita dessa etapa para configurar os detalhes de como o modelo será treinado, como o otimizador (algoritmo de treinamento), a função de perda (métrica de quão bem o modelo está indo) e as métricas de avaliação. O "compile" define as regras e objetivos do treinamento, permitindo que o modelo ajuste seus parâmetros de acordo com os dados de treinamento.

A seguir usamos o otimizador "Adam" (Adaptive Moment Estimation) para ajustar os parâmetros do modelo, a função de perda "categorical\_crossentropy" para medir o quão bem o modelo está indo (ideal para medir a distância para a codificação "one-hot") e a métrica "accuracy" para avaliar o desempenho.

```

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

```

## 2.4 Treinando e Salvando CNNs

Para treinar nosso modelo utilizamos o método “fit” passando os seguintes parâmetros:

- **X**: Dados de entrada;
- **y**: Dados de saída;
- **batch\_size**: Quantidade de dados a cada iteração de treinamento;
- **epochs**: Número de vezes que o conjunto de dados será mostrado ao modelo;
- **verbose**: Mostrar andamento do treino.
- **validation\_split**: Porcentagem do conjunto de dados para validação.

```
modelo.fit(  
    X_train,  
    y_train,  
    batch_size=256,  
    epochs=30,  
    verbose=1,  
    validation_split=0.2  
)
```

A saída do programa exibirá cada iteração com os dados (*epochs*) e mostrará métricas, conforme ilustrado na imagem a seguir. É notável que a função de perda (*loss*) diminui progressivamente ao longo das iterações, enquanto a acurácia aumenta gradualmente. É de extrema importância monitorar as métricas de validação (*val\_loss* e *val\_accuracy*) ao longo das iterações, pois se esses parâmetros não mostrarem melhora, enquanto apenas os parâmetros dos dados de treinamento continuam evoluindo, poderá indicar que o modelo está sofrendo de *overfitting*.

```

Epoch 1/30
157/157 [=====] - 17s 105ms/step - loss: 1.7134 - accuracy: 0.3844 - val_loss: 1.4307 - val_accuracy: 0.4976
Epoch 2/30
157/157 [=====] - 16s 99ms/step - loss: 1.3845 - accuracy: 0.5096 - val_loss: 1.2821 - val_accuracy: 0.5508
Epoch 3/30
157/157 [=====] - 16s 99ms/step - loss: 1.2561 - accuracy: 0.5570 - val_loss: 1.2091 - val_accuracy: 0.5796
Epoch 4/30
157/157 [=====] - 16s 99ms/step - loss: 1.1774 - accuracy: 0.5844 - val_loss: 1.1477 - val_accuracy: 0.5988
Epoch 5/30
157/157 [=====] - 16s 99ms/step - loss: 1.1106 - accuracy: 0.6093 - val_loss: 1.0993 - val_accuracy: 0.6169
Epoch 6/30
157/157 [=====] - 16s 99ms/step - loss: 1.0618 - accuracy: 0.6266 - val_loss: 1.0875 - val_accuracy: 0.6191
Epoch 7/30
157/157 [=====] - 16s 99ms/step - loss: 1.0221 - accuracy: 0.6390 - val_loss: 1.0522 - val_accuracy: 0.6341
Epoch 8/30
157/157 [=====] - 16s 99ms/step - loss: 0.9737 - accuracy: 0.6554 - val_loss: 1.0251 - val_accuracy: 0.6450
Epoch 9/30
157/157 [=====] - 16s 99ms/step - loss: 0.9435 - accuracy: 0.6680 - val_loss: 1.0194 - val_accuracy: 0.6494
Epoch 10/30
157/157 [=====] - 16s 99ms/step - loss: 0.9050 - accuracy: 0.6784 - val_loss: 1.0218 - val_accuracy: 0.6494

```

Figura 27. Treinamento em Tensorflow. Fonte: Autoria própria.

Podemos visualizar os dados ao adicionar um *callback* à função. O Tensorflow oferece uma interface gráfica que podemos visualizar o treino em tempo real chamada Tensorboard. Crie uma instância da classe *Tensorboard* passando o parâmetro de qual diretório os logs serão registrados, assim como o código abaixo:

```

from keras.callbacks import Tensorboard
tbCallback = Tensorboard('logs')
modelo.fit(
    ...
    callbacks=[tbCallback]
)

```

Esse código irá criar a pasta “logs” em seu diretório para salvar as informações do treino do modelo. Enquanto seu modelo está treinando acesse no VSCode o menu “View > Command Palette...” e busque por “Tensorboard”. Selecione a opção “Python: Launch Tensorboard” e em seguida “Select another folder”. Selecione a pasta “logs” e o painel do Tensorboard irá aparecer para avaliarmos as métricas em tempo real.

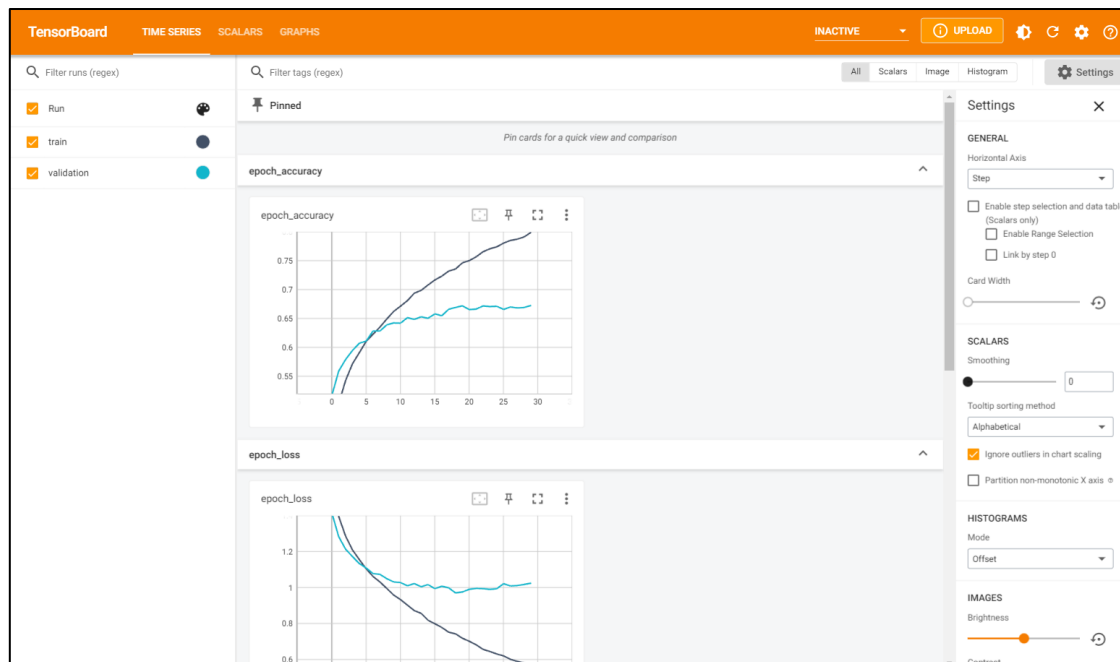


Figura 28. Painel do Tensorboard no VSCode. Fonte: autoria própria.

Para salvar o modelo treinado em um arquivo para uso posterior, podemos utilizar o método `save` da variável que contém nosso modelo, passando um parâmetro com o nome a ser salvo. Uma pasta será criada com este nome com as informações do seu modelo.

```
modelo.save('MinhaCNN')
```

## 2.5 Abrindo e Avaliando a CNN

Após treinar e salvar o nosso modelo, podemos utilizar a função `load_model` o módulo `models` para abrir o modelo treinado e utilizá-lo para fazer inferências.

```
from keras import models
modelo = models.load_model('MinhaCNN')
```

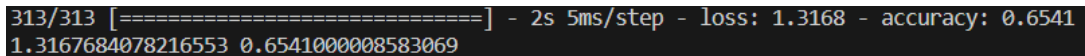
Vamos colocar o modelo para fazer inferências com os dados de teste que havíamos separado anteriormente. Para isso, utilizamos o método `evaluate`,



passando dois parâmetros: os dados de entrada de teste e a classificação desses dados. O retorno será uma tupla contendo a perda e a acurácia

```
test_loss, test_acc = modelo.evaluate(X_test, y_test)
print(test_loss, test_acc)
```

A saída no terminal irá mostrar a inferência sendo feita.



```
313/313 [=====] - 2s 5ms/step - loss: 1.3168 - accuracy: 0.6541
1.3167684078216553 0.6541000008583069
```

Figura 29. Avaliando o modelo com dados de teste. Fonte: autoria própria.

Mas e agora, como podemos usar a rede neural para classificar novas imagens? Para isso, podemos obter imagens do arquivo ou da câmera e utilizar o OpenCV para ajustar a imagem. O ajuste é feito para o tamanho de entrada da rede neural (32x32x3), em seguida, faz-se a normalização e por fim utilizamos o método *predict* com uma lista das imagens. Como é apenas uma imagem, encapsulamos a imagem para ser um *array numpy* multidimensional. Abaixo, um exemplo de uso da CNN com as imagens obtidas da Webcam.

```
import cv2 as cv
import numpy as np

modelo = models.load_model('MinhaCNN')
class_names = [
    'aviao',
    'carro',
    'passaro',
    'gato',
    'cervo',
    'cachorro',
    'sapo',
    'cavalo',
    'barco',
    'caminhao'
]

webcam = cv.VideoCapture(0, cv.CAP_DSHOW)
```

```

while True:
    _, frame = webcam.read()

    cnnInput = cv.resize(frame, (32, 32))
    cnnInput = cnnInput.astype('float32') / 255
    inferencia = modelo.predict(np.array([cnnInput]))
    idx = np.argmax(inferencia[0])
    if inferencia[0, idx] > 0.8:
        texto = class_names[idx] + ' ' + str(inferencia[0, idx])
        cv.putText(frame, texto, (10, 20), cv.FONT_HERSHEY_PLAIN, 1, (0, 255,
0), 2)

    cv.imshow('Webcam', frame)
    tecla = cv.waitKey(1)
    if tecla == ord('q'):
        break
    cv.destroyAllWindows()

```

## 2.6 Exercício

1. Faça um modelo de rede neural convolucional que classifique suas emoções na webcam, por exemplo: sorrindo, surpreso e neutro. Experimente acrescentar mais camadas convolucionais para melhorar seus resultados.

Após a execução da atividade prática, faça upload dos *arquivos* “.py” em formato “.zip”, e suba na plataforma do fiteducação (edX).

### 3 Detecção de Objetos com YOLO

YOLO (You Only Look Once) é um avançado e amplamente usado algoritmo de detecção de objetos em imagens e vídeos criado e mantido pela Ultralytics. Ele é especialmente popular em visão computacional e aplicações de aprendizado de máquina. YOLO é notável por sua eficiência e capacidade de realizar detecção de objetos em tempo real, justamente por ser feito para realizar o processamento da imagem uma só vez, daí o nome do inglês “Você Olha Apenas Uma Vez”.



Figura 30. Logo da Ultralytics. Fonte: Ultralytics.

Com o YOLO podemos fazer classificação, detecção de objetos, segmentação, rastreamento e detecção de poses. Em nosso módulo, focaremos na parte de detecção de objetos, em que temos a posição retangular do objeto detectado.

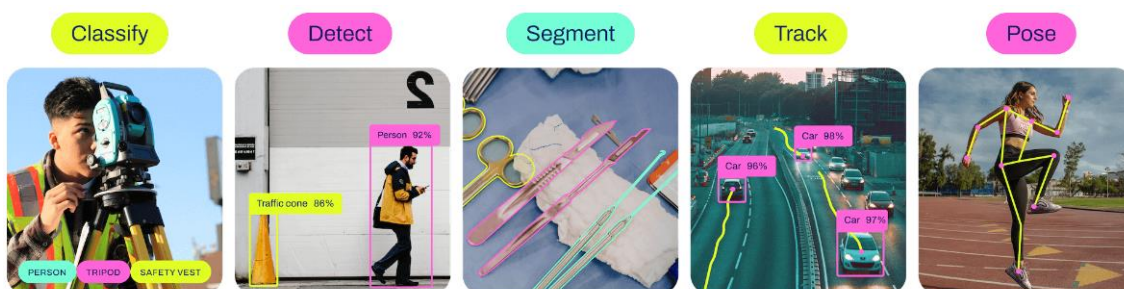


Figura 31. Tarefas disponíveis no YOLO. Fonte: Ultralytics.

Algumas das características presentes do algoritmo YOLO:

- Detecção de Objetos em Tempo Real: YOLO é conhecido por sua capacidade de realizar detecção de objetos em tempo real, tornando-

o adequado para sistemas de vigilância, veículos autônomos e muito mais.

- Uma Única Passagem: YOLO realiza a detecção de objetos em uma única passagem pela imagem, ao contrário de abordagens anteriores que requerem várias passagens. Isso o torna mais eficiente.
- Detecção Multiclasse: YOLO pode detectar múltiplos objetos de diferentes classes em uma única imagem. Ele é usado em aplicações que envolvem a detecção de várias categorias de objetos.
- Precisão e Velocidade: YOLO equilibra precisão e velocidade. Ele é capaz de alcançar resultados competitivos em termos de precisão, ao mesmo tempo que oferece desempenho em tempo real.
- Flexibilidade: YOLO pode ser treinado para detectar objetos de categorias personalizadas, tornando-o flexível para uma variedade de aplicações.



Figura 32. Detecção de objetos com YOLO. Fonte: Ultralytics.

O YOLO já passou por inúmeras versões. Hoje a versão que oferece os melhores resultados em termos de precisão e rapidez é a versão 8 (YOLOv8). A Ultralytics possui modelos pretreinados com diferentes tamanhos para uso imediato.

Model	size (pixels)	mAP <sup>val</sup> 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figura 33. Modelos pretreinados YOLOv8. Fonte: Ultralytics.

### 3.1 Modelos YOLOv8

Para baixar um modelo, utilizamos a classe YOLO da biblioteca *ultralytics*, passando o nome do modelo seguido da extensão *yaml* ou *pt*. A extensão *yaml* irá construir um modelo novo para ser treinado enquanto a extensão *pt* irá criar um modelo com os pesos pretreinados.

```
from ultralytics import YOLO

modelo = YOLO('yolov8n.yaml') # modelo novo a ser treinado
modelo = YOLO('yolov8n.pt') # modelo pretreinado
modelo = YOLO('yolov8n.yaml').load('yolov8n.pt') # transferência de pesos
```

O modelo criado pode ser treinado com o método *train*, utilizando um banco de imagens comum na validação de modelos de detecção de objetos chamado COCO. Como mostrado abaixo, são passados parâmetros dos dados (*data*), quantas iterações a ser feitas (*epochs*) e o tamanho da imagem (*imgsz*).

```
resultados = modelo.train(data='coco128.yaml', epochs=100, imgsz=640)
```

O treino pode ser acompanhado utilizando o Tensorboard de forma semelhante ao capítulo anterior. O diretório com os dados de treino em tempo real será criado em “*runs/detect/train*”. Ao final, utiliza-se o comando *export* para salvar o modelo, ao qual irá salvar o modelo com o nome no qual foi aberto anteriormente (*'yolov8n'* no nosso exemplo até aqui).

```
modelo.export()  
modelo.export(format='saved_model') # salva em formato do keras
```

### 3.2 Inferências com modelos

A grande vantagem dos modelos YOLO é que um modelo pode receber como entrada os dados das imagens de inúmeras formas: o caminho da imagem do arquivo, uma imagem aberta pelo OpenCV, uma pasta de imagens, um *screenshot* atual da tela, um link da internet e até mesmo de um vídeo do Youtube.

```
from ultralytics import YOLO  
modelo = YOLO('yolov8n.pt')  
  
source = 'path/to/image.jpg' # imagem do arquivo  
source = 'screen' # screenshot  
source = cv2.imread('path/to/image.jpg') # OpenCV  
source = 'https://ultralytics.com/images/bus.jpg'  
  
resultados = modelo(source)
```

Quando se trata de dados em streaming, no caso de vídeos, devemos passar o parâmetro *stream* para *True*.

```
from ultralytics import YOLO  
modelo = YOLO('yolov8n.pt')  
  
source = 'path/to/video.mp4' # vídeo  
source = 'path/to/dir' # diretório com imagens e vídeos  
source = 'https://youtu.be/LNwODJXcvt4' # Youtube  
  
resultados = modelo(source, stream=True)
```

Cada elemento na lista de resultados é referente à uma imagem, portanto, se a inferência foi feita em uma imagem somente, a lista de resultados conterá

apenas um elemento. Já no caso de um vídeo, por exemplo, a lista de resultados conterá um elemento para cada *frame*.

Para visualizar a imagem com as inferências desenhadas, utilizamos a função *plot* de um elemento da lista de resultados. Esta função irá devolver uma imagem desenhada com os retângulos dos objetos encontrados, bem como o nome e o nível de confiança.

```
from ultralytics import YOLO
import cv2 as cv

modelo = YOLO('yolov8n.pt')
source = 'https://ultralytics.com/images/bus.jpg'
resultados = modelo(source)

cv.imshow('onibus', resultados[0].plot())
cv.waitKey(0)
```

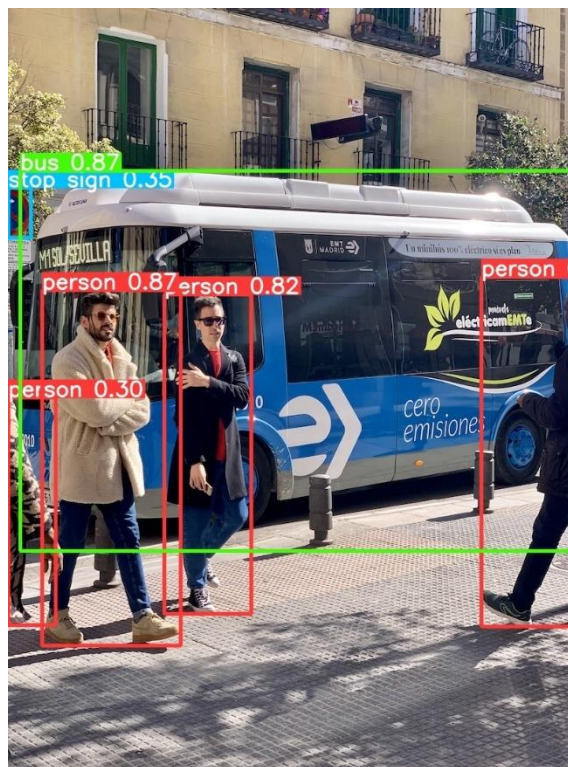


Figura 34. Inferência em imagem com YOLOv8. Fonte: Ultralytics.

No caso de um vídeo de Youtube, podemos obter os resultados e ir mostrando os desenhos à cada imagem da seguinte forma.

```

from ultralytics import YOLO
import cv2 as cv

model = YOLO('yolov8n.pt')

source = 'https://youtu.be/LNwODJXcvt4'
resultados = model(source, stream=True)

for resultado in resultados:
    cv.imshow('frame', resultado.plot())
    tecla = cv.waitKey(1)
    if tecla == ord('q'):
        break

```

Existe também a possibilidade de obter os dados de cada resultado para termos as coordenadas encontradas por exemplo. Utilize a propriedade *boxes* para obter uma lista com todos os retângulos dos objetos encontrados. Para cada elemento em *boxes*, utilize as propriedades abaixo:

- **xyxy**: Coordenadas do retângulo;
- **cls**: Classe encontrada, em formato de índice;
- **conf**: Nível de confiança da inferência.

E por fim, utilize a propriedade *names* presente no modelo para obter o nome em texto da classe encontrada na inferência. Segue abaixo um exemplo utilizando a webcam.

```

from ultralytics import YOLO
import cv2 as cv

model = YOLO('yolov8n.pt')
webcam = cv.VideoCapture(0, cv.CAP_DSHOW)

while True:
    _, frame = webcam.read()
    resultados = model(frame)

    resultado = resultados[0]
    boxes = resultado.boxes

```



```

for box in boxes:
    if box.conf[0] < 0.8:
        continue
    x1, y1, x2, y2 = box.xyxy[0]
    x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)
    cv.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 1)
    texto = model.names[int(box.cls[0])] + ' ' +
str(round(float(box.conf[0]), 2))
    cv.putText(frame, texto, (x1, y1-2), cv.FONT_HERSHEY_PLAIN, 1, (0, 0,
0), 1)

cv.imshow('frame', frame)
tecla = cv.waitKey(1)

if tecla == ord('q'):
    break

```

Mas se não houver a necessidade de obter os dados separadamente, um modelo YOLO pode ser utilizado na webcam de forma bem mais simples da seguinte forma.

```

from ultralytics import YOLO
model = YOLO('runs/detect/train9/weights/best.pt')
model(source=0, show=True, conf=0.7)

```

Onde “source=0” é a webcam (semelhante à classe VideoCapture do OpenCV), “show=True” é a janela a ser mostrada (semelhante à função “imshow” do OpenCV) e “conf=0.7” é o nível de confiança mínimo a ser mostrado.

### 3.3 Validação no YOLOv8

Uma grande vantagem de utilizar os modelos YOLO é também a facilidade de fazer validação do modelo, pois o próprio modelo lembra as configurações do treino. Com o modelo carregado é possível utilizar métricas de acurácia já inclusas na função *val*.

```

from ultralytics import YOLO

```

```

model = YOLO('yolov8n.pt')

metrics = model.val()
metrics.box.map    # map50-95
metrics.box.map50  # map50
metrics.box.map75  # map75
metrics.box.maps   # uma lista com map50-95 de cada categoria

```

O "mAP" (mean Average Precision) é uma métrica amplamente usada para avaliar o desempenho de modelos de detecção de objetos, a qual combina várias métricas para fornecer uma medida geral da precisão da detecção de objetos. O método *val* irá analisar as imagens e trará as métricas para cada classe.

Class	Images	Instances	Box(P)	R	mAP50	mAP50-95
all	5000	36335	0.633	0.475	0.521	0.371
person	5000	10777	0.754	0.673	0.745	0.514
bicycle	5000	314	0.687	0.392	0.457	0.265
car	5000	1918	0.646	0.515	0.561	0.364
motorcycle	5000	367	0.71	0.58	0.655	0.413
airplane	5000	143	0.814	0.766	0.832	0.653
bus	5000	283	0.746	0.643	0.739	0.62
train	5000	190	0.798	0.77	0.834	0.646
truck	5000	414	0.549	0.399	0.435	0.293
boat	5000	424	0.583	0.3	0.377	0.21
traffic light	5000	634	0.644	0.345	0.409	0.211
fire hydrant	5000	101	0.85	0.703	0.774	0.609
stop sign	5000	75	0.695	0.64	0.692	0.63
parking meter	5000	60	0.631	0.5	0.558	0.441
bench	5000	411	0.57	0.258	0.296	0.193
bird	5000	427	0.686	0.358	0.425	0.278
cat	5000	202	0.776	0.824	0.856	0.652
dog	5000	218	0.656	0.701	0.729	0.591
horse	5000	272	0.7	0.653	0.693	0.525
sheep	5000	354	0.609	0.667	0.662	0.46
cow	5000	372	0.697	0.601	0.682	0.487
elephant	5000	252	0.702	0.833	0.821	0.629
bear	5000	71	0.847	0.775	0.842	0.689
zebra	5000	266	0.808	0.797	0.882	0.658
giraffe	5000	232	0.857	0.828	0.885	0.683
backpack	5000	371	0.489	0.156	0.197	0.1
umbrella	5000	407	0.631	0.504	0.542	0.359

Figura 35. Avaliação do modelo. Fonte: Autoria própria.

O mAP é valioso porque leva em consideração não apenas a precisão, mas também a capacidade do modelo de recuperar objetos (recall) e é calculada para cada classe de objeto separadamente. Isso significa que você obtém uma visão detalhada do desempenho do modelo em relação a diferentes categorias de objetos. Já número após o nome "mAP" refere-se ao limiar de confiança aceito no cálculo. Por exemplo, a métrica "mAP50" aceita todos os objetos encontrados com 50% de confiança ou mais.

### 3.4 Treinando com outras imagens

Para que o modelo seja capaz de identificar e prever objetos, é necessário seguir um processo de preparação das imagens, que envolve a criação de anotações para os objetos presentes nas imagens. Para isso, é crucial criar um arquivo de texto com o mesmo nome de cada imagem no conjunto de dados, seguindo um formato específico.

Em cada linha desse arquivo de texto, são inseridos valores que descrevem o objeto a ser detectado. Esses valores consistem em um número que representa a classe à qual o objeto pertence e quatro números normalizados que definem as coordenadas do retângulo de delimitação do objeto na imagem.

Por exemplo, se desejamos marcar uma pessoa na imagem e atribuímos a classe "pessoa" ao número "0", as anotações para essa pessoa podem ser definidas da seguinte forma: "0 0.2 0.25 0.3 0.35". Nesse caso, o "0" representa a classe "pessoa", e os quatro números seguintes indicam as coordenadas normalizadas que definem o retângulo de marcação, representando a posição (coordenadas x e y) e o tamanho da área onde a pessoa está localizada na imagem (largura e altura).

```
6 0.44053125 0.511166666666667 0.128640625 0.4811666666666674  
5 0.5369843750000001 0.523145833333333 0.1095312500000004 0.4566041666666666
```

Figura 36. Arquivo txt para treino com YOLO. Fonte: Autoria própria.

Ao invés de abrirmos as imagens e ver as coordenadas para criar as marcações manualmente, vamos utilizar uma ferramenta online chamada **Roboflow**.



Figura 37. Ferramenta Roboflow. Fonte: Roboflow.

Cadastre-se e acesse o banco de imagem do link [https://universe.roboflow.com/hoangdinhdan/fire\\_dataset\\_update1](https://universe.roboflow.com/hoangdinhdan/fire_dataset_update1) ou procure na seção “Universe” por “fire dataset”. Utilizaremos um banco de imagens para detectar fogo e fumaça. Clique em “Download this Dataset” para obter as imagens, utilizaremos a biblioteca do *Roboflow* para fazer o download já com anotações. Caso fosse preciso editar as anotações, seria necessário criar um projeto, clonar as imagens para esse projeto e então editar as anotações.

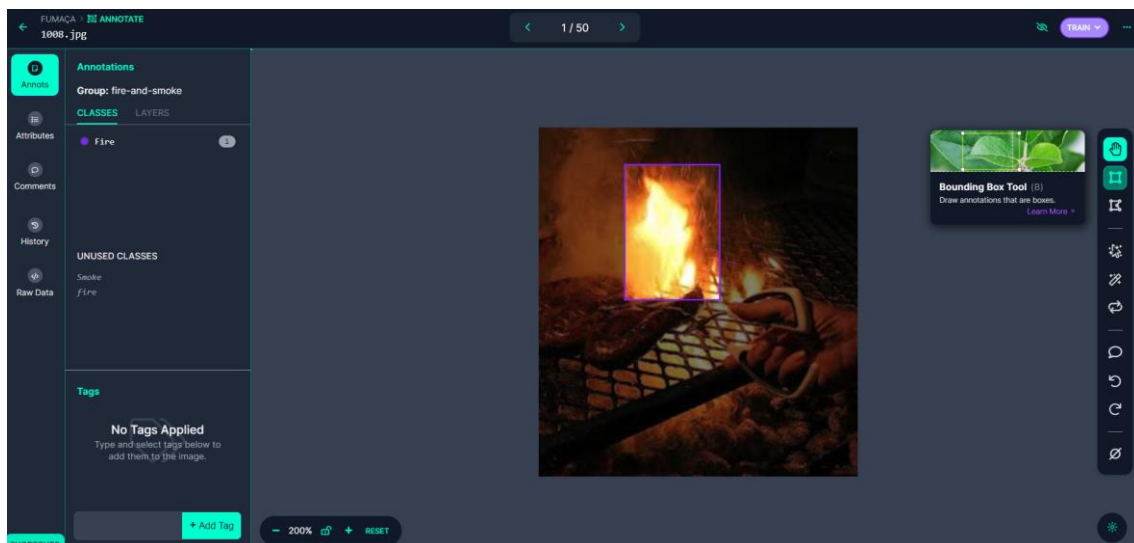


Figura 38. Anotação de objetos no Roboflow. Fonte: Autoria própria.

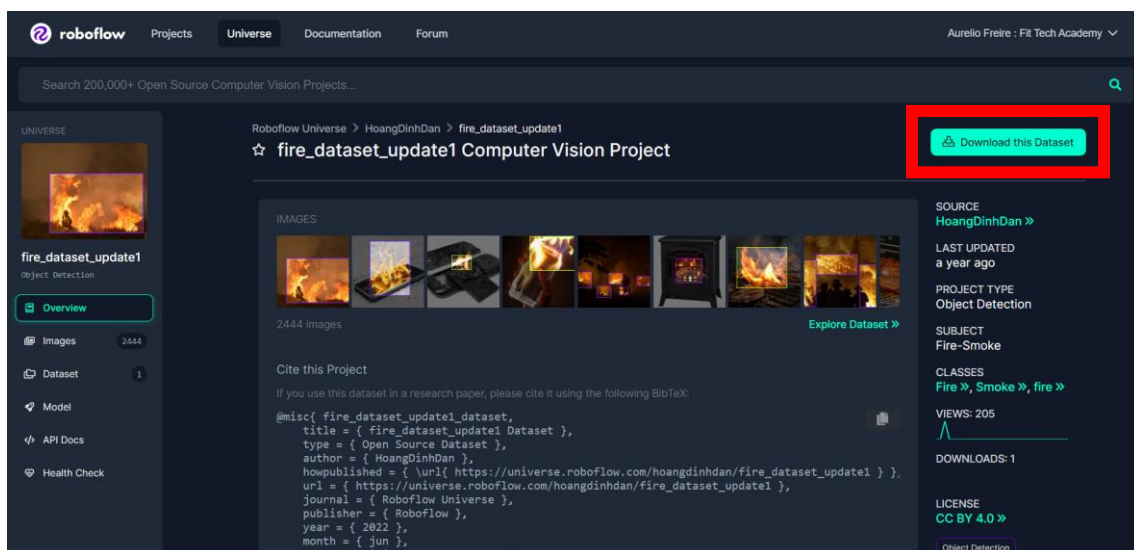


Figura 39. Banco de imagens de fogo e fumaça. Fonte: Roboflow.

Clique em “YOLOv8 -> Continue” e copie o código da aba “Jupyter” para seu código python.



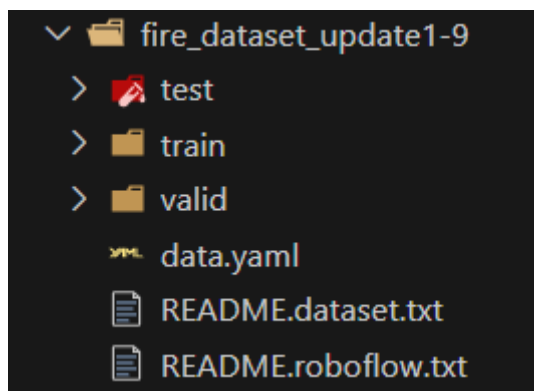


Figura 42. Diretório de imagens “fire\_dataset\_pudate1-9”. Fonte: Autoria própria.

Para treinar o modelo, utilize o comando *train* passando o arquivo YAML como dados.

```
from ultralytics import YOLO

model = YOLO('yolov8n.pt')
model.train(data='fire_dataset_update1-9/data.yaml', epochs=10,
imgsz=640)
```

Os dados do treino estarão na pasta “runs/detect/train”. Caso mais treinos sejam feitos, haverá pastas para cada treino. Dentro da pasta do treino podemos obter os melhores pesos encontrados em “weights/best.pt”.

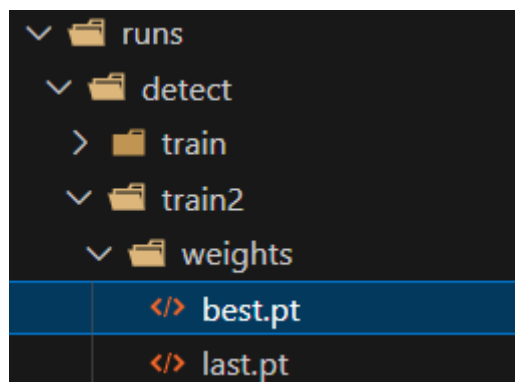


Figura 43. Melhores pesos do treino 2. Fonte: Autoria própria.

Para utilizar os pesos encontrados, crie um novo objeto YOLO a partir desses pesos, como no código abaixo.

```
model = YOLO('runs/detect/train2/weights/best.pt')
```

## 4 Projeto Final

Utilize todo o conhecimento adquirido na trilha de Visão Computacional para realizar um projeto de processamento de imagens. Seguem algumas sugestões:

- Braço robótico para separar blocos coloridos;
- Mira a laser automática;
- Liberação de portaria de condomínio;
- Robô seguidor de faixa colorida;

## Conclusão

A abordagem desta apostila permite ao leitor uma interação ativa com as ferramentas de processamento de imagem e visão computacional disponíveis na tecnologia apresentada. Ensinar como inspecionar imagens e extrair informações, para dar ao aluno autonomia, estímulo, senso crítico e contribuir para uma aprendizagem mais efetiva.

Começamos com uma introdução ao aprendizado de máquina para aprendermos os principais conceitos de inteligência artificial em visão computacional com a biblioteca Scikit Learn. Depois partimos para as redes neurais convolucionais, onde aprendemos a melhor arquitetura de redes neurais para imagens para conseguirmos extrair informações futuramente. Em seguida, abordou-se um dos melhores algoritmos modernos de detecção de objetos em visão computacional, o YOLO. E por fim, ao final foi proposto um projeto final englobando todos os conhecimentos apresentados na trilha de Visão Computacional.

Assim, o conteúdo do curso Aplicações com Inteligência Artificial familiariza o leitor com o mundo tecnológico, onde a virtualização ganha, cada vez mais, espaço e agrada as empresas por reduzir custos e aumentar as margens de lucro (adaptar a frase para seu curso). Obrigada por fazer parte do curso e ter realizado a leitura desta apostila. Espero que o interesse pela Visão Computacional Avançado apresentado neste curso esteja aguçado, para que você pratique e conheça ainda mais as suas maravilhas.



## Referências

“Undertanding Convolutional Neural Network (CNN): A Complete Guide”. Disponível em> <https://learnopencv.com/understanding-convolutional-neural-networks-cnn/>

Barelli, F. (2019). Introdução à Visão Computacional – Uma abordagem prática com Python e OpenCV. Casa do Código. São Paulo.

Documentação da biblioteca OpenCV para Python. (2022). Disponível em> [https://docs.opencv.org/4.x/d6/d00/tutorial\\_py\\_root](https://docs.opencv.org/4.x/d6/d00/tutorial_py_root).

Documentação da biblioteca Scikit-Learn. Disponível em> <https://scikit-learn.org/stable/>

Documentação da biblioteca Tensorflow. (2023). Disponível em> [https://www.tensorflow.org/api\\_docs](https://www.tensorflow.org/api_docs)

Documentação do algoritmo YOLOv8. Disponível em> <https://docs.ultralytics.com/tasks/detect/>

Documentação do módulo Keras. Disponível em> <https://keras.io/>

Livro Online *Deep Learning Book*. (2023). Disponível em> <https://www.deeplearningbook.com.br/introducao-as-redes-neurais-convolucionais/>

**CONTROLE DE REVISÃO DO DOCUMENTO / DOCUMENT REVISION  
CONTROL**

Revisão	Descrição	Razão	Autor	Data
A	Elaboração e revisão inicial	Elaboração inicial/Revisão pedagógica	Aurelio Aquino/Luana Forte	24/10/23



**BOM CURSO!**