

INTEGRAÇÃO DE ROBÓTICA COLABORATIVA COM IA

TM Cobot

Jailson Bina da Silva – Engenheiro I

Aurélio Aquino - Engenheiro III

Autores da apostila

Jailson Bina da Silva – Engenheiro I

Instrutores do curso

Luana Dias Pena Forte – Assistente de Suporte Pedagógico

Revisão da apostila

Fit Instituto de Tecnologia

Manaus, Outubro de 2023.

Autores



Jailson Bina da Silva

- Engenheiro de Controle e Automação pelo IFAM;
- FIT desde 2022;
- Experiência com aplicações de automação industrial e IoT.



Aurélio Aquino

- Engenheiro da Computação pela UFAM;
- FIT desde 2022;
- Pós-graduação *lato sensu* em *Lean Manufacturing* e MBA em Gestão de Projetos Ágeis pelo IDAAM;
- Aluno na California State University de Sacramento – Califórnia por um ano.
- Experiência na área de automação industrial, robótica, visão computacional e desenvolvimento de *software*.

APRESENTAÇÃO

A presente apostila é um instrumento teórico que complementa o curso de capacitação de Robô Colaborativo, promovido pelo FIT-Instituto de Tecnologia, para colaboradores do FIT, Flextronics e comunidade. Nela, veremos as principais ferramentas de IA, componentes e configurações para a integração com os robôs colaborativos TM5-900 e TM-12 da Omron. Este material é baseado em artigos científicos, periódicos, revistas científicas e livros científicos. 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 3 seções, iniciando com uma introdução sobre algoritmos de aprendizado, visão computacional, aprendizagem profunda e exercícios.

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

Boa Leitura !!

Sumário

1. INTRODUÇÃO AOS ALGORITMOS DE APRENDIZADO	6
1.1 O QUE É APRENDIZADO DE MÁQUINA.....	6
1.2 POR QUE USAR A APRENDIZAGEM AUTOMÁTICA?	6
1.3 TIPOS DE SISTEMAS DE APRENDIZADO DE MÁQUINA	10
1.3.1 <i>Aprendizagem supervisionada</i>	10
1.3.2 <i>Aprendizagem não supervisionada</i>	12
1.3.3 <i>Aprendizado por reforço</i>	15
1.4 APRENDIZADO DE MÁQUINA APLICADO COM PYTHON.....	16
1.4.1 <i>Scikit-Learn</i>	16
1.5 EXERCÍCIO PRÁTICO	22
1.6 EXERCÍCIO PRÁTICO.....	22
1.7 EXERCÍCIO PRÁTICO.....	22
2. VISÃO COMPUTACIONAL	23
2.1. O QUE É PERCEPÇÃO VISUAL?	23
2.2. SISTEMAS DE VISÃO	23
2.2.1. <i>Sistemas de Visão Humana</i>	24
2.2.2. <i>Sistemas de Visão AI</i>	25
2.3. DISPOSITIVOS DE DETECÇÃO.....	25
2.4. DISPOSITIVOS DE INTERPRETAÇÃO	26
2.5. APLICAÇÕES DE VISÃO COMPUTACIONAL	28
2.5.1. <i>Classificação de imagens</i>	29
2.5.2. <i>Detecção e localização de objetos</i>	31
2.5.3. <i>Geração de arte (transferência de estilo)</i>	31
2.5.4. <i>Criando imagens</i>	32
2.5.5. <i>Reconhecimento facial</i>	33
2.6. OPENCV - BIBLIOTECA DE VISÃO COMPUTACIONAL.....	34
2.6.1. <i>Primeiros passos com imagens</i>	35
2.7. EXERCÍCIO PRÁTICO	36
2.8. PROCESSAMENTO DE IMAGENS	36
2.9. EXERCÍCIO PRÁTICO.....	37
3. APRENDIZAGEM PROFUNDA E REDES NEURAIS	39
3.1. O QUE É APRENDIZAGEM PROFUNDA?	39
3.1.1. <i>Perceptron</i>	41
3.1.2. <i>Como o perceptron aprende?</i>	44
3.1.3. <i>Um neurônio é suficiente para resolver problemas complexos?</i>	45

3.1.4.	<i>Perceptrons multicamadas</i>	47
3.1.5.	<i>Arquitetura de perceptron multicamada</i>	48
3.1.6.	<i>Funções de ativação</i>	49
3.2	EXERCÍCIO PRÁTICO (OPCIONAL)	52
3.3	REDE NEURAL CONVOLUCIONAL (CNN).....	52
3.3.1	<i>Diferentes Camadas em uma CNN</i>	52
3.3.2	<i>Arquiteturas de CNN</i>	55
3.3.3	<i>Aplicação do Tensorflow e Keras em Modelos CNN</i>	56
3.3.4	<i>Métodos para Treinar e Salvar CNNs</i>	61
3.3.5	<i>Carregando um Modelo de CNN Pré-treinado</i>	63
3.4	EXERCÍCIO PRÁTICO (OPCIONAL)	65
3.5	YOU ONLY LOOK ONCE (YOLO)	65
3.5.1	<i>Modelos YOLOv8</i>	67
3.5.2	<i>Inferências com modelos</i>	68
3.5.3	<i>Validação no YOLOv8</i>	71
3.5.4	<i>Treinando com outras imagens</i>	73
3.6	EXERCÍCIO PRÁTICO (OPCIONAL)	77
4.	SITUAÇÃO PROBLEMA (OPCIONAL)	78
	CONCLUSÃO	79
	REFERÊNCIAS	80
	CONTROLE DE REVISÃO DO DOCUMENTO / DOCUMENT REVISION CONTROL	81

1. Introdução aos algoritmos de aprendizado

1.1 O que é aprendizado de máquina

O aprendizado de máquina é a ciência (e a arte) de programar computadores para que eles possam aprender com os dados.

O termo "aprendizado de máquina" foi relatado pela primeira vez em 1959 por Arthur Samuel. No entanto, Tom M. Mitchell forneceu uma definição bem proposta dos algoritmos usados no campo do aprendizado de máquina. Sua definição é a seguinte: "Seu filtro de spam é um programa de aprendizado de máquina que, com exemplos de e-mails de spam (sinalizados pelos usuários) e exemplos de e-mails normais (não spam, também chamados de "ham"), pode aprender a sinalizar spam. Os exemplos que o sistema usa para aprender são chamados de conjunto de treinamento. Cada exemplo de treinamento é chamado de instância de treinamento (ou amostra). A parte de um sistema de aprendizado de máquina que aprende e faz previsões é chamada de modelo. As redes neurais e as florestas aleatórias são exemplos de modelos."

Neste caso, a **tarefa T** é sinalizar spam para novos e-mails, a **experiência E** são os dados de treinamento e a medida de **desempenho P** precisa ser definida; por exemplo, você pode usar a proporção de e-mails classificados corretamente. Essa medida de desempenho específica é chamada de precisão e é usada com frequência em tarefas de classificação.

Se você simplesmente fizer o download de uma cópia de todos os artigos da Wikipédia, seu computador terá muito mais dados, mas não ficará subitamente melhor em nenhuma tarefa. Isso não é aprendizado de máquina.

1.2 Por que usar a aprendizagem automática?

Considere como você escreveria um filtro de spam usando técnicas de programação tradicionais, conforme ilustrado na Figura 1:

1. Primeiro, você examinaria a aparência típica do spam. Você pode notar que algumas palavras ou frases (como "4U", "cartão de crédito", "grátis" e "incrível") tendem a aparecer muito na linha de

assunto. Talvez você também perceba alguns outros padrões no nome do remetente, no corpo do e-mail e em outras partes do e-mail.

2. Você escreveria um algoritmo de detecção para cada um dos padrões que observou, e seu programa sinalizaria os e-mails como spam se vários desses padrões fossem detectados.
3. Você testaria seu programa e repetiria as etapas 1 e 2 até que ele estivesse bom o suficiente para lançar.

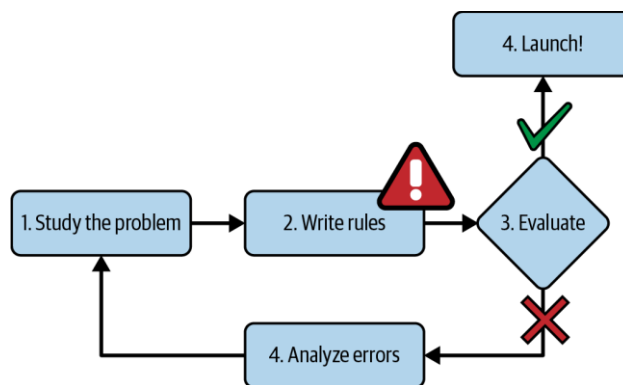


Figura 1. A abordagem tradicional. Fonte: RUSSANO; AVELINO, 2020.

Como o problema é difícil, seu programa provavelmente se tornará uma longa lista de regras complexas, muito difíceis de manter.

Por outro lado, um filtro de spam baseado em técnicas de aprendizado de máquina aprende automaticamente quais palavras e frases são boas preditoras de spam, detectando padrões de palavras com frequência incomum nos exemplos de spam em comparação com os exemplos de spam ilustrado na Figura 2. O programa é muito mais curto, mais fácil de manter e, provavelmente, mais preciso.

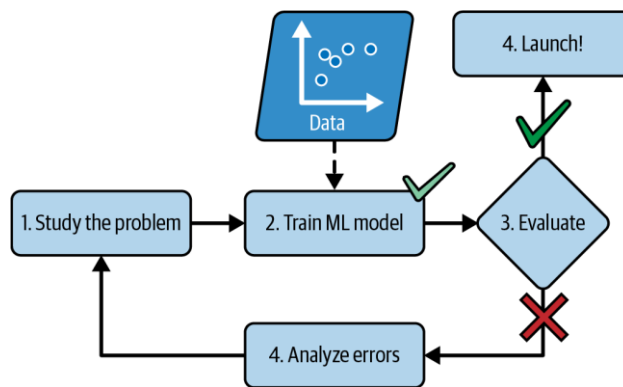


Figura 2. A abordagem de aprendizado de máquina. Fonte: RUSSANO; AVELINO, 2020.

E se os remetentes de spam perceberem que todos os seus e-mails contendo "4U" estão bloqueados? Eles podem começar a escrever "For U" em vez disso. Um filtro de spam que usa técnicas tradicionais de programação precisaria ser atualizado para sinalizar os e-mails "For U". Se os remetentes de spam continuarem a contornar seu filtro de spam, você precisará continuar escrevendo novas regras para sempre.

Por outro lado, um filtro de spam baseado em técnicas de aprendizado de máquina percebe automaticamente que "For U" se tornou incomumente frequente nos spams sinalizados pelos usuários e começará a sinalizá-los sem a sua intervenção, visualizado na Figura 3.

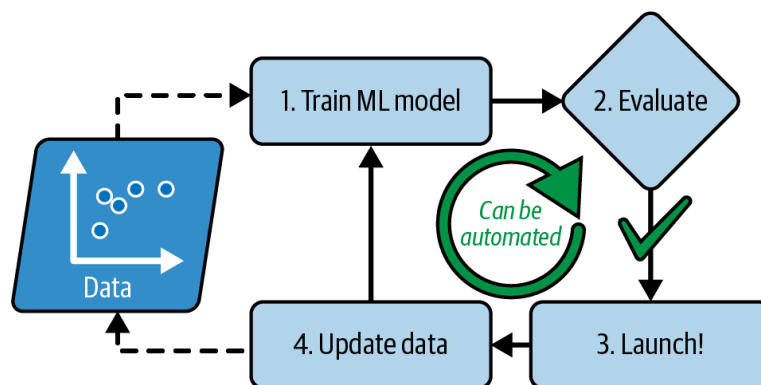


Figura 3. Adaptação automática às mudanças. Fonte: RUSSANO; AVELINO, 2020.

Outra área em que o aprendizado de máquina se destaca é a de problemas que são muito complexos para abordagens tradicionais ou que não têm algoritmo conhecido. Por exemplo, considere o reconhecimento de fala. Digamos que você queira começar de forma simples e escrever um programa capaz de distinguir as palavras "um" e "dois". Talvez você perceba que a palavra

"dois" começa com um som agudo ("T"), portanto, você poderia codificar um algoritmo que mede a intensidade do som agudo e usá-lo para distinguir um e dois - mas, obviamente, essa técnica não será dimensionada para milhares de palavras faladas por milhões de pessoas muito diferentes em ambientes ruidosos e em dezenas de idiomas. A melhor solução (pelo menos atualmente) é escrever um algoritmo que aprenda sozinho, com muitos exemplos de registros de cada palavra.

Por fim, o aprendizado de máquina pode ajudar os humanos a aprender conforme é ilustrado na Figura 4. Os modelos de AM podem ser inspecionados para ver o que aprenderam (embora para alguns modelos isso possa ser difícil). Por exemplo, uma vez que um filtro de spam tenha sido treinado com spam suficiente, ele pode ser facilmente inspecionado para revelar a lista de palavras e combinações de palavras que ele acredita serem os melhores preditores de spam. Às vezes, isso revelará correlações insuspeitadas ou novas tendências e, assim, levará a uma melhor compreensão do problema. A pesquisa em grandes quantidades de dados para descobrir padrões ocultos é chamada de datamining, e o aprendizado de máquina é excelente nisso.

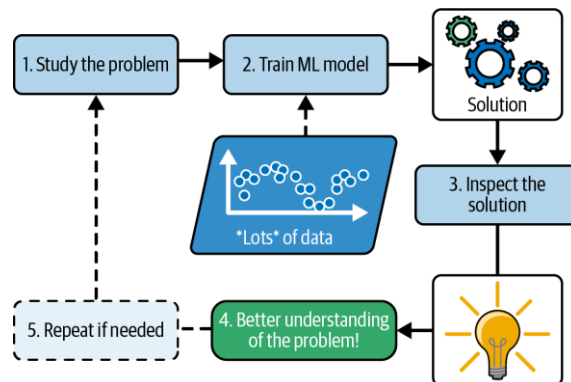


Figura 4. A aprendizagem automática pode ajudar os humanos a aprender. Fonte: RUSSANO; AVELINO, 2020.

Em resumo, o aprendizado de máquina é ótimo para:

- Problemas para os quais as soluções existentes exigem muito ajuste fino ou longas listas de regras.
- Problemas complexos para os quais o uso de uma abordagem tradicional não produz uma boa solução.
- Ambientes flutuantes.

- Obter insights sobre problemas complexos e grandes quantidades de dados.

1.3 Tipos de sistemas de aprendizado de máquina

Há tantos tipos diferentes de sistemas de aprendizado de máquina que é útil classificá-los em categorias amplas, com base nos seguintes critérios:

- Como eles são supervisionados durante o treinamento (supervisionado, não supervisionado, semissupervisionado, autossupervisionado e outros)
- Se podem ou não aprender de forma incremental em tempo real (on-line versus aprendizado em lote)
- Se eles funcionam simplesmente comparando novos pontos de dados com pontos de dados conhecidos, ou, em vez disso, detectando padrões nos dados de treinamento e criando um modelo preditivo, como fazem os cientistas (aprendizado baseado em instância versus aprendizado baseado em modelo)

Esses critérios não são exclusivos; você pode combiná-los da maneira que desejar. Por exemplo, um filtro de spam de última geração pode aprender na hora usando um modelo de rede neural profunda treinado com exemplos de spam e ham fornecidos por humanos; isso o torna um sistema de aprendizado supervisionado on-line, baseado em modelos.

1.3.1 Aprendizagem supervisionada

No aprendizado supervisionado, o conjunto de treinamento que você fornece ao algoritmo inclui as soluções desejadas, chamadas de rótulos, mostrados na Figura 5.

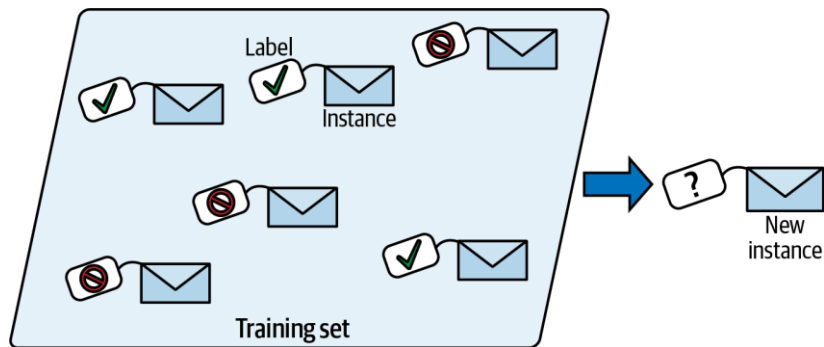


Figura 5. Um conjunto de treinamento rotulado para classificação de spam. Fonte: RUSSANO; AVELINO, 2020.

Uma tarefa típica de aprendizado supervisionado é a classificação. O filtro de spam é um bom exemplo disso: ele é treinado com muitos e-mails de exemplo, juntamente com sua classe (spam ou ham), e precisa aprender a classificar novos e-mails.

Outra tarefa típica é prever um valor numérico alvo, como o preço de um carro, com base em um conjunto de características (quilometragem, idade, marca etc.). Esse tipo de tarefa é chamado de regressão, representada graficamente **Erro! Fonte de referência não encontrada..** Para treinar o sistema, você precisa dar a ele muitos exemplos de carros, incluindo seus recursos e suas metas (ou seja, seus preços).

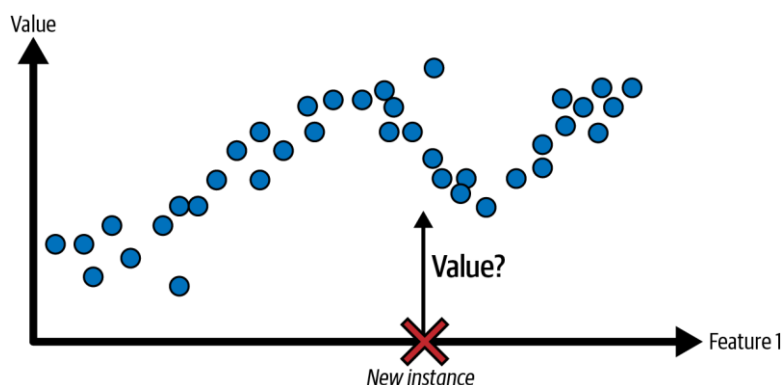


Figura 6. Um problema de regressão: prever um valor, dado um recurso de entrada. Fonte: RUSSANO; AVELINO, 2020.

Observe que alguns modelos de regressão também podem ser usados para classificação e vice-versa. Por exemplo, a regressão logística é comumente usada para classificação, pois pode gerar um valor que corresponde à

probabilidade de pertencer a uma determinada classe (por exemplo, 20% de chance de ser spam).

1.3.2 Aprendizagem não supervisionada

No aprendizado não supervisionado, como você pode imaginar, os dados de treinamento não são rotulados, representados visualmente na Figura 7). O sistema tende a aprender sem um professor.

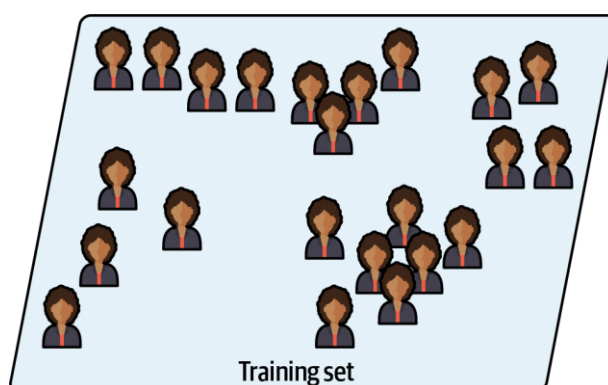


Figura 7. Um conjunto de treinamento sem rótulos para aprendizado não supervisionado.
Fonte: RUSSANO; AVELINO, 2020.

Por exemplo, digamos que você tenha muitos dados sobre os visitantes do seu blog. Você pode querer executar um algoritmo de agrupamento para tentar detectar grupos de visitantes semelhantes, destacados na Figura 8.

Em nenhum momento você informa ao algoritmo a que grupo um visitante pertence: ele encontra essas conexões sem a sua ajuda. Por exemplo, ele pode perceber que 40% dos seus visitantes são adolescentes que adoram histórias em quadrinhos e geralmente leem seu blog depois da escola, enquanto 20% são adultos que gostam de ficção científica e visitam o site nos fins de semana. Se você usar um algoritmo de agrupamento hierárquico, ele também poderá subdividir cada grupo em grupos menores. Isso pode ajudá-lo a direcionar suas postagens para cada grupo.

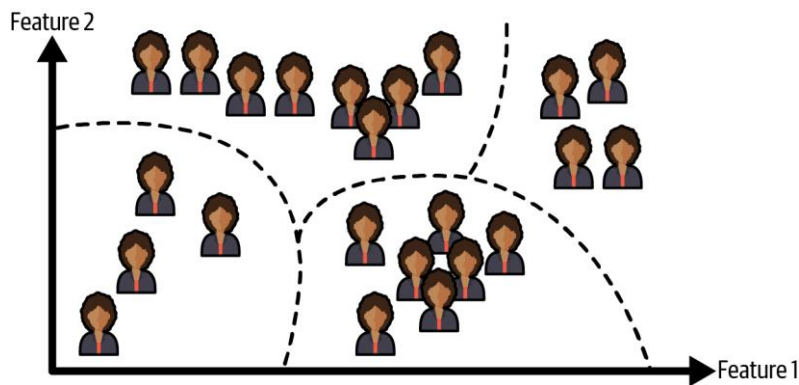


Figura 8. Agrupamento. Fonte: RUSSANO; AVELINO, 2020.

Os algoritmos de visualização também são bons exemplos de aprendizado não supervisionado: você os alimenta com muitos dados complexos e não rotulados, e eles produzem uma representação 2D ou 3D dos dados que podem ser facilmente plotada, ambos representados graficamente Figura 9. Esses algoritmos tentam preservar o máximo de estrutura possível (por exemplo, tentando evitar que clusters separados no espaço de entrada se sobreponham na visualização) para que você possa entender como os dados estão organizados e talvez identificar padrões insuspeitados.

Uma tarefa relacionada é a redução da dimensionalidade, na qual o objetivo é simplificar os dados sem perder muitas informações. Uma maneira de fazer isso é mesclar vários recursos correlacionados em um só. Por exemplo, a quilometragem de um carro pode estar fortemente correlacionada com sua idade, de modo que o algoritmo de redução de dimensionalidade os mesclará em um único recurso que represente o desgaste do carro. Isso é chamado de extração de recursos.

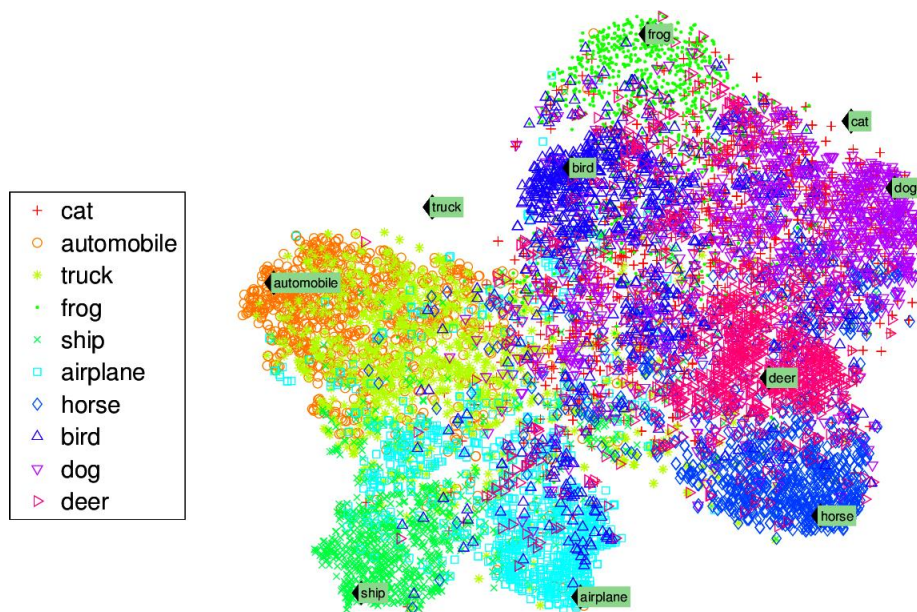


Figura 9. Exemplo de uma visualização t-SNE destacando grupos semânticos. Fonte: RUSSANO; AVELINO, 2020.

Outra importante tarefa não supervisionada é a detecção de anomalias, por exemplo, detectar transações incomuns com cartão de crédito para evitar fraudes, detectar defeitos de fabricação ou remover automaticamente as exceções de um conjunto de dados antes de alimentá-lo com outro algoritmo de aprendizado.

Durante o treinamento, são mostradas ao sistema, em sua maioria, instâncias normais, para que ele aprenda a reconhecê-las; então, quando vê uma nova instância, ele pode dizer se ela se parece com uma instância normal ou se provavelmente é uma anomalia, veja a Figura 10. Uma tarefa muito semelhante é a detecção de novidades: ela visa detectar novas instâncias que parecem diferentes de todas as instâncias do conjunto de treinamento. Para isso, é necessário ter um conjunto de treinamento muito "limpo", sem nenhuma instância, o qual você gostaria que o algoritmo detectasse. Por exemplo, se você tiver milhares de fotos de cães e 1% dessas fotos representar Chihuahuas, um algoritmo de detecção de novidades não deve tratar as novas fotos de Chihuahuas como novidades. Por outro lado, os algoritmos de detecção de anomalias podem considerar esses cães tão raros e tão diferentes de outros cães que provavelmente os classificariam como anomalias (sem ofensa aos Chihuahuas).

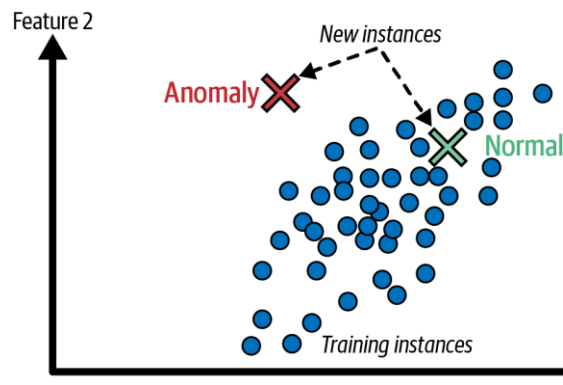


Figura 10. Detecção de anomalias. Fonte: RUSSANO; AVELINO, 2020.

Por fim, outra tarefa comum não supervisionada é o aprendizado de regras de associação, em que o objetivo é analisar grandes quantidades de dados e descobrir relações interessantes entre os atributos. Por exemplo, suponha que você seja proprietário de um supermercado. A execução de uma regra de associação em seus registros de vendas pode revelar que as pessoas que compram molho barbecue e batatas fritas também tendem a comprar bife. Portanto, talvez você queira colocar esses itens próximos uns dos outros.

1.3.3 Aprendizado por reforço

O aprendizado por reforço é muito diferente. O sistema de aprendizagem, chamado de agente nesse contexto, pode observar o ambiente, selecionar e executar ações e receber recompensas em troca (ou penalidades na forma de recompensas negativas, conforme mostrado na Figura 11. Em seguida, ele deve aprender por si mesmo qual é a melhor estratégia, chamada de política, para obter a maior recompensa ao longo do tempo. Uma política define a ação que o agente deve escolher quando estiver em uma determinada situação.

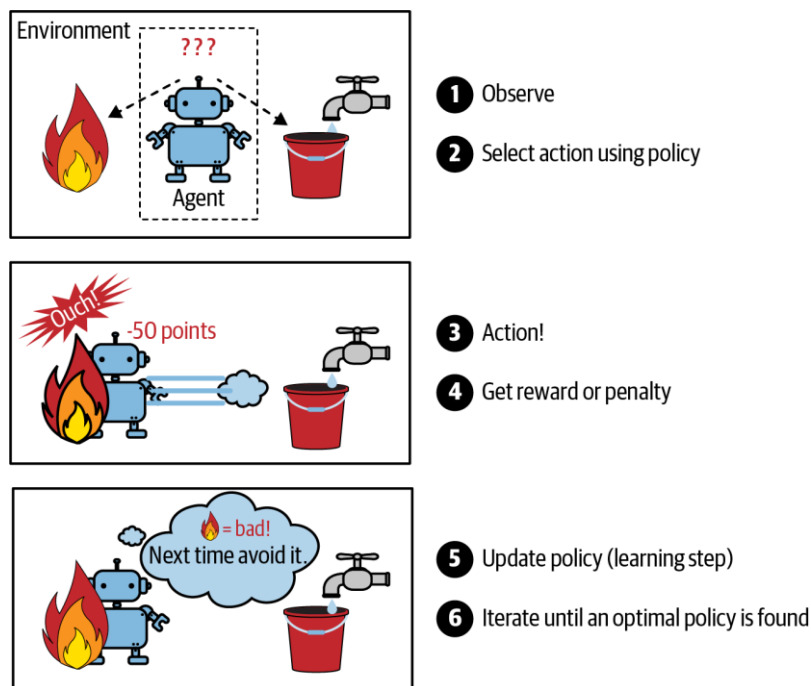


Figura 11. Aprendizagem por reforço. Fonte: RUSSANO; AVELINO, 2020.

Por exemplo, muitos robôs implementam algoritmos de aprendizagem por reforço para aprender a andar. O programa AlphaGo da DeepMind também é um bom exemplo de aprendizado por reforço: ele ganhou as manchetes em maio de 2017 quando derrotou Ke Jie, o jogador número um do ranking mundial na época, no jogo Go. Ele aprendeu sua política de vitória analisando milhões de jogos e, em seguida, jogando muitos jogos contra si mesmo. Observe que o aprendizado foi desativado durante os jogos contra o campeão; o AlphaGo estava apenas aplicando a política que havia aprendido. Como você verá na próxima seção, isso é chamado de aprendizagem off-line.

1.4 Aprendizado de máquina aplicado com python

1.4.1 Scikit-Learn

O Scikit-learn é uma biblioteca para a linguagem de programação Python que oferece suporte ao aprendizado de máquina. Ela oferece suporte a recursos de aprendizado de máquina, como classificação, regressão e agrupamento, juntamente com APIs para dar suporte a algoritmos populares, como:

1. KNN
2. Logistic Regression
3. Support Vector Machine
4. Linear Regression
5. Naive Bayes
6. Neural Network
7. Random Forest
8. LDA

Vamos apresentar a principal pipeline de **modelagem de aprendizado de máquina** com um algoritmo simples: o chamado **Nearest Neighbors** (Vizinhos mais próximos). Ilustraremos esse algoritmo com uma tarefa de classificação, em um vetor bidimensional de recursos, mas observe que ele também pode ser usado em tarefas clássicas de regressão. Ao longo desta seção, usaremos principalmente o scikitlearn.

O projeto scikit começou em 2011 (consulte Pedregosa et al. (2011) para obter mais referências) e atualmente é uma das principais plataformas de código aberto Python para aprendizado de máquina. Nos últimos anos, o Tensorflow, desenvolvido pelo Google em 2015 (consulte Abadi et al. (2015) para obter detalhes), ganhou uma popularidade notável, especialmente na comunidade de aprendizagem profunda, e hoje é amplamente usado para executar projetos e pipelines de ML.

O **K-Nearest Neighbors (KNN)** é um algoritmo simples e eficiente para tarefas de classificação e regressão no aprendizado de máquina. Ele se baseia na ideia de que pontos de dados semelhantes tendem a ter valores-alvo semelhantes. O algoritmo funciona encontrando os **K**, pontos de dados mais próximos de uma determinada entrada e usando a classe majoritária ou o valor médio dos pontos de dados mais próximos para fazer uma previsão.

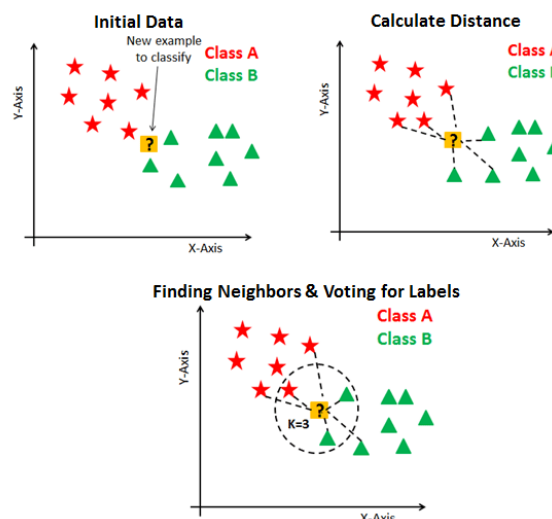


Figura 12. Processo de criação de um modelo KNN. Fonte: Medium.

O processo de criação de um modelo KNN começa com a seleção de um valor para **K**, que é o número de vizinhos mais próximos a serem considerados para a previsão. Em seguida, os dados são divididos em conjuntos de treinamento e de teste, com o conjunto de treinamento usado para encontrar os vizinhos mais próximos. Para fazer uma previsão para uma nova entrada, o algoritmo calcula a distância entre a entrada e cada ponto de dados no conjunto de treinamento e seleciona os **K** pontos de dados mais próximos. A classe majoritária ou o valor médio dos pontos de dados mais próximos é então usado como previsão.

Uma das principais vantagens do KNN é sua simplicidade e flexibilidade. Ele pode ser usado tanto para tarefas de classificação quanto de regressão e não faz nenhuma suposição sobre a distribuição de dados subjacente. Além disso, ele pode lidar com dados de alta dimensão e pode ser usado para aprendizado supervisionado e não supervisionado.

A principal desvantagem do KNN é sua complexidade computacional. À medida que o tamanho do conjunto de dados aumenta, o tempo e a memória necessários para encontrar os vizinhos mais próximos podem se tornar proibitivos. Além disso, o KNN pode ser sensível à escolha de **k**, e encontrar o valor ideal para **K** pode ser difícil.

A importação do modelo é feita da seguinte forma:

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

O **Logistic Regression** é um método estatístico usado para prever resultados binários, como sucesso ou fracasso, com base em uma ou mais variáveis independentes. É uma técnica popular em aprendizado de máquina e é frequentemente usada para tarefas de classificação, como determinar se um e-mail é spam ou não, ou prever se um cliente vai se desligar.

O modelo de regressão logística é baseado na função logística, que é uma função sigmoide que mapeia as variáveis de entrada para uma probabilidade entre 0 e 1. A probabilidade é então usada para fazer uma previsão sobre o resultado.

O modelo de regressão logística é representado pela seguinte equação:

$$P(y=1|x) = 1/(1+e^{-(b_0 + b_1x_1 + b_2x_2 + \dots + b_n * x_n)})$$

Em que $P(y=1|x)$ é a probabilidade de que o resultado y seja 1, dadas as variáveis de entrada x , b_0 é o intercepto e b_1, b_2, \dots, b_n são os coeficientes das variáveis de entrada x_1, x_2, \dots, x_n .

Os coeficientes são determinados pelo treinamento do modelo em um conjunto de dados e pelo uso de um algoritmo de otimização, como a descida de gradiente, para minimizar a função de custo, que normalmente é a perda de log.

Depois que o modelo é treinado, ele pode ser usado para fazer previsões inserindo novos dados e calculando a probabilidade de o resultado ser 1. O limite para classificar o resultado como 1 ou 0 é normalmente definido como 0,5, mas isso pode ser ajustado dependendo da tarefa específica e da compensação desejada entre falsos positivos e falsos negativos.

Na Figura 13, está um diagrama que representa o modelo de regressão logística:

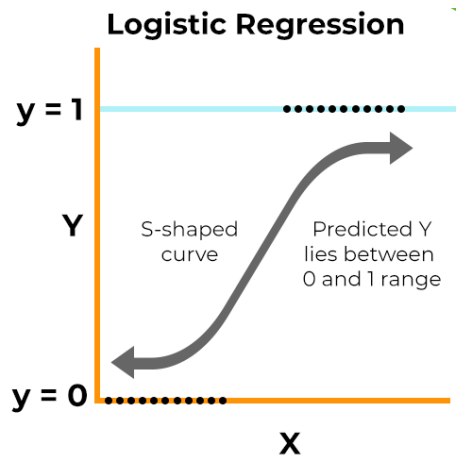


Figura 13. Diagrama do modelo Logistic Regression. Fonte: Medium.

Nesse diagrama, as variáveis de entrada x_1 e x_2 são usadas para prever o resultado binário y . A função logística mapeia as variáveis de entrada para uma probabilidade, que é então usada para fazer uma previsão sobre o resultado. Os coeficientes b_1 e b_2 são determinados pelo treinamento do modelo em um conjunto de dados e o limite é definido como 0,5.

A importação do modelo é feita da seguinte forma:

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

Support Vector Machines (SVMs) são um tipo de algoritmo de aprendizado supervisionado que pode ser usado para problemas de classificação ou regressão. A principal ideia por trás das SVMs é encontrar o limite que separa diferentes classes nos dados, maximizando a margem, que é a distância entre o limite e os pontos de dados mais próximos de cada classe. Esses pontos de dados mais próximos são chamados de vetores de suporte, conforme ilustrado na Figura 14.

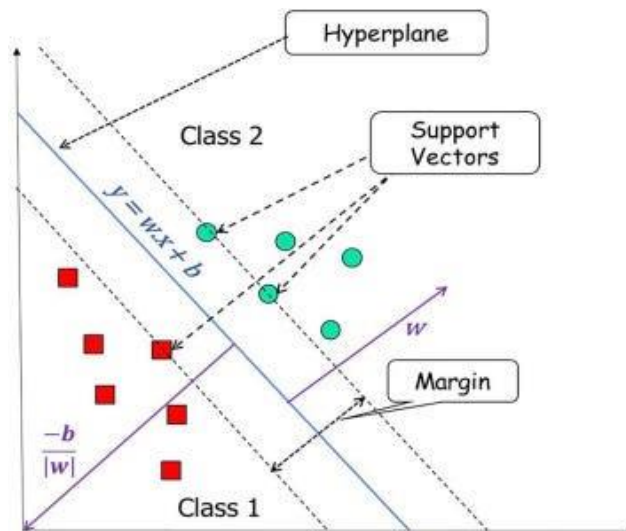


Figura 14. Diagrama do modelo Support Vector Machines. Fonte: Medium.

As SVMs são particularmente úteis quando os dados não são linearmente separáveis, o que significa que não podem ser separados por uma linha reta. Nesses casos, as SVMs podem transformar os dados em um espaço dimensional mais alto usando uma técnica chamada truque de kernel, em que um limite não linear pode ser encontrado. Algumas funções de kernel comuns usadas em SVMs são polinomiais, função de base radial (RBF) e sigmoide.

Uma das principais vantagens dos SVMs é que eles são muito eficazes em espaços de alta dimensão e têm um bom desempenho mesmo quando o número de recursos é maior que o número de amostras. Além disso, os SVMs são eficientes em termos de memória porque só precisam armazenar os vetores de suporte e não todo o conjunto de dados.

Por outro lado, as SVMs podem ser sensíveis à escolha da função de kernel e aos parâmetros do algoritmo. Também é importante observar que as SVMs não são adequadas para grandes conjuntos de dados, pois o tempo de treinamento pode ser bastante longo.

A importação do modelo é feita da seguinte forma:

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

1.5 Exercício prático

Desenvolva um programa em python no **VS Code** para realizar predição de espécie de flor iris com base no comprimento das pétalas e sépalas do **Iris Dataset** aplicando o modelo **KNN**.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

1.6 Exercício Prático

Desenvolva um programa em python no **VS Code** para realizar predição de espécie de flor iris com base no comprimento das pétalas e sépalas do **Iris Dataset** aplicando o modelo **Logistic Regression**.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

1.7 Exercício Prático

Desenvolva um programa em python no **VS Code** para realizar a classificação de dígitos escritos à mão com base no **Digits dataset** aplicando o modelo **SVM**.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

2. Visão computacional

O conceito central de qualquer sistema de IA é que ele pode perceber seu ambiente e realizar ações com base em suas percepções. A visão computacional se preocupa com a parte da percepção visual: é a ciência de perceber e compreender o mundo por meio de imagens e vídeos, construindo um modelo físico do mundo para que um sistema de IA possa tomar as ações apropriadas. Para os seres humanos, a visão é apenas um aspecto da percepção. Percebemos o mundo por meio da visão, mas também por meio do som, do olfato e de nossos outros sentidos. Dependendo do aplicativo que estiver criando, você seleciona o dispositivo de detecção que melhor capta o mundo.

2.1. O que é percepção visual?

A percepção visual, em sua forma mais básica, é o ato de observar padrões e objetos por meio da visão ou da entrada visual. Com um veículo autônomo, por exemplo, a percepção visual significa entender os objetos ao redor e seus detalhes específicos - como pedestres ou se há uma faixa específica na qual o veículo precisa estar centrado - e detecta sinais de trânsito entendendo o que eles significam. É por isso que a palavra percepção faz parte da definição. Não estamos apenas buscando capturar o ambiente ao redor, estamos tentando criar sistemas que possam realmente entender esse ambiente por meio de informações visuais.

2.2. Sistemas de visão

Nas décadas passadas, as técnicas tradicionais de processamento de imagens eram consideradas sistemas CV, mas isso não é totalmente exato. Uma máquina que processa uma imagem é completamente diferente dessa máquina que entende o que está acontecendo dentro da imagem, o que não é uma tarefa trivial. O processamento de imagens agora é apenas uma parte de um sistema maior e mais complexo que tem como objetivo interpretar o conteúdo da imagem.

2.2.1. Sistemas de Visão Humana

No nível mais alto, os sistemas de visão são praticamente os mesmos para seres humanos, animais, insetos e a maioria dos organismos vivos. Eles consistem em um sensor ou um olho para capturar a imagem e um cérebro para processar e interpretar a imagem. O sistema produz uma previsão dos componentes da imagem com base nos dados extraídos da imagem, mostrada na Figura 15.

Vamos ver como o sistema de visão humana funciona. Suponhamos que queiramos interpretar a imagem de cães da Figura 15. Olhamos para ela e entendemos diretamente que a imagem consiste em um grupo de cães (três, para ser mais específico). É bastante natural para nós classificar e detectar objetos nessa imagem porque fomos treinados ao longo dos anos para identificar cães.

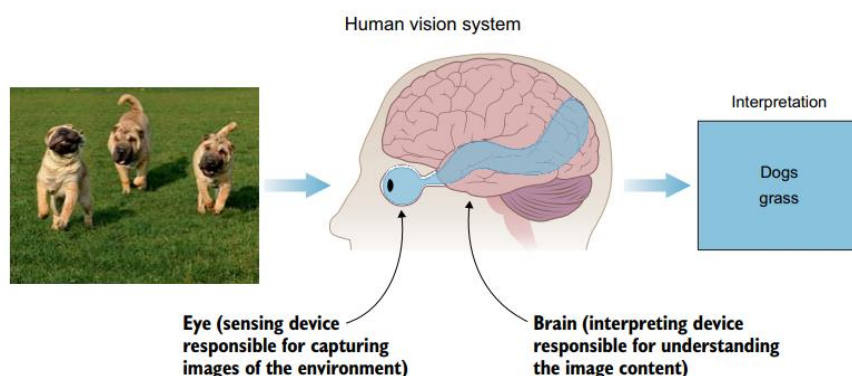


Figura 15. O sistema de visão humana usa os olhos e o cérebro para sentir e interpretar uma imagem. Fonte: ELGENDY, 2020.

Suponha que alguém lhe mostre a foto de um cachorro pela primeira vez - você definitivamente não sabe o que é. Em seguida, a pessoa lhe diz que se trata de um cão. Após alguns experimentos como esse, você terá sido treinado para identificar cães. Agora, em um exercício posterior, eles lhe mostrarão a foto de um cavalo. Quando você olha para a imagem, seu cérebro começa a analisar as características do objeto: hmmm, ele tem quatro patas, rosto comprido, orelhas compridas. Poderia ser um cachorro? "Errado: é um cavalo", dizem a você. Em seguida, seu cérebro ajusta alguns parâmetros em seu algoritmo para aprender as diferenças entre cães e cavalos. Parabéns! Você acabou de treinar

seu cérebro para classificar cães e cavalos. Você pode adicionar mais animais à equação, como gatos, tigres, guepardos e assim por diante? Sem dúvida. Você pode treinar seu cérebro para identificar quase tudo. O mesmo acontece com os computadores. Você pode treinar máquinas para aprender e identificar objetos, mas os seres humanos são muito mais intuitivos do que as máquinas. São necessárias apenas algumas imagens para que você aprenda a identificar a maioria dos objetos, ao passo que, com as máquinas, são necessários milhares ou, em casos mais complexos, milhões de amostras de imagens para aprender a identificar objetos.

2.2.2. Sistemas de Visão AI

Os cientistas se inspiraram no sistema de visão humana e, nos últimos anos, fizeram um trabalho incrível ao copiar a capacidade visual das máquinas. Para imitar o sistema de visão humana, precisamos dos mesmos dois componentes principais: um dispositivo de detecção para imitar a função do olho e um algoritmo poderoso para imitar a função do cérebro na interpretação e classificação do conteúdo da imagem, mostrada na Figura 16.

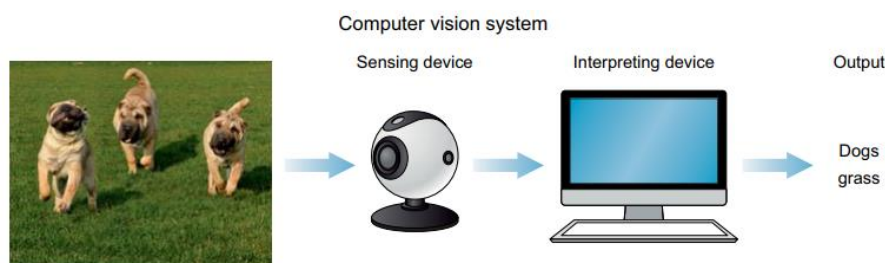


Figura 16. Os componentes do sistema de visão computacional são um dispositivo de detecção e um dispositivo de interpretação. Fonte: ELGENDY, 2020.

2.3. Dispositivos de detecção

Os sistemas de visão são projetados para cumprir uma tarefa específica. Um aspecto importante do design é a seleção do melhor dispositivo de detecção para capturar os arredores de um ambiente específico, seja uma câmera, um radar, um raio X, uma tomografia computadorizada, um Lidar ou uma

combinação de dispositivos para fornecer a cena completa de um ambiente para cumprir a tarefa em questão.

Vejamos novamente o exemplo do veículo autônomo (AV). O principal objetivo do sistema de visão do AV é permitir que o carro compreenda o ambiente ao seu redor e se mova do ponto A ao ponto B com segurança e em tempo hábil. Para cumprir essa meta, os veículos são equipados com uma combinação de câmeras e sensores que podem detectar 360 graus de movimento - pedestres, ciclistas, veículos, obras na estrada e outros objetos - a uma distância de até três campos de futebol.

Aqui estão alguns dos dispositivos de detecção normalmente usados em carros autônomos para perceber a área ao redor:

- Lidar, uma técnica semelhante a um radar, usa pulsos invisíveis de luz para criar um mapa 3D de alta resolução da área ao redor.
- As câmeras podem ver sinais de trânsito e marcações na estrada, mas não podem medir a distância.
- O radar pode medir a distância e a velocidade, mas não pode ver em detalhes finos.

Os aplicativos de diagnóstico médico usam raios X ou tomografias computadorizadas como dispositivos de detecção. Ou talvez seja necessário usar algum outro tipo de radar para capturar a paisagem para sistemas de visão agrícola. Há uma variedade de sistemas de visão, cada um projetado para executar uma tarefa específica. A primeira etapa no projeto de sistemas de visão é identificar a tarefa para a qual eles foram criados. Isso é algo que se deve ter em mente ao projetar sistemas de visão de ponta a ponta.

2.4. Dispositivos de interpretação

Os algoritmos de visão computacional são normalmente empregados como dispositivos de interpretação. O interpretador é o cérebro do sistema de visão. Sua função é pegar a imagem de saída do dispositivo de detecção e aprender recursos e padrões para identificar objetos. Portanto, precisamos construir um cérebro.

Os cientistas se inspiraram na forma como nossos cérebros funcionam e tentam fazer engenharia reversa no sistema nervoso central para obter alguma ideia de como construir um cérebro artificial. Assim, nasceram as redes neurais artificiais (RNNs), visualizada na Figura 17.

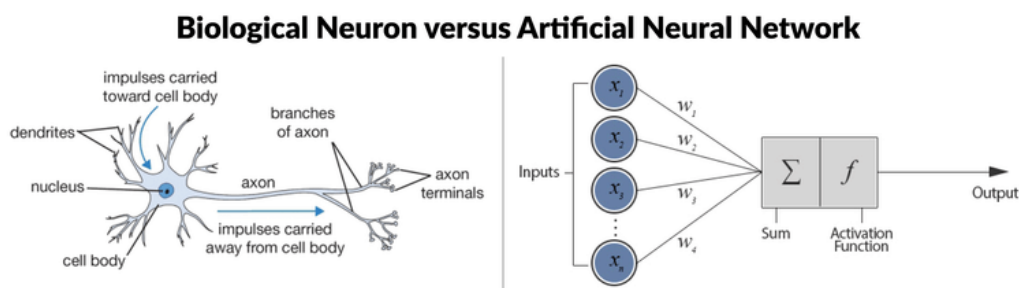


Figura 17. As semelhanças entre os neurônios biológicos e os sistemas artificiais. Fonte: ROUT; DWIVEDI; SRINIVASAN, 2019.

Na Figura 17, podemos ver uma analogia entre os neurônios biológicos e os sistemas artificiais. Ambos contêm um elemento de processamento principal, um neurônio, com sinais de entrada (x_1, x_2, \dots, x_n) e uma saída.

O comportamento de aprendizagem dos neurônios biológicos inspirou os cientistas a criar uma rede de neurônios conectados entre si. Imitando a forma como as informações são processadas no cérebro humano, onde cada neurônio artificial dispara um sinal para todos os neurônios aos quais está conectado, quando um número suficiente de seus sinais de entrada é ativado. Assim, os neurônios têm um mecanismo muito simples em nível individual, mas quando há milhões desses neurônios empilhados em camadas e conectados entre si, cada neurônio está conectado a milhares de outros neurônios, produzindo um comportamento de aprendizagem. A criação de uma rede neural multicamada é chamada de aprendizagem profunda, conforme pode ser visualizada na Figura 18.

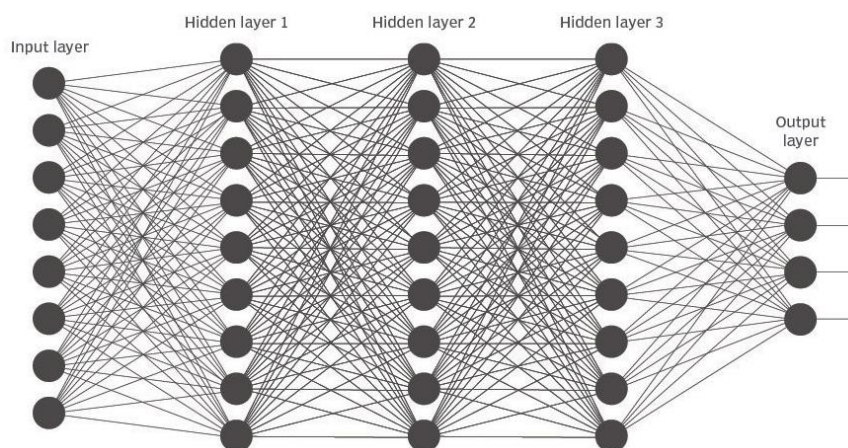


Figura 18. A aprendizagem profunda envolve camadas de neurônios em uma rede. Fonte: YASAR, 2023.

Os métodos de Aprendizagem Profunda (DL) aprendem representações por meio de uma sequência de transformações de dados em camadas de neurônios. Neste livro, exploraremos diferentes arquiteturas de DL, como Redes Neurais Artificiais (RNN) e Redes Neurais Convolucionais (CNNs), e como elas são usadas em aplicações de Visão Computacional (CV).

2.5. Aplicações de Visão Computacional

Os computadores começaram a ser capazes de reconhecer rostos humanos em imagens há décadas, mas agora os sistemas de AI estão rivalizando com a capacidade dos computadores de classificar objetos em fotos e vídeos. Graças à evolução drástica do poder computacional e da quantidade de dados disponíveis, a AI e a DL conseguiram alcançar um desempenho sobre-humano em muitas tarefas complexas de percepção visual, como pesquisa e legenda de imagens, classificação de imagens e vídeos e detecção de objetos. Além disso, as redes neurais profundas não se restringem a tarefas de CV: elas também são bem-sucedidas no processamento de linguagem natural e em tarefas de interface de usuário de voz.

A DL é usada em muitas aplicações de visão computacional para reconhecer objetos e seu comportamento. Nesta seção, não vou tentar listar todas as aplicações de visão computacional que existem. Eu precisaria de um livro inteiro para isso. Em vez disso, darei a você uma visão geral de alguns dos

algoritmos de DL mais populares e suas possíveis aplicações em diferentes setores. Entre esses setores estão carros autônomos, drones, robôs, câmeras em lojas e scanners de diagnóstico médico que podem detectar câncer de pulmão em estágios iniciais.

2.5.1. Classificação de imagens

É a tarefa de atribuir a uma imagem um rótulo de um conjunto predefinido de categorias. Uma rede neural convolucional é um tipo de rede neural que realmente se destaca no processamento e na classificação de imagens em muitas aplicações diferentes:

- Diagnóstico de câncer de pulmão - O câncer de pulmão é um problema crescente. A principal razão pela qual o câncer de pulmão é muito perigoso é que, quando é diagnosticado, geralmente está nos estágios intermediário ou final. Ao diagnosticar o câncer de pulmão, os médicos normalmente usam os olhos para examinar imagens de tomografia computadorizada, procurando pequenos nódulos nos pulmões. Nos estágios iniciais, os nódulos geralmente são muito pequenos e difíceis de detectar. Várias empresas de CV decidiram enfrentar esse desafio usando a tecnologia DL. Quase todo câncer de pulmão começa como um pequeno nódulo, e esses nódulos aparecem em uma variedade de formas que os médicos levam anos para aprender a reconhecer. Os médicos são muito bons em identificar nódulos de tamanho médio e grande, como os de 6 a 10 mm. Mas quando os nódulos são de 4 mm ou menores, às vezes os médicos têm dificuldade em identificá-los. As redes de DL, especificamente as CNNs, agora são capazes de aprender esses recursos automaticamente a partir de imagens de raios X e tomografia computadorizada e detectar pequenos nódulos precocemente, antes que se tornem mortais, mostrada na Figura 19.

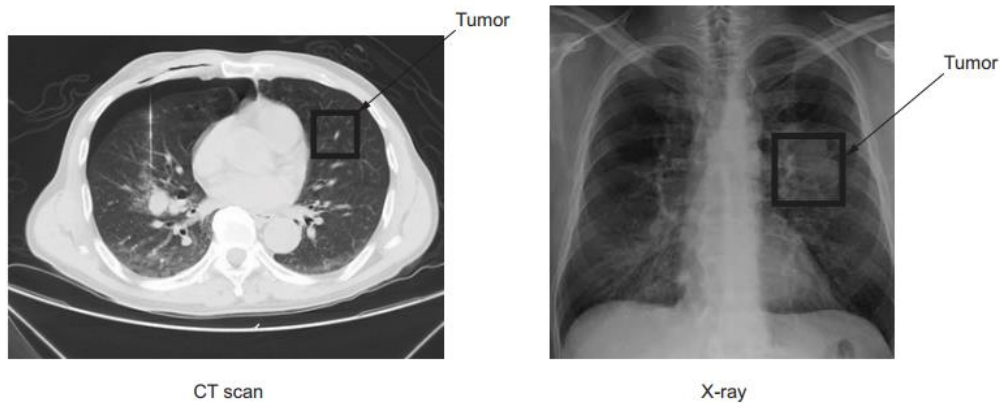


Figura 19. Detecção de tumores em estágios iniciais por AI. Fonte: ELGENDY, 2020.

- Reconhecimento de sinais de trânsito - Tradicionalmente, os métodos VC padrão eram empregados para detectar e classificar sinais de trânsito, mas essa abordagem exigia um trabalho manual demorado para criar manualmente recursos importantes nas imagens. Em vez disso, ao aplicar a AP a esse problema, podemos criar um modelo que classifique de forma confiável os sinais de trânsito, aprendendo a identificar os recursos mais adequados para esse problema por si só, conforme mostrada na Figura 20.



Figura 20. Os sistemas de visão podem detectar sinais de trânsito com um desempenho muito alto. Fonte: ELGENDY, 2020.

2.5.2. Detecção e localização de objetos

Os problemas de classificação de imagens são as aplicações mais básicas das CNNs. Nesses problemas, cada imagem contém apenas um objeto, e nossa tarefa é identificá-lo. Mas se quisermos atingir níveis humanos de compreensão, teremos de adicionar complexidade a essas redes para que elas possam reconhecer vários objetos e suas localizações em uma imagem. Para isso, podemos criar sistemas de detecção de objetos como YOLO (você só olha uma vez), SSD (detector de disparo único) e Faster R-CNN, que não só classificam imagens, mas também podem localizar e detectar cada objeto em imagens que contêm vários objetos. Esses sistemas de DL podem examinar uma imagem, dividi-la em regiões menores e rotular cada região com uma classe, de modo que um número variável de objetos em uma determinada imagem possa ser localizado e rotulado, visualizada na Figura 21. É possível imaginar que essa tarefa seja um pré-requisito básico para aplicações como sistemas autônomos.

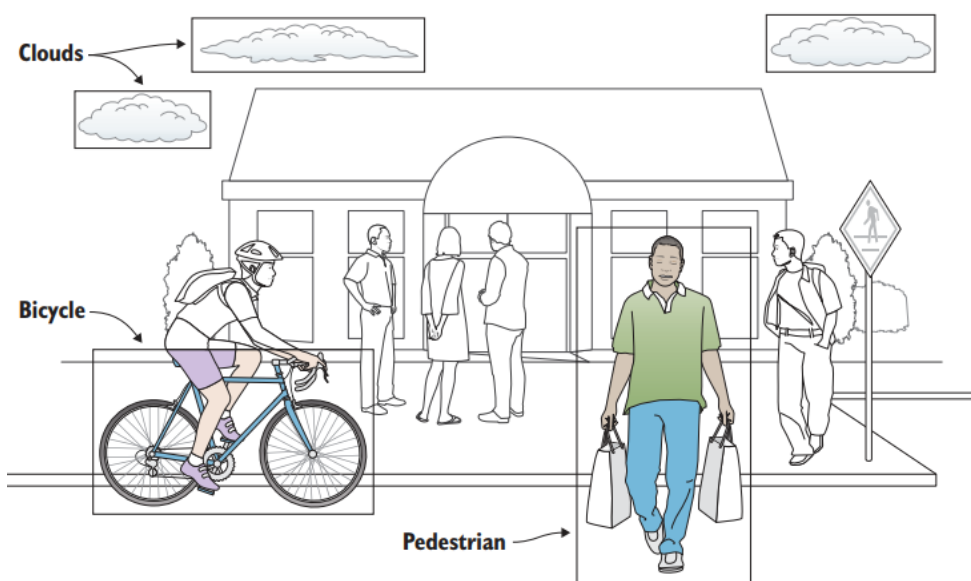


Figura 21. Os sistemas de aprendizagem profunda podem segmentar objetos em uma imagem.
Fonte: ELGENDY, 2020.

2.5.3. Geração de arte (transferência de estilo)

A transferência de estilo neural, um dos aplicativos de VC mais interessantes, é usada para transferir o estilo de uma imagem para outra. A ideia

básica da transferência de estilo é a seguinte: você pega uma imagem - digamos, de uma cidade - e aplica um estilo de arte a essa imagem - digamos, A Noite Estrelada (de Vincent Van Gogh) - e obtém a mesma cidade da imagem original, mas como se tivesse sido pintada por Van Gogh (Figura 22).

Na verdade, esse é um aplicativo interessante. O mais surpreendente, se você conhece algum pintor, é que pode levar dias ou até semanas para terminar uma pintura, mas aqui está um aplicativo que pode pintar uma nova imagem inspirada em um estilo existente em questão de segundos.



Figura 22. Transferência de estilo de A Noite Estrelada, de Van Gogh, para a imagem original.
Fonte: ELGENDY, 2020.

2.5.4. Criando imagens

Embora os exemplos anteriores sejam aplicações de AI de CV realmente impressionantes, é aqui que vejo a verdadeira magia acontecendo: a magia da criação. Em 2014, Ian Goodfellow inventou um novo modelo de AP que pode imaginar coisas novas, chamado de redes adversárias generativas (GANs). O nome faz com que elas pareçam um pouco intimidadoras, mas prometo a você que não são. Uma GAN é uma arquitetura CNN evoluída que é considerada um grande avanço em AP. Portanto, quando você entender as CNNs, as GANs farão muito mais sentido para você. Os GANs são modelos sofisticados de DL que geram imagens sintetizadas incrivelmente precisas de objetos, pessoas e lugares, entre outras coisas. Se você fornecer a eles um conjunto de imagens, eles poderão criar imagens totalmente novas e realistas. Por exemplo, o Stack GAN é uma das variações da arquitetura GAN que pode usar uma descrição textual de um objeto para gerar uma imagem de alta resolução do objeto que

corresponda a essa descrição. Essas "fotos" nunca foram vistas antes e são totalmente imaginárias, podemos observar na Figura 23.



Figura 23. As redes adversárias generativas (GANS) podem criar imagens novas e "inventadas" a partir de um conjunto de imagens existentes. Fonte: ELGENDY, 2020.

2.5.5. Reconhecimento facial

O reconhecimento facial (FR) nos permite identificar ou marcar com exatidão a imagem de uma pessoa. As aplicações cotidianas incluem a busca de celebridades na Web e a marcação automática de amigos e familiares em imagens. O reconhecimento facial é uma forma de classificação refinada.

Podemos categorizar dois modos de um sistema de FR:

- Identificação de face - A identificação de face envolve correspondências de um para muitos que comparam uma imagem de face de consulta com todas as imagens de modelo no banco de dados para determinar a identidade da face de consulta. Outro cenário de reconhecimento facial envolve uma verificação de lista de observação pelas autoridades municipais, em que um rosto de consulta é comparado a uma lista de suspeitos.
- Verificação de face - A verificação de face envolve uma correspondência de um para um que compara uma imagem de face

de consulta com uma imagem de face modelo cuja identidade está sendo reivindicada, mostrada na Figura 24.

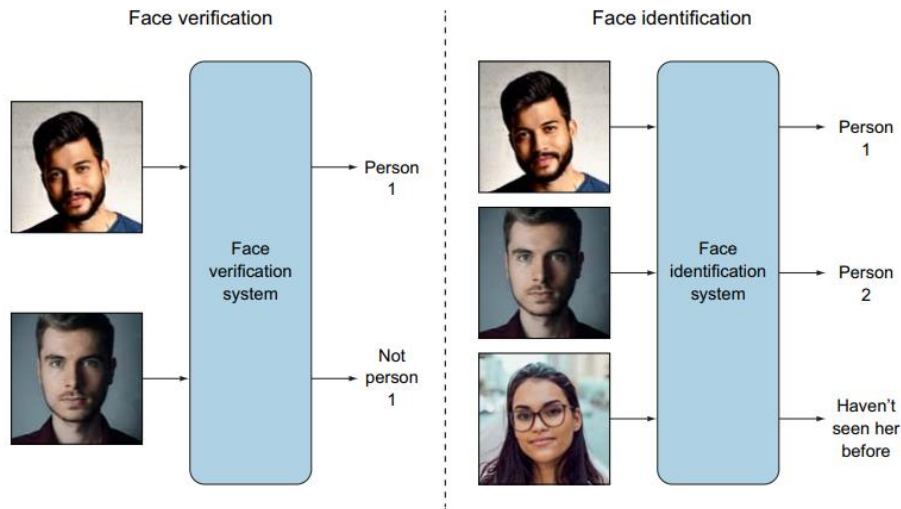


Figura 24. Verificação facial (esquerda) e Reconhecimento facial (direita). Fonte: ELGENDY, 2020.

2.6. Opencv - Biblioteca de Visão Computacional

É uma biblioteca de software de visão computacional e aprendizado de máquina de código aberto. O OpenCV foi criado para fornecer uma infraestrutura comum para aplicativos de visão computacional e para acelerar o uso da percepção de máquina nos produtos comerciais. Por ser um produto licenciado pelo Apache 2, o OpenCV facilita a utilização e a modificação do código pelas empresas.

O OpenCV tem uma estrutura modular, o que significa que o pacote inclui várias bibliotecas compartilhadas ou estáticas. Os seguintes módulos estão disponíveis:

- Funcionalidade principal (core).
- Processamento de imagens (imgproc).
- Análise de vídeo (video).
- Calibração de câmera e reconstrução 3D (calib3d).
- Estrutura de recursos 2D (features2d).
- GUI de alto nível (highgui).
- E/S de vídeo (videoio).

- Alguns outros módulos auxiliares, como FLANN e Google test wrappers, Python bindings e outros.

2.6.1. Primeiros passos com imagens

Para começar a usar as imagens do OpenCV, os usuários precisarão instalar o OpenCV em seus computadores. O OpenCV está disponível para várias plataformas, incluindo Windows, Linux e macOS. Há várias maneiras diferentes de instalar o OpenCV, mas a maneira mais fácil é usar um gerenciador de pacotes, como o apt-get ou o yum. Uma vez instalado o OpenCV, os usuários podem começar a usá-lo para ler e gravar imagens. Para ler uma imagem, os usuários podem usar a função **cv2.imread()**. Essa função usa o caminho para a imagem como entrada e retorna uma matriz NumPy contendo os dados da imagem.

Para gravar uma imagem, os usuários podem usar a função **cv2.imwrite()**. Essa função usa o caminho para a imagem e a matriz NumPy, que contém os dados da imagem como entrada. Para exibir uma imagem no OpenCV, os usuários podem usar a função **cv2.imshow()**. Essa função recebe como entrada o nome da janela em que a imagem será exibida e a matriz **NumPy** que contém os dados da imagem. O OpenCV fornece uma variedade de funções para realizar operações básicas de processamento de imagens, como redimensionamento, corte, conversão de espaços de cores e filtragem de imagens.

Por exemplo, para redimensionar uma imagem, os usuários podem usar a função **cv2.resize()**. Essa função recebe como entrada a matriz NumPy que contém os dados da imagem, a largura e a altura desejadas da imagem redimensionada e o método de interpolação a ser usado. Para cortar uma imagem, os usuários podem usar a classe **cv2.Rect()**.

Para converter uma imagem de um espaço de cor para outro, os usuários podem usar a função **cv2.cvtColor()**. Essa função usa como entrada a matriz NumPy que contém os dados da imagem e os espaços de cores de origem e destino. Para filtrar uma imagem, os usuários podem usar a função **cv2.filter2D()**.

Essa função usa a matriz NumPy que contém os dados da imagem, o kernel a ser usado para filtragem e a imagem de destino como entrada.

2.7. Exercício prático

Desenvolva um programa em python no **VS Code** para realizar a leitura, escrita e redimensionamento de uma imagem qualquer.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

2.8. Processamento de imagens

O processamento de imagens é o processo de manipulação de imagens digitais para melhorar sua qualidade ou extrair informações delas. Algumas tarefas comuns de processamento de imagens incluem:

- Redução de ruído: Remoção de ruídos indesejados das imagens, como granulação ou manchas.
- Aumento da nitidez da imagem: Fazer com que as imagens pareçam mais claras e distintas.
- Aprimoramento de contraste: Aumentar ou diminuir o contraste entre diferentes partes de uma imagem.
- Detecção de bordas: Identificar as bordas dos objetos em uma imagem.
- Conversão de espaço de cores: Conversão de imagens de um espaço de cores para outro, como de RGB para escala de cinza.

O OpenCV oferece uma variedade de funções para cada uma das tarefas de processamento de imagem listadas acima. Por exemplo, a função

cv2.GaussianBlur() pode ser usada para reduzir o ruído em uma imagem, e a função **cv2.Canny()** pode ser usada para detectar bordas em uma imagem.

A limiarização simples é um tipo de técnica de processamento de imagens usada para binarizar uma imagem. Isso significa que a imagem é convertida em uma imagem em preto e branco, em que cada pixel é preto ou branco.

Para executar a limiarização simples no OpenCV, você pode usar a função **cv2.threshold()**. A função **cv2.threshold()** recebe três argumentos: a imagem a ser limiarizada, o valor de limiar e o valor máximo. O valor de limiar é um valor de pixel usado para separar a imagem em duas regiões: os pixels maiores ou iguais ao valor de limiar são definidos como o valor máximo, e os pixels menores que o valor de limiar são definidos como zero.

A erosão e a dilatação são duas operações morfológicas básicas usadas no processamento de imagens. A erosão remove os pixels dos limites dos objetos, enquanto a dilatação adiciona pixels aos limites dos objetos.

A erosão é usada para reduzir o tamanho dos objetos em uma imagem. Também é usada para remover o ruído das imagens. Para executar a erosão no OpenCV, você pode usar a função **cv2.erode()**. A função **cv2.erode()** recebe dois argumentos: a imagem a ser erodida e o kernel. O kernel é uma pequena matriz que é usada para definir a forma da operação de erosão.

A dilatação é o oposto da erosão. Ela aumenta o tamanho dos objetos em uma imagem. Também é usada para preencher buracos em objetos. Para realizar a dilatação no OpenCV, você pode usar a função **cv2.dilate()**. A função **cv2.dilate()** recebe dois argumentos: a imagem a ser dilatada e o kernel. O kernel é uma pequena matriz usada para definir a forma da operação de dilatação.

2.9. Exercício Prático

Desenvolva um programa em python no **VS Code** para realizar o aprimoramento das imagens aplicando binarização, erosão ou dilatação.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os

arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

3. Aprendizagem Profunda e Redes Neurais

Nos últimos anos, a inteligência artificial (AI) tem sido objeto de intenso alarde na mídia. Aprendizado de máquina, aprendizado profundo e IA aparecem em inúmeros artigos, muitas vezes fora de publicações voltadas para a tecnologia. Prometem-nos um futuro de chatbots inteligentes, carros autônomos e assistentes virtuais - um futuro às vezes pintado de forma sombria e outras vezes como utópico, em que os empregos humanos serão escassos e a maior parte das atividades econômicas será realizada por robôs ou agentes de AI. Para um futuro ou atual praticante de aprendizado de máquina, é importante ser capaz de reconhecer o sinal em meio ao ruído, para que você possa distinguir desenvolvimentos que mudarão o mundo de comunicados de imprensa exagerados.

3.1. O que é aprendizagem profunda?

A aprendizagem profunda é um subcampo da aprendizagem de máquina que enfatiza a aprendizagem de camadas sucessivas de representações a partir de dados, sem necessariamente se relacionar com a neurobiologia. O termo "profundo" refere-se à ideia de várias camadas de representações. As redes neurais são usadas para aprender essas representações em camadas, empilhando-as umas sobre as outras. Ao contrário de abordagens de aprendizado de máquina mais superficiais, a aprendizagem profunda envolve dezenas ou centenas de camadas. Não há evidências de que a aprendizagem profunda esteja relacionada à biologia. Em vez disso, é uma estrutura matemática para aprender representações a partir de dados. As representações são aprendidas por algoritmos de aprendizagem profunda, e o texto menciona a transformação de imagens como exemplo de sua aplicação.

Como são as representações aprendidas por um algoritmo de aprendizagem profunda? Vamos examinar como uma rede com várias camadas de profundidade, mostrada na Figura 25, transforma uma imagem de um dígito para reconhecer qual é o dígito.

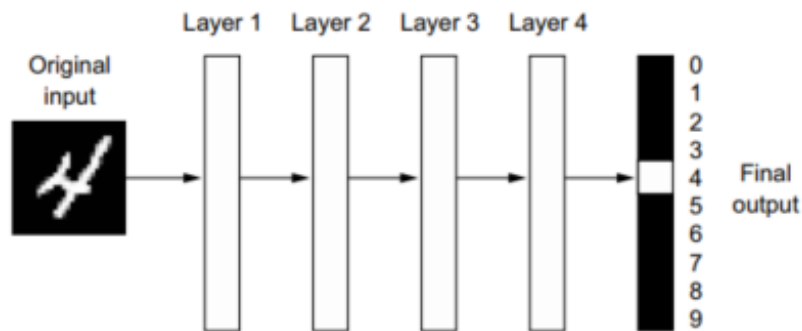


Figura 25. Uma rede neural profunda para classificação de dígitos. Fonte: CHOLLET, 2021.

Como você pode ver na Figura 26, a rede transforma a imagem do dígito em representações cada vez mais diferentes da imagem original e cada vez mais informativas sobre o resultado. Você pode pensar em uma rede profunda como um processo de destilação de informações de vários estágios, em que as informações passam por filtros sucessivos e saem cada vez mais purificadas (ou seja, úteis em relação a alguma tarefa).

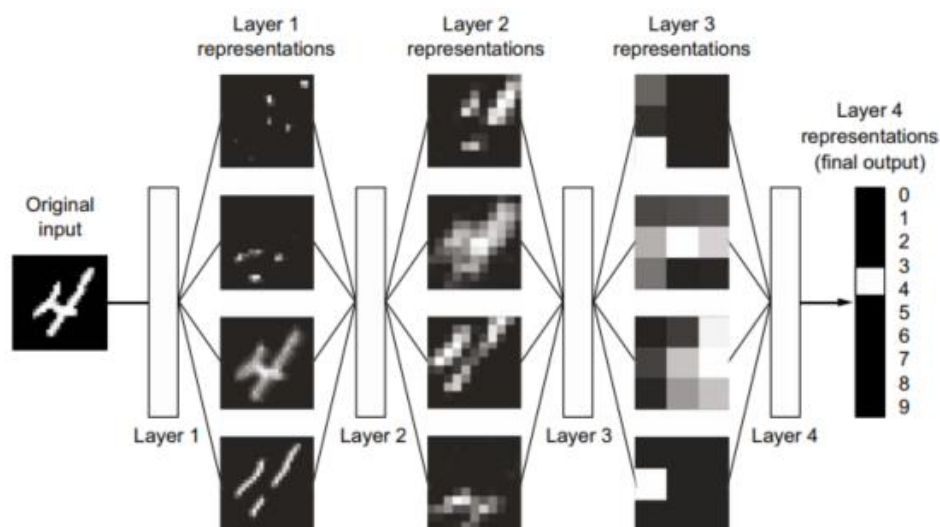


Figura 26. Representações de dados aprendidas por um modelo de classificação de dígitos. Fonte: CHOLLET, 2021.

Então, tecnicamente, é isso que a aprendizagem profunda é: uma maneira de aprender representações de dados em vários estágios. É uma ideia simples, mas, como se vê, mecanismos muito simples, em escala suficiente, podem acabar parecendo mágica.

3.1.1. Perceptron

A rede neural mais simples é o perceptron, que consiste em um único neurônio. Conceitualmente, o perceptron funciona de forma semelhante a um neurônio biológico, representada na Figura 27. Um neurônio biológico recebe sinais elétricos de seus dendritos, modula os sinais elétricos em várias quantidades e, em seguida, dispara um sinal de saída por meio de suas sinapses somente quando a intensidade total dos sinais de entrada ultrapassa um determinado limite. A saída é então enviada a outro neurônio, e assim por diante.

Para modelar o fenômeno do neurônio biológico, o neurônio artificial executa duas funções consecutivas: calcula a soma ponderada das entradas para representar a força total dos sinais de entrada e aplica uma função de etapa ao resultado para determinar se deve disparar a saída 1 se o sinal exceder um determinado limite ou 0 se o sinal não exceder o limite.

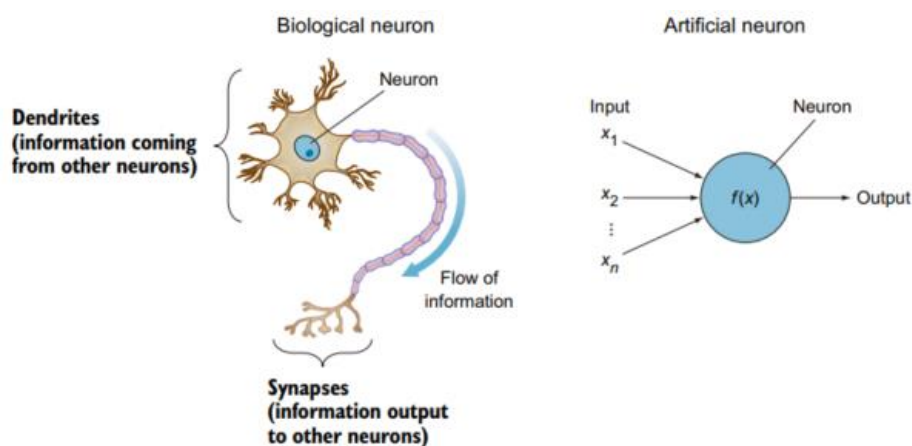


Figura 27. Os neurônios artificiais foram inspirados nos neurônios biológicos. Fonte: ELGENDY, 2020.

No diagrama do perceptron da Figura 28, você pode ver o seguinte:

- Vetor de entrada - O vetor de recursos que é alimentado ao neurônio. Geralmente é denotado com um X maiúsculo para representar um vetor de entradas (x_1, x_2, \dots, x_n) .
- Vetor de pesos - A cada x_1 é atribuído um valor de peso w_1 que representa sua importância para distinguir entre diferentes pontos de dados de entrada.

- Funções do neurônio - Os cálculos realizados no neurônio para modular os sinais de entrada: a soma ponderada e a função de ativação por etapas.
- Saída - Controlada pelo tipo de função de ativação que você escolher para a rede. Há diferentes funções de ativação, conforme discutiremos em detalhes nesta seção. Para uma função de etapa, a saída é 0 ou 1. Outras funções de ativação produzem saída de probabilidade ou números flutuantes. O nó de saída representa a previsão do perceptron.

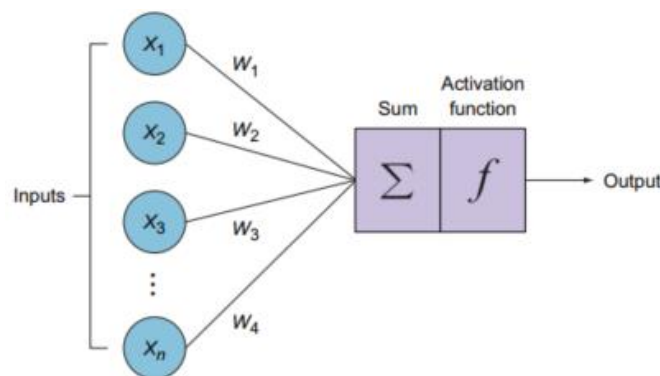


Figura 28. Os vetores de entrada são alimentados ao neurônio, com pesos atribuídos para representar a importância. Fonte: ELGENDY, 2020.

▪ Função de Soma Ponderada

Também conhecida como combinação linear, a função de soma ponderada é a soma de todos os inputs multiplicada por seus pesos e, em seguida, adicionada a um termo de polarização. Essa função produz uma linha reta representada na equação a seguir:

$$z = \sum x_i \cdot w_i + b(\text{bias})$$

$$z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \dots + x_n \cdot w_n + b$$

Veja como implementamos a soma ponderada em Python:

$$z = \text{np.dot}(w.T, X) + b$$

▪ Função de Ativação por Etapas

Tanto nas redes neurais artificiais quanto nas biológicas, um neurônio não produz apenas a entrada rara que recebe. Em vez disso, há mais uma etapa, chamada de função de ativação; essa é a unidade de tomada de decisão do cérebro. Nas RNAs, a função de ativação recebe a mesma entrada de soma ponderada de antes ($z = \sum x_i \cdot w_i + b$) e ativa (dispara) o neurônio, caso a soma ponderada seja maior que um determinado limite. Essa ativação ocorre com base nos cálculos da função de ativação. Mais adiante nesta seção, analisaremos os diferentes tipos de funções de ativação e sua finalidade geral no contexto mais amplo das redes neurais. A função de ativação mais simples usada pelo algoritmo perceptron é a função passo, que produz uma saída binária (0 ou 1). Basicamente, ela diz que se a entrada somada for ≥ 0 , ela "dispara" (saída = 1); caso contrário (entrada somada < 0), ela não dispara (saída = 0), destacada no gráfico da Figura 29.

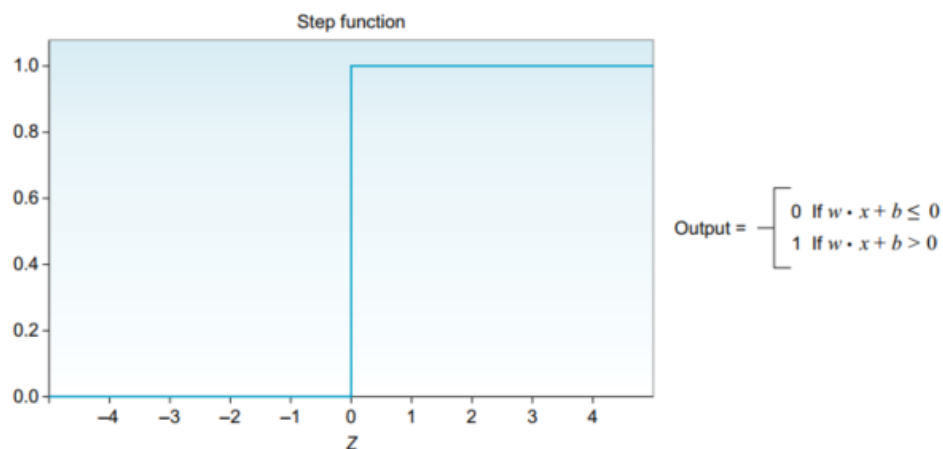


Figura 29. A função de etapa produz uma saída binária (0 ou 1). Fonte: ELGENDY, 2020.

$\hat{y} = g(x)$, onde g é uma função de ativação e z é a soma ponderada $z = \sum x_i \cdot w_i + b$.

Esta é a aparência da função de etapa em Python:

```
def step_function(z):  
    if z <= 0:  
        return 0
```

```
else:  
    return 1
```

3.1.2. Como o perceptron aprende?

O perceptron usa tentativa e erro para aprender com seus erros. Utiliza os pesos como botões, ajustando seus valores para cima e para baixo até que a rede seja treinada, visualizada na Figura 30.

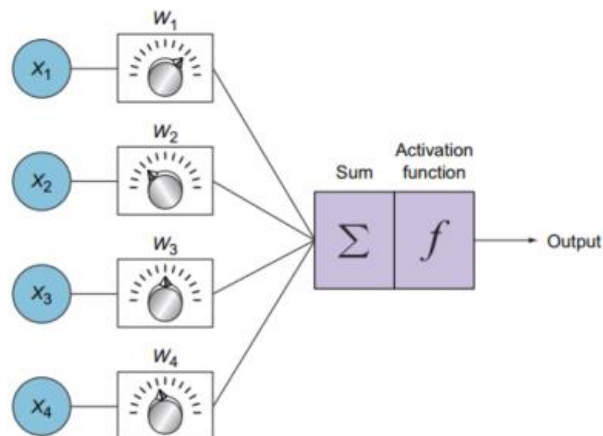


Figura 30. Pesos ajustados durante o processo de aprendizado. Fonte: ELGENDY, 2020.

A lógica de aprendizado do perceptron é a seguinte:

1. O neurônio calcula a soma ponderada e aplica a função de ativação para fazer uma previsão \hat{y} . Isso é chamado de processo de feedforward:

$$\hat{y} = \text{Ativação}(\sum x_i \cdot w_i + b)$$

2. Ele compara a previsão de saída com o rótulo correto para calcular o erro:

$$\text{error} = y - \hat{y}$$

3. Em seguida, ele atualiza o peso. Se a previsão for muito alta, ele ajusta o peso para fazer uma previsão mais baixa na próxima vez, e vice-versa.
4. Repita!

Esse processo é repetido várias vezes, e o neurônio continua a atualizar os pesos para melhorar suas previsões até que a etapa 2 produza um erro muito pequeno (próximo de zero), o que significa que a previsão do neurônio está muito próxima do valor correto. Nesse ponto, podemos interromper o treinamento e salvar os valores de peso que produziram os melhores resultados para aplicar em casos futuros em que o resultado é desconhecido.

3.1.3. Um neurônio é suficiente para resolver problemas complexos?

A resposta curta é não, mas vamos ver por quê. O perceptron é uma função linear. Isso significa que o neurônio treinado produzirá uma linha reta que separa nossos dados. Suponhamos que queiramos treinar um perceptron para prever se um jogador será aceito no time da faculdade. Coletamos todos os dados dos anos anteriores e treinamos o perceptron para prever se os jogadores serão aceitos com base em apenas dois recursos (altura e peso). O perceptron treinado encontrará os melhores pesos e valores de polarização para produzir a linha reta que melhor separa os aceitos dos não aceitos (melhor ajuste). A linha tem a seguinte equação:

$$z = \text{height} \cdot w_1 + \text{age} \cdot w_2 + b$$

Após a conclusão do treinamento com os dados de treinamento, podemos começar a usar o perceptronto para fazer previsões com novos jogadores. Quando temos um jogador com 150 cm de altura e 12 anos de idade, calculamos a equação anterior com os valores (150, 12). Quando plotada em um gráfico, da Figura 31, você pode ver que ela fica abaixo da linha: o neurônio está prevendo que esse jogador não será aceito. Se ficar acima da linha, então o jogador será aceito.

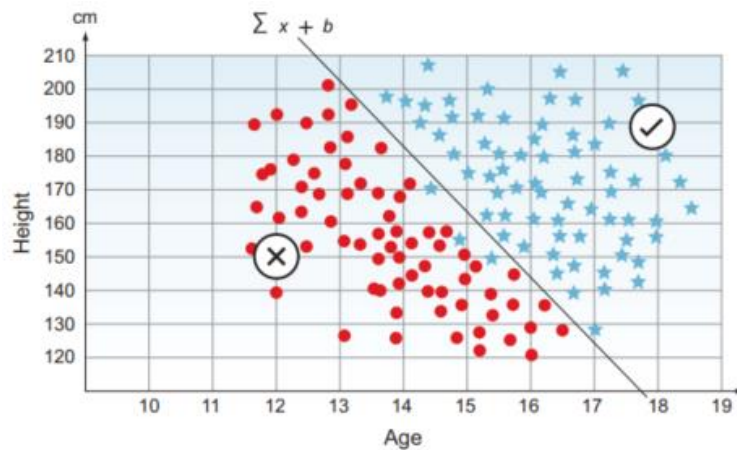


Figura 31. Os dados linearmente separáveis podem ser separados por uma linha reta. Fonte: ELGENDY, 2020.

Na Figura 31, o perceptron único funciona bem porque nossos dados eram linearmente separáveis, o que significa que os dados de treinamento podem ser separados por uma linha reta. Mas a vida nem sempre é tão simples. O que acontece quando temos um conjunto de dados mais complexo que não pode ser separado por uma linha reta (conjunto de dados não linear)?

Como você pode ver na Figura 32, uma única linha reta não separará nossos dados de treinamento. Dizemos que ela não se ajusta aos nossos dados. Precisamos de uma rede mais complexa para dados mais complexos como esse. E se construíssemos uma rede com dois perceptrons? Isso produziria duas linhas. Isso nos ajudaria a separar melhor os dados?

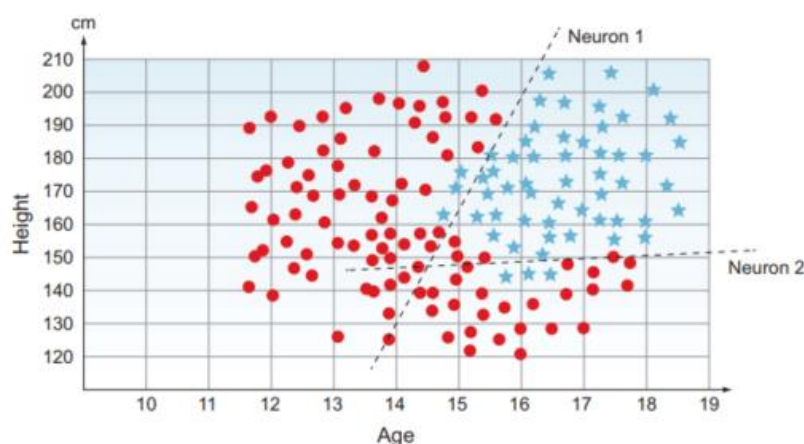


Figura 32. Em um conjunto de dados não linear, uma única linha reta não pode separar os dados de treinamento. Fonte: ELGENDY, 2020.

Ok, isso é definitivamente melhor do que a linha reta. Mas ainda vejo algumas previsões errôneas de cores. Podemos adicionar mais neurônios para que a função se ajuste melhor? Agora você está entendendo. Conceitualmente, quanto mais neurônios adicionarmos, melhor a rede se ajustará aos nossos dados de treinamento. Na verdade, se adicionarmos muitos neurônios, a rede se ajustará demais aos dados de treinamento (o que não é bom). Mas falaremos sobre isso mais tarde. A regra geral aqui é que quanto mais complexa for a rede, melhor ela aprenderá os recursos dos dados.

3.1.4. Perceptrons multicamadas

Vimos que um único perceptron funciona muito bem com conjuntos de dados simples que podem ser separados por uma linha. Mas, como você pode imaginar, o mundo real é muito mais complexo do que isso, e é nesse ponto que as redes neurais podem mostrar todo o seu potencial.

Para dividir um conjunto de dados não linear, precisamos de mais de uma linha. Isso significa que precisamos criar uma arquitetura para usar dezenas e centenas de neurônios em nossa rede neural. Vejamos o exemplo da Figura 33. Lembre-se de que um perceptron é uma função linear que produz uma linha reta. Portanto, para ajustar esses dados, tentamos criar uma forma semelhante a um triângulo que divide os pontos escuros. Parece que três linhas seriam suficientes.

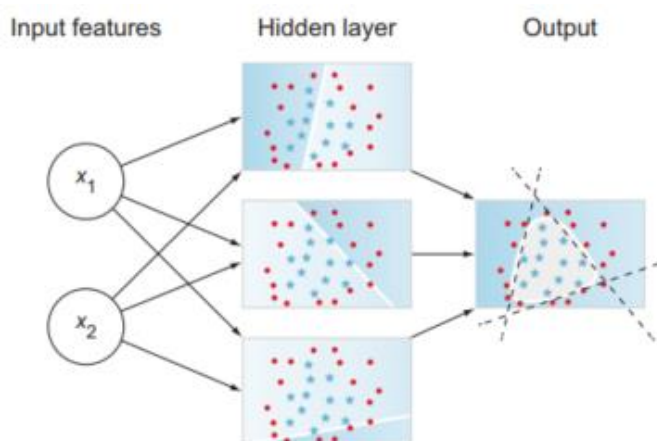


Figura 33. Pequena rede neural usada para modelar dados não lineares. Fonte: ELGENDY, 2020.

3.1.5. Arquitetura de perceptron multicamada

Vimos como uma rede neural pode ser projetada para ter mais de um neurônio. Vamos expandir essa ideia com um conjunto de dados mais complexo. O diagrama da Figura 34 foi extraído do site do playground do Tensorflow (<https://playground.tensorflow.org>). Tentamos modelar um conjunto de dados em espiral para distinguir entre duas classes. Para ajustar esse conjunto de dados, precisamos construir uma rede neural que contenha dezenas de neurônios. Uma arquitetura de rede neural muito comum é empilhar os neurônios em camadas umas sobre as outras, chamadas de camadas ocultas. Cada camada tem um número n de neurônios. As camadas são conectadas umas às outras por conexões de peso. Isso leva à arquitetura do perceptron de múltiplas camadas (MLP) mostrada na Figura 34.

Os principais componentes da arquitetura da rede neural são os seguintes:

- Camada de entrada - Contém o vetor de recursos.
- Camadas ocultas - Os neurônios são empilhados uns sobre os outros nas camadas ocultas. Elas são chamadas de camadas "ocultas" porque não vemos nem controlamos a entrada que vai para essas camadas nem a saída. para essas camadas ou a saída. Tudo o que fazemos é alimentar o vetor de recursos para a camada de entrada de entrada e ver o resultado que sai da camada de saída.
- Conexões de peso (bordas) - Os pesos são atribuídos a cada conexão entre os nós para refletir a importância de sua influência no resultado. Em termos de rede gráfica, essas conexões são chamadas de bordas que conectam os nós.

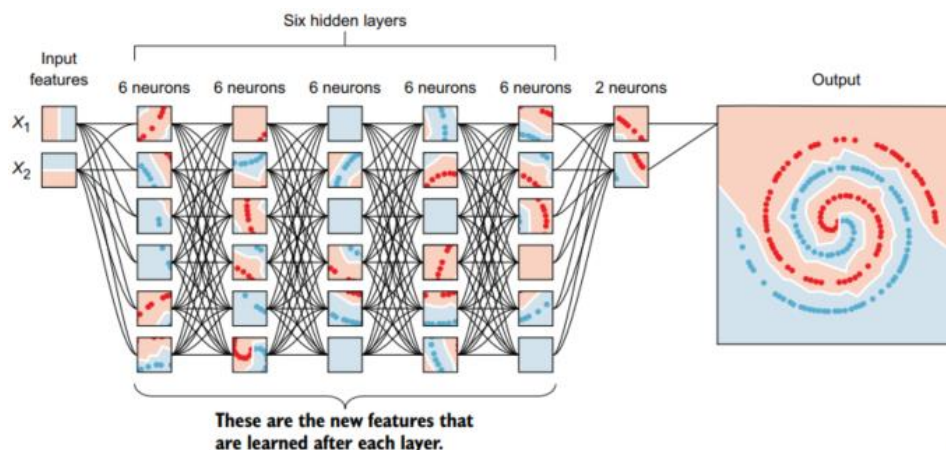


Figura 34. Representação de exemplo de playground do Tensorflow. Fonte: ELGENDY, 2020.

- Camada de saída - Obtemos a resposta ou a previsão do nosso modelo na camada de saída. Dependendo da configuração da rede neural, a saída final pode ser uma saída com valor real (problema de regressão) ou um conjunto de probabilidades (problema de classificação). Isso é determinado pelo tipo de função de ativação que usamos nos neurônios da camada de saída. Discutiremos os diferentes tipos de funções de ativação na próxima seção.

3.1.6. Funções de ativação

Ao criar uma rede neural, uma das decisões de projeto que você precisará tomar é a função de ativação a ser usada nos cálculos dos neurônios. As funções de ativação também são chamadas de funções de transferência ou não linearidades porque transformam a combinação linear de uma soma ponderada em um modelo não linear. Uma função de ativação é colocada no final de cada perceptron para decidir se esse neurônio deve ser ativado.

Por que usar funções de ativação? Por que não calcular apenas a soma ponderada de nossa rede e propagá-la pelas camadas ocultas para produzir uma saída? O objetivo da função de ativação é introduzir a não linearidade na rede. Sem ela, um perceptron multicamadas terá um desempenho semelhante ao de um único perceptron, independentemente do número de camadas que adicionarmos. As funções de ativação são necessárias para restringir o valor de

saída a um determinado valor finito. Vamos rever o exemplo de prever se um jogador será aceito, conforme mostrado no gráfico da Figura 35.

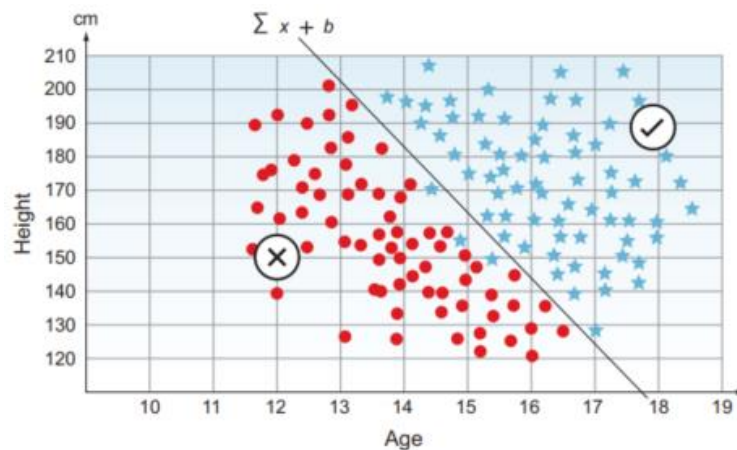


Figura 35. Previsão de aceitação de um jogador. Fonte: ELGENDY, 2020.

Primeiro, o modelo calcula a soma ponderada e produz a função linear (z):

$$z = \text{height} \cdot w_1 + \text{age} \cdot w_2 + b$$

A saída dessa função não tem limite, z pode ser literalmente qualquer número. Usamos uma função de ativação para envolver os valores de previsão em um valor finito. Neste exemplo, usamos uma função de degrau em que, se $z > 0$, então acima da linha (aceito) e se $z < 0$, então abaixo da linha (rejeitado). Portanto, sem a função de ativação, temos apenas uma função linear que produz um número, mas nenhuma decisão é tomada nesse perceptron. A função de ativação é o que decide se esse perceptron deve ser acionado. De fato, nos últimos anos, houve muito progresso na criação de ativações de última geração. Entretanto, ainda há relativamente poucas ativações que respondem pela grande maioria das necessidades de ativação. Vamos nos aprofundar em alguns dos tipos mais comuns de funções de ativação.

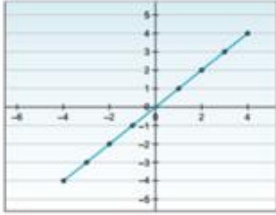
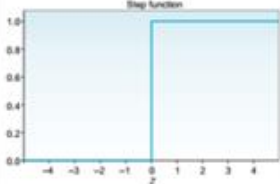
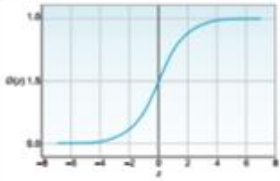
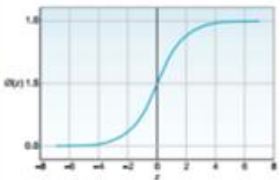
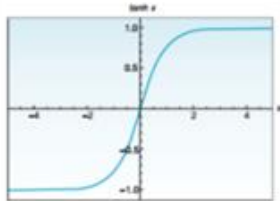
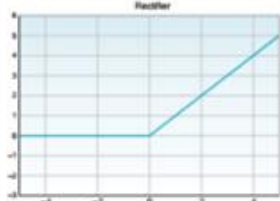
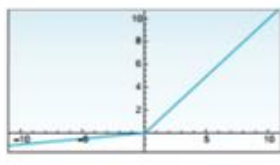
Activation function	Description	Plot	Equation
Linear transfer function (identity function)	The signal passes through it unchanged. It remains a linear function. Almost never used.		$f(x) = x$
Heaviside step function (binary classifier)	Produces a binary output of 0 or 1. Mainly used in binary classification to give a discrete value.		output = $\begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$
Sigmoid/logistic function	Squishes all the values to a probability between 0 and 1, which reduces extreme values or outliers in the data. Usually used to classify two classes.		$\sigma(x) = \frac{1}{1 + e^{-x}}$
Softmax function	A generalization of the sigmoid function. Used to obtain classification probabilities when we have more than two classes.		$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$
Hyperbolic tangent function (tanh)	Squishes all values to the range of -1 to 1. Tanh almost always works better than the sigmoid function in hidden layers.		$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Rectified linear unit (ReLU)	Activates a node only if the input is above zero. Always recommended for hidden layers. Better than tanh.		$f(x) = \max(0, x)$
Leaky ReLU	Instead of having the function be zero when $x < 0$, leaky ReLU introduces a small negative slope (around 0.01) when (x) is negative.		$f(x) = \max(0.01x, x)$

Figura 36. Funções de ativação mais comuns. Fonte: ELGENDY, 2020.

3.2 Exercício Prático (opcional)

Desenvolva uma simulação de uma rede neural no **Play Ground** <https://playground.tensorflow.org/>. Escolhendo um conjunto de dados de entrada que deseja usar. Configure a **taxa de aprendizado**, a **função de ativação** e o **tipo de problema**.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

3.3 Rede Neural Convolucional (CNN)

Uma rede neural convolucional (CNN) é uma rede neural artificial profunda e alimentada, na qual a rede neural preserva a estrutura hierárquica ao aprender representações de recursos internos e generalizar os recursos nos problemas comuns de imagem, como reconhecimento de objetos e outros problemas de visão computacional. Não se restringe a imagens, também obtém resultados de última geração em problemas de processamento de linguagem natural e reconhecimento de fala.

3.3.1 Diferentes Camadas em uma CNN

Uma CNN consiste em várias camadas, conforme mostrado na Figura 37.

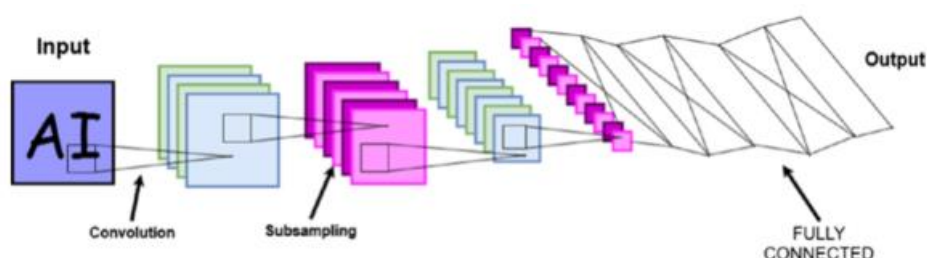


Figura 37. Camadas em uma CNN. Fonte: MANASWI, 2018.

As camadas de convolução consistem em filtros e mapas de imagem. Considere que a imagem de entrada em escala de cinza tem um tamanho de 5×5, que é uma matriz de 25 valores de pixel. Os dados da imagem são expressos como uma matriz tridimensional de largura × altura × canais.

A convolução tem como objetivo extrair recursos da imagem de entrada e, portanto, preserva a relação espacial entre os pixels ao aprender os recursos da imagem usando pequenos quadrados de dados de entrada. É possível esperar invariância rotacional, invariância de translação e invariância de escala. Por exemplo, uma imagem de gato girada ou uma imagem de gato redimensionada pode ser facilmente identificada por uma CNN devido à etapa de convolução. Você desliza o filtro (matriz quadrada) sobre a imagem original (aqui, 1 pixel) e, em cada posição determinada, calcula a multiplicação por elementos (entre as matrizes do filtro e a imagem original) e adiciona os resultados da multiplicação para obter o número inteiro final que forma os elementos da matriz de saída.

A subamostragem é simplesmente o agrupamento médio com pesos aprendidos por mapa de características, conforme mostrado na Figura 38.

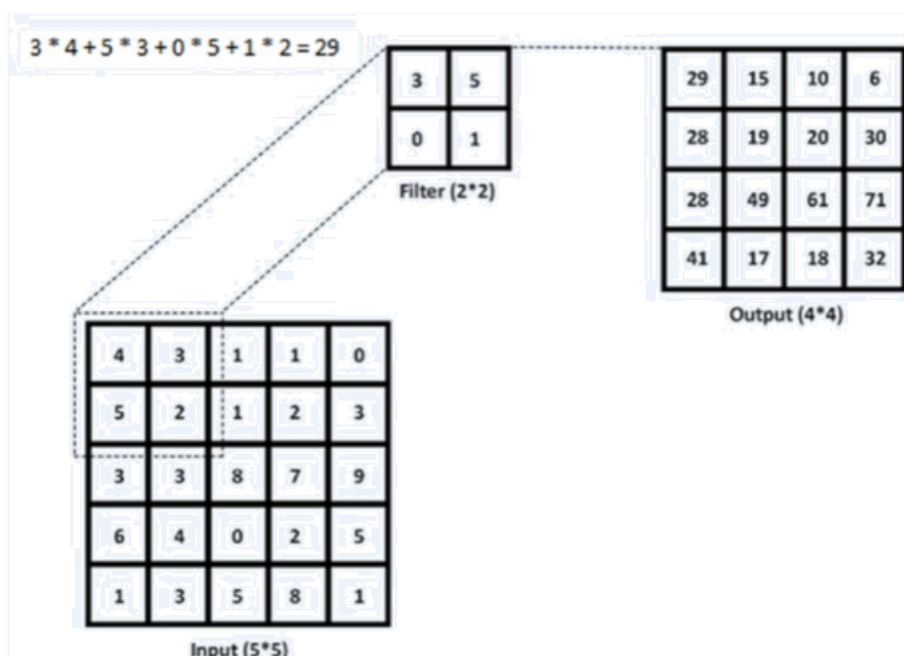


Figura 38. Subamostragem. Fonte: MANASWI, 2018.

Conforme mostrado na Figura 38, os filtros têm pesos de entrada e geram um neurônio de saída. Digamos que você defina uma camada convolucional com

seis filtros e campos receptivos com 2 pixels de largura e 2 pixels de altura e use uma largura de passo padrão de 1, e o preenchimento padrão seja definido como 0. Cada filtro recebe entrada de uma seção de imagem de 2x2 pixels. Em outras palavras, são 4 pixels de cada vez. Portanto, pode-se dizer que serão necessários 4 + 1 (bias) pesos de entrada.

O volume de entrada é 5x5x3 (largura x altura x número de canais), há seis filtros de tamanho 2x2 com stride 1 e pad 0. Portanto, o número de parâmetros nessa camada para cada filtro é $2*2*3 + 1 = 13$ parâmetros (adicionado+1 para bias). Como há seis filtros, você tem $13*6 = 78$ parâmetros.

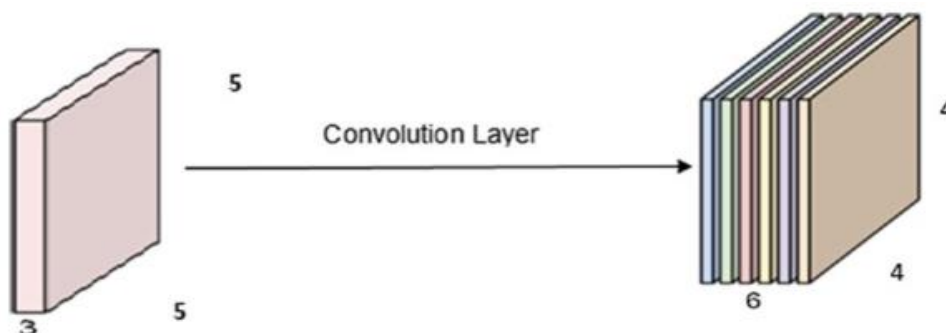


Figura 39. Volume de entrada. Fonte: MANASWI, 2018.

Aqui está um resumo:

- O volume de entrada é de tamanho $W1 \times H1 \times D1$.
- O modelo requer hiperparâmetros: número de filtros
- filtros (f), stride (S), quantidade de preenchimento zero (P).
- Isso produz um volume de tamanho $W2 \times H2 \times D2$.
- $W2 = (W1 - f + 2P) / S + 1 = 4$.
- $H2 = (H1 - f + 2P) / S + 1 = 4$.
- $D2 = \text{Número de filtros} = f = 6$.

As camadas de agrupamento reduzem os mapas de ativação das camadas anteriores. Ela é seguida por uma ou mais camadas convolucionais e consolida todos os recursos que foram aprendidos nos mapas de ativação das camadas anteriores. Isso reduz o excesso de ajuste dos dados de treinamento e generaliza os recursos representados pela rede. O tamanho do campo receptivo é quase sempre definido como 2x2 e use um passo de 1 ou 2 (ou mais) para garantir que não haja sobreposição. Você usará uma operação máxima

para cada campo receptivo de modo que a ativação seja o valor máximo de entrada. Aqui, cada quatro números são mapeados para apenas um número, portanto, o número de pixels cai para um quarto do original nessa etapa, de acordo com a Figura 40.

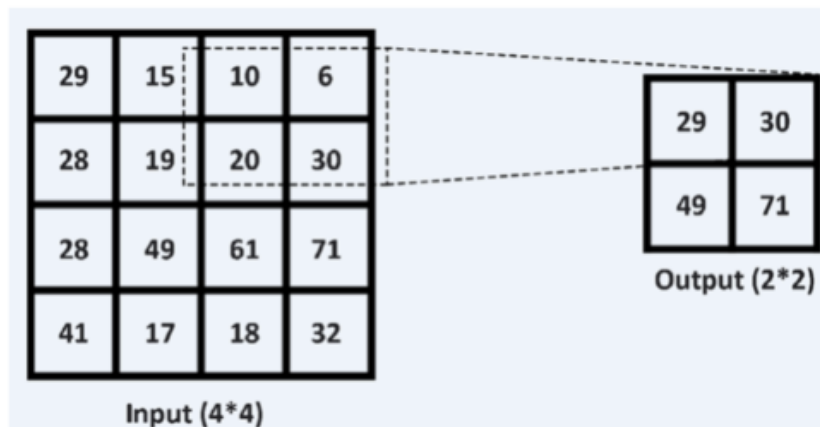


Figura 40. Maxpooling - redução do número de pixels. Fonte: MANASWI, 2018.

Uma camada totalmente conectada é uma camada de rede neural artificial de alimentação direta. Essas camadas têm uma função de ativação não linear para gerar probabilidades de previsão de classe. Elas são usadas no final, depois que todos os recursos são identificados e extraídos por camadas convolucionais e consolidados pelas camadas de pooling na rede. Aqui, as camadas ocultas e de saída são as camadas totalmente conectadas.

3.3.2 Arquiteturas de CNN

Uma CNN é uma arquitetura de rede neural profunda feed-forward composta por algumas camadas convolucionais, cada uma seguida por uma camada de pooling, função de ativação e, opcionalmente, normalização de lote. Ela também é composta de camadas totalmente conectadas. À medida que uma imagem passa pela rede, ela fica menor, principalmente devido ao pooling máximo. A camada final gera a previsão das probabilidades de classe.

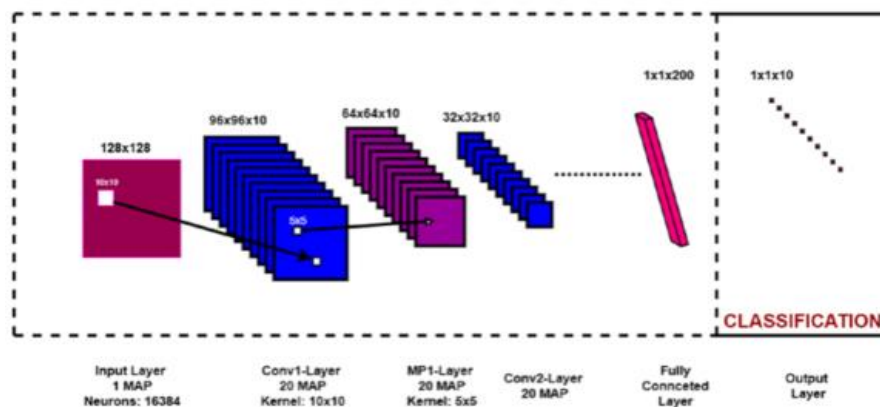


Figura 41. Arquitetura CNN para classificação. Fonte: MANASWI, 2018.

Nos últimos anos, foram desenvolvidas muitas arquiteturas que fizeram um enorme progresso no campo da classificação de imagens. Redes pré-treinadas premiadas (VGG16, VGG19, ResNet50, InceptionV3 e Xception) foram usadas para vários desafios de classificação de imagens, inclusive imagens médicas. A aprendizagem por transferência é o tipo de prática em que você usa modelos pré-treinados, além de algumas camadas, e pode ser usada para resolver desafios de classificação de imagens em todos os campos.

3.3.3 Aplicação do Tensorflow e Keras em Modelos CNN

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 42. 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 43. 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'])
```

3.3.4 Métodos para Treinar e Salvar 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, mostrada na Figura 44, 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, pode 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 44. 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. Conforme mostrado na Figura 45.

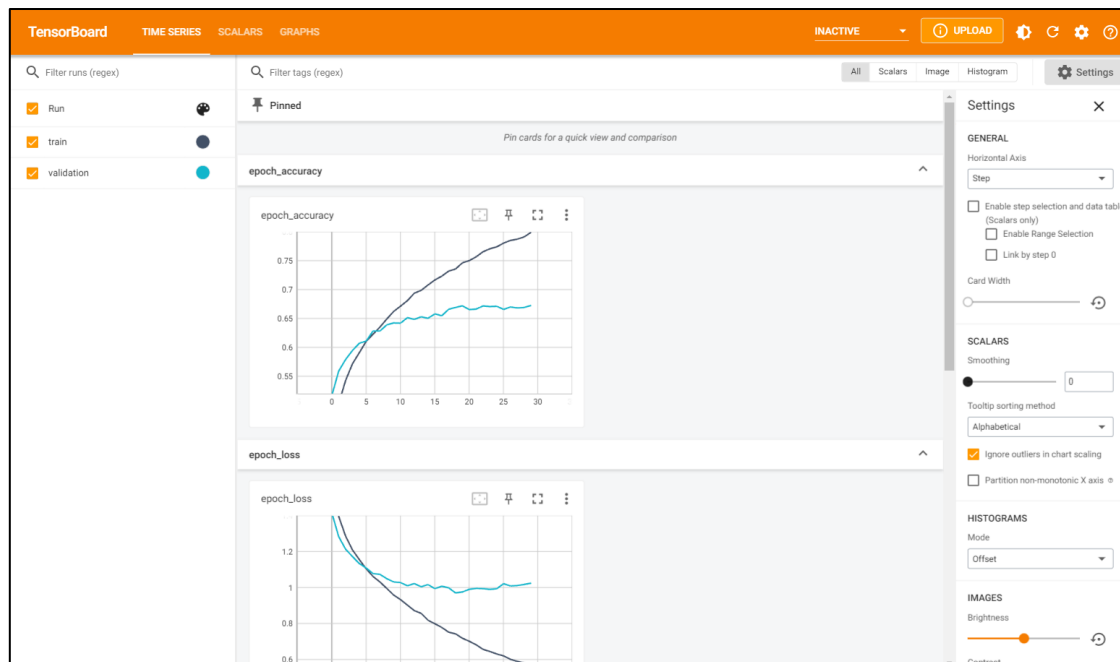


Figura 45. Painel do Tensorboard no VSCode. Fonte: Autoria própria.

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')
```

3.3.5 Carregando um Modelo de CNN Pré-treinado

Após treinar e salvar o nosso modelo, podemos utilizar a função `load_model` do 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 Figura 46, mostra 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 46. 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()

```

3.4 Exercício Prático (opcional)

Desenvolva um programa em python no **VS Code** para realizar a classificação de objetos quaisquer na webcam utilizando **tensorflow** e **keras**.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

3.5 You only look once (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 47. 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.

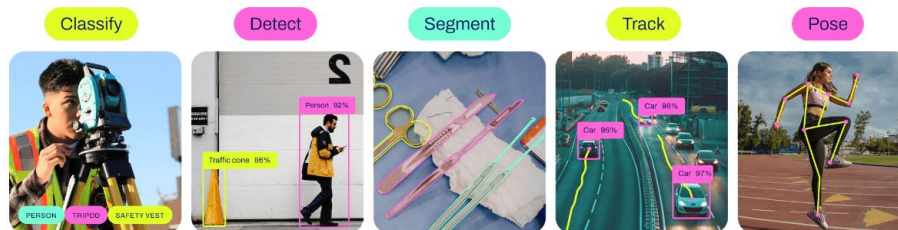


Figura 48. Tarefas disponíveis no YOLO. Fonte: Ultralytics.

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 49. 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 ^{val} 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 50. Modelos pretreinados YOLOv8. Fonte: Ultralytics.

3.5.1 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 ao qual foi aberto anteriormente (*'yolov8n'* no nosso exemplo até aqui).

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

3.5.2 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)
```

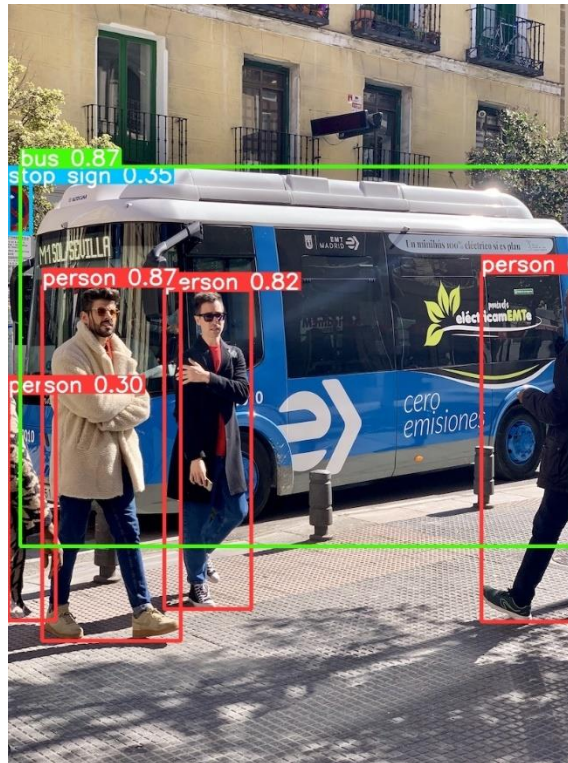


Figura 51. Inferência em imagem com YOLOv8. Fonte: Ultralytics.

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
```

3.5.3 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 acuraria 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, tal método pode ser visualizado na Figura 52.

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	161	0.85	0.783	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 52. 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.

3.5.4 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), mostradas na Figura 53.

```
6 0.44053125 0.5111666666666667 0.128640625 0.48116666666666674  
5 0.5369843750000001 0.5231458333333333 0.10953125000000004 0.4566041666666666
```

Figura 53. Arquivo txt para treino com YOLO. Fonte: Autoria própria.

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



Figura 54. Ferramenta Roboflow. Fonte: Roboflow.

Cadastre-se e acesse o banco de imagem do link 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.

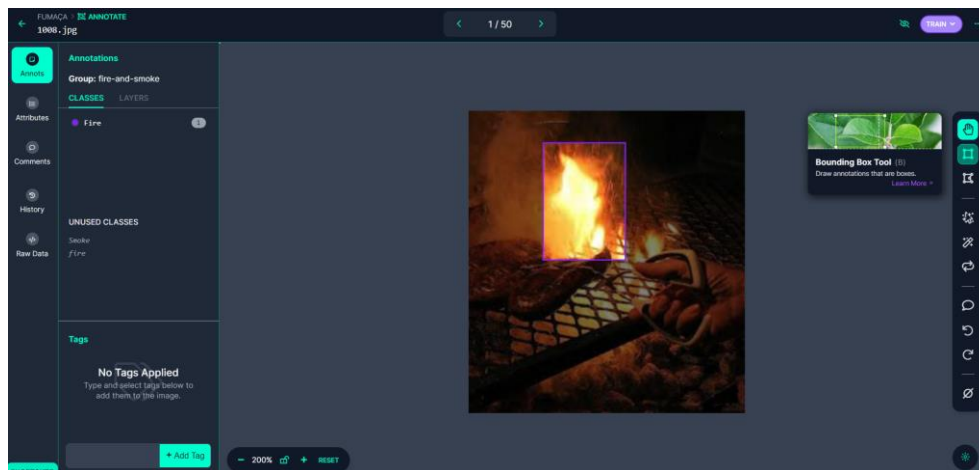


Figura 55. Anotação de objetos no Roboflow. Fonte: Autoria própria.

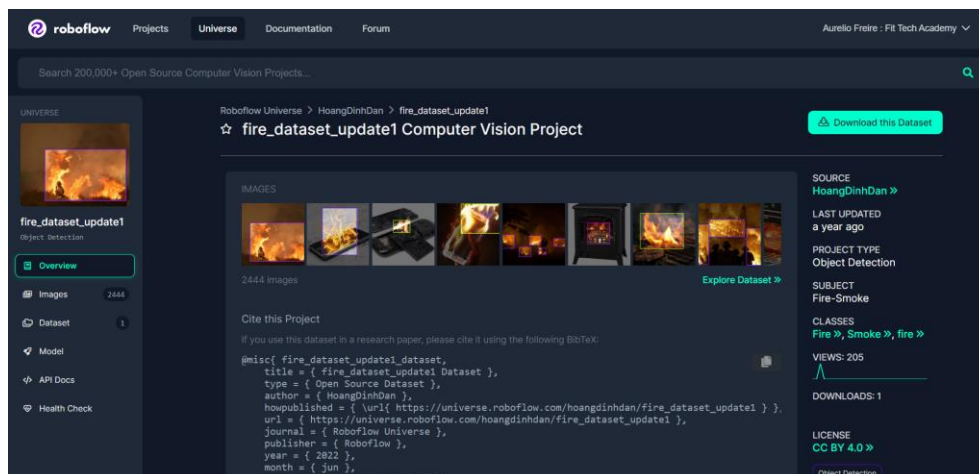


Figura 56. Banco de imagens de fogo e fumaça. Fonte: Roboflow.

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

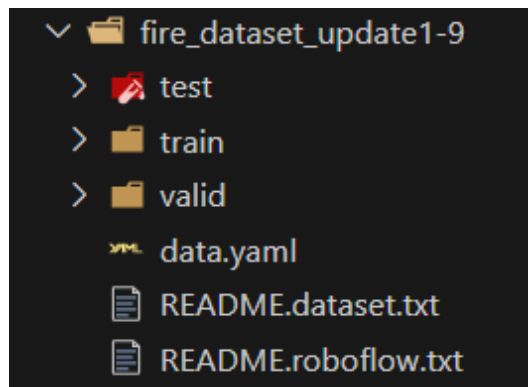


Figura 59. Diretório de imagens “fire_dataset_pudate1-9”. Fonte: Autoria própria.

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”.

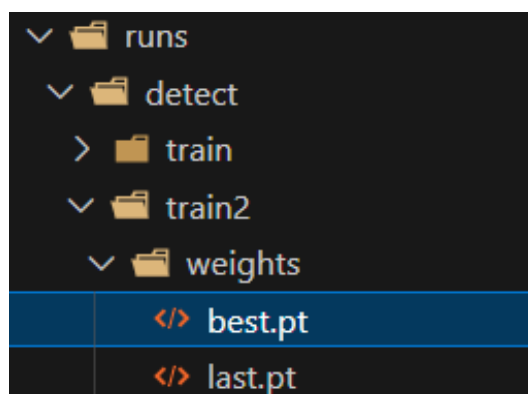


Figura 60. Melhores pesos do treino 2. Fonte: Autoria própria.

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')
```

3.6 Exercício Prático (opcional)

Desenvolva um programa em python no **VS Code** para realizar a classificação de objetos quaisquer na webcam utilizando **YOLO**.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

4. Situação Problema (opcional)

Desenvolva um programa em python no **VS Code** para realizar a identificação de uma pessoa utilizando **YOLO** e encontrar as coordenadas do centro da **bounding box** (cx e cy), tais coordenadas devem ser enviadas ao robô colaborativo (TM5 ou TM12) para realizar **Tracking** (rastreamento) do rosto da pessoa.

Ao terminar o desafio, insira as evidências sobre a atividade (estrutura) em arquivos: na sua estrutura original, caso os diretórios sejam HTML, CSS, JS, JPG e/ou PNG, ou pode-se zipar os arquivos (ZIP e RAR), inserindo todos os arquivos de uma vez só. As evidências são: Código-fonte do exercício (arquivo do código fonte e foto da montagem/ tela).

Conclusão

A abordagem desta apostila permite ao leitor uma interação ativa com os principais conteúdos sobre o robô colaborativo presente na indústria. Ensinar como um robô colaborativo opera foi trabalhada para dar ao aluno autonomia, estímulo, senso crítico e contribuir para uma aprendizagem mais efetiva.

Nesta, fora exposto um pouco sobre o conceito de robô colaborativo, funcionamento e as principais funcionalidades do TM5-700. Este material é baseado em artigos científicos, periódicos, revistas científicas, manuais do fabricante e livros científicos.

A apostila explica um pouco do conceito de um robô colaborativo, em seguida comenta sobre o hardware do equipamento, abordando aplicação, instalação, configuração e principais características. Também fora abordado a linguagem de programação, mostrando as principais instruções e uma breve explicação sobre redes de comunicação presentes.

Assim, o conteúdo do curso de Robô Colaborativo é 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 automação industrial com robôs colaborativos apresentado neste curso esteja aguçado, para que você pratique e conheça ainda mais as maravilhas da automação.

Referências

- CHOLLET, Francois. **Deep learning with Python. Simon and Schuster**, 2021.
- ELGENDY, Mohamed. Deep learning for vision systems. Simon and Schuster, 2020.
- GÉRON, Aurélien. **Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow**. " O'Reilly Media, Inc.", 2022.
- HUNT, John. **A beginners guide to Python 3 programming**. Springer, 2019.
- ISIS, G.2018. O que faz um robô colaborativo? Todas as aplicações possíveis!, disponível em: <<https://pollux.com.br/blog/o-que-faz-um-robo-colaborativo-todas-as-aplicacoes-possiveis/>> acessado em Março de 2022;
- KETKAR, Nikhil; MOOLAYIL, Jojo. **Deep learning with Python: learn best practices of deep learning models with PyTorch**. New York, NY, USA:: Apress, 2021.
- LAMBERT, Kenneth A. **Fundamentals of Python: first programs**. Cengage Learning, 2018.
- LEE, Kent D. **Python programming fundamentals**. London: Springer, 2011.
- LONG, Liangqu; ZENG, Xiangming. **Beginning Deep Learning with TensorFlow: Work with Keras, MNIST Data Sets, and Advanced Neural Networks**. 2022.
- MOOCARME, Matthew; ABDOLAHNEJAD, Mahla; BHAGWAT, Ritesh. The Deep Learning with Keras Workshop: Learn how to define and train neural network models with just a few lines of code. Packt Publishing Ltd, 2020.
- RUSSANO, Euan; AVELINO, Elaine Ferreira. **Fundamentals of Machine Learning Using Python**. Arcler Press, 2020.
- SWAMYNATHAN, Manohar. **Mastering machine learning with python in six steps: A practical implementation guide to predictive data analytics using python**. Manohar Swamynathan, 2017.
- VASILEV, Ivan et al. **Python Deep Learning: Exploring deep learning techniques and neural network architectures with Pytorch, Keras, and TensorFlow**. Packt Publishing Ltd, 2019.

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

Revisão	Descrição	Razão	Autor	Data
A	Elaboração e revisão inicial	Elaboração inicial/Revisão pedagógica	Jailson Bina/Luana Forte	29/10/23
B	Modificações dos exercícios	Processo de Melhorias	Jailson Bina/Luana Forte	29/04/24



BOM CURSO!