Disciplina:

Análise de Imagem e Visão Computacional

Professor: Octavio Santana

PUC Minas

Aula 4

Redes Neurais Convolucionais (CNN)

Professor: Octavio Santana

PUC Minas

Objetivos da Aula

Ao final desta aula, os alunos serão capazes de:

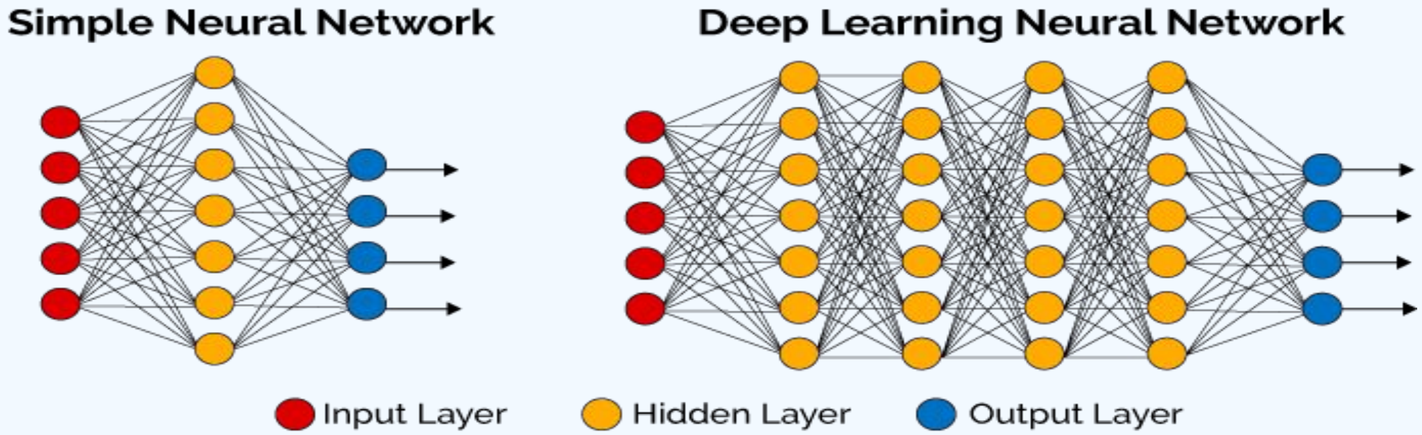
1. Compreender os fundamentos das Redes Neurais Convolucionais (CNNs) e sua aplicação em visão computacional.
2. Identificar e explicar os principais componentes de uma CNN, incluindo camadas convolucionais, de pooling e totalmente conectadas.
3. Implementar uma CNN básica para classificação de imagens utilizando a biblioteca TensorFlow
4. Avaliar o desempenho de modelos de CNN e aplicar técnicas para melhorar sua acurácia.

PUC Minas

Introdução às Redes Neurais Convolucionais

Visão Computacional com Aprendizado Profundo

Usa camadas de neurônios matemáticos para processar dados, compreender a fala humana e reconhecer objetos visualmente. A informação é passada através de cada camada, com a saída da camada anterior fornecendo entrada para a próxima camada. A primeira camada em uma rede é chamada de camada de entrada, enquanto a última é chamada de camada de saída. Todas as camadas entre as duas são referidas como camadas ocultas. Cada camada é tipicamente um algoritmo simples e uniforme contendo um tipo de função de ativação.



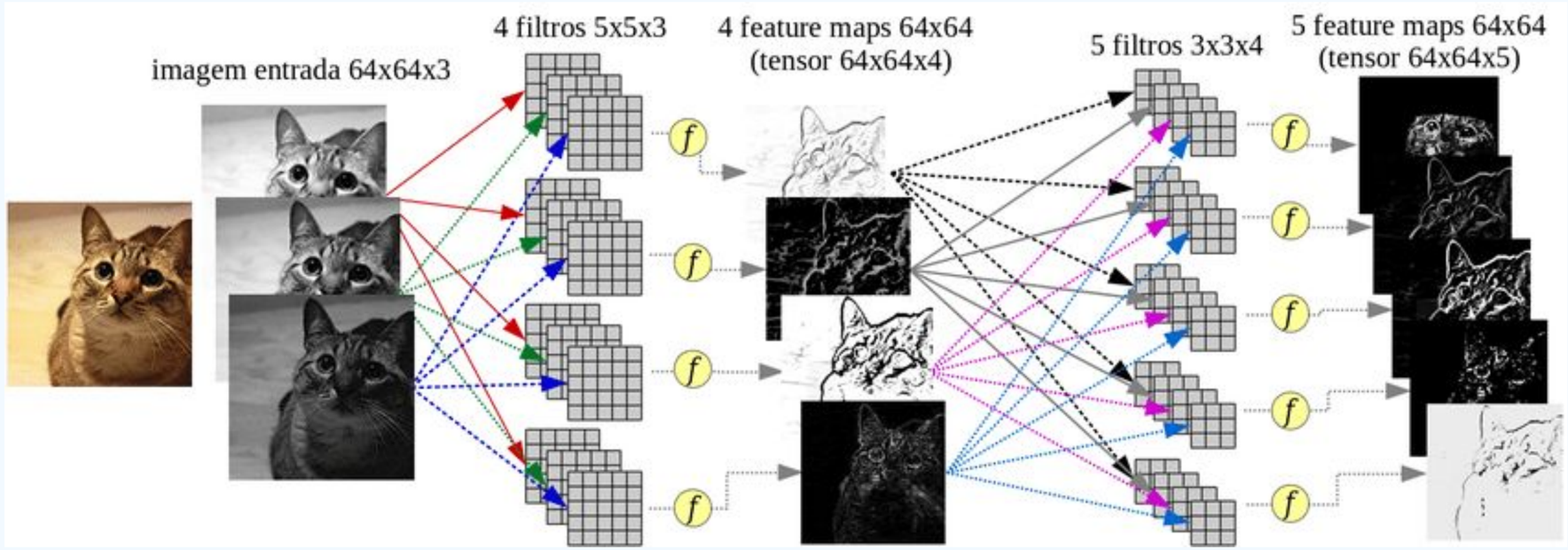
É pouco eficiente usar redes com camadas totalmente conectadas para classificar imagens. A razão é que tal arquitetura de rede não leva em conta a estrutura espacial das imagens. Por exemplo, ela trata os pixels de entrada que estão distantes e próximos exatamente no mesmo nível. Tais conceitos de estrutura espacial devem ser inferidos dos dados de treinamento.

PUC Minas

Visão Computacional com Aprendizado Profundo

Deep Learning revolucionou a Visão Computacional com redes neurais convolucionais (CNNs), permitindo modelos mais precisos para reconhecimento de padrões complexos.

É uma classe de redes neurais de aprendizado profundo. Resumindo, pense na CNN como um algoritmo de aprendizado de máquina que pode receber uma imagem de entrada, atribuir importância (pesos e vieses que podem ser aprendidos) a vários aspectos/objetos na imagem e ser capaz de diferenciar um do outro.



Arquitetura de uma CNN

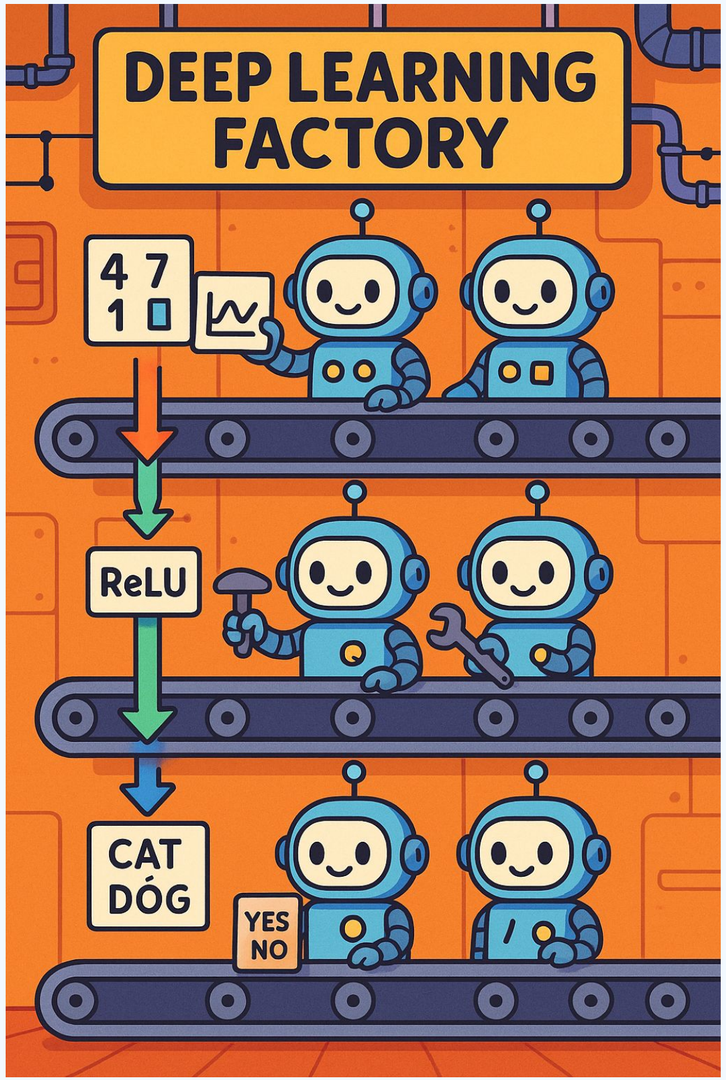
PUC Minas

Como Funciona um Modelo de Aprendizado Profundo

Imagine um modelo de aprendizado profundo como uma "fábrica de decisões" com várias camadas de trabalhadores (neurônios).

Cada trabalhador recebe informações, faz um cálculo (usando uma função de ativação) e passa o resultado adiante.

Essas funções permitem que o modelo aprenda relações complexas nos dados, não apenas coisas simples e lineares.

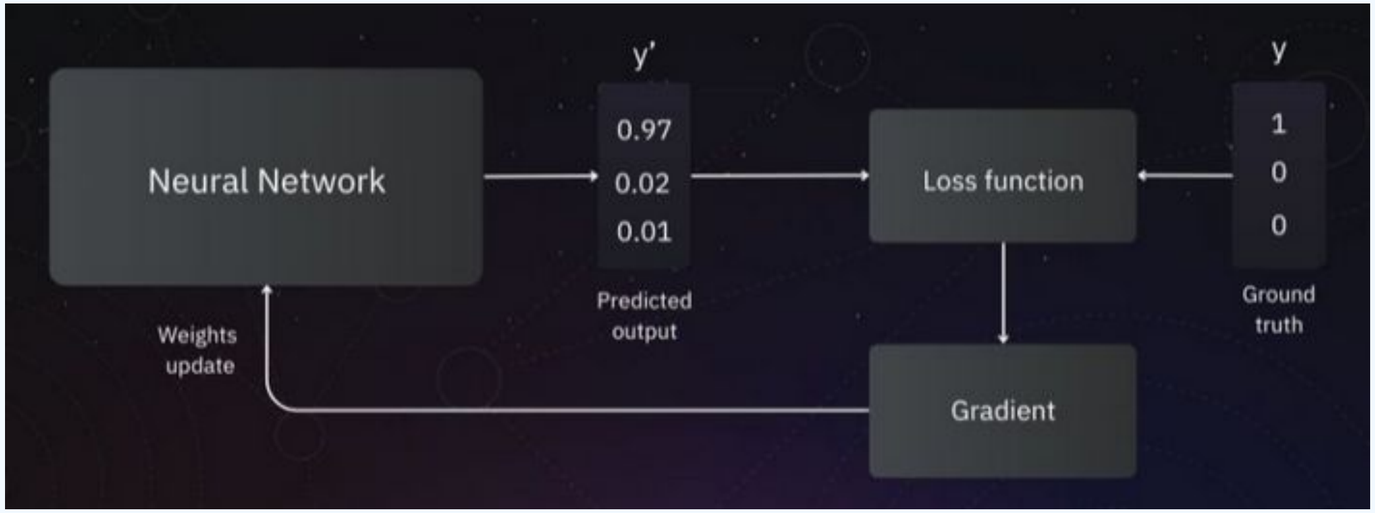


PUC Minas

Como Funciona um Modelo de Aprendizado Profundo

Como o Modelo Aprende?

- **Pesos e Vieses:** Cada conexão entre neurônios tem um "peso" (importância) e um "viés" (ajuste fino). Eles são como botões que o modelo gira para melhorar suas respostas.
- **Função de Perda:** O modelo comete erros no início, e essa função mede "quão errado" ele está comparado à resposta certa.
- **Otimizador:** É o "treinador" do modelo. Ele usa cálculo (gradientes e retropropagação) para ajustar os pesos e vieses, diminuindo o erro a cada passo.
- **Estratégia do Otimizador:** Ele busca o melhor caminho no "labirinto" de possíveis ajustes, evitando ficar preso em soluções ruins (mínimos locais) e tentando achar a melhor solução possível.

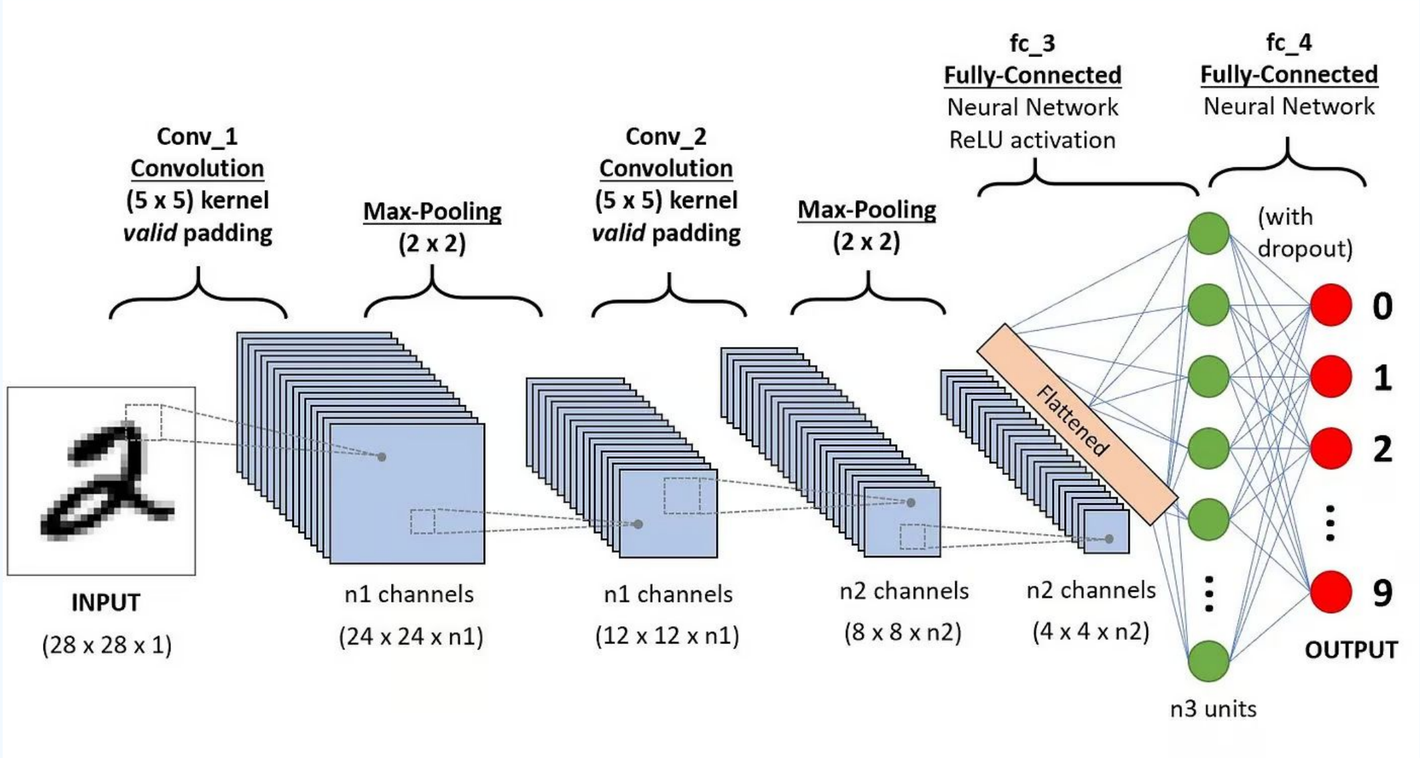


PUC Minas

Principais componentes de uma CNN

A rede neural convolucional é composta de quatro partes principais.

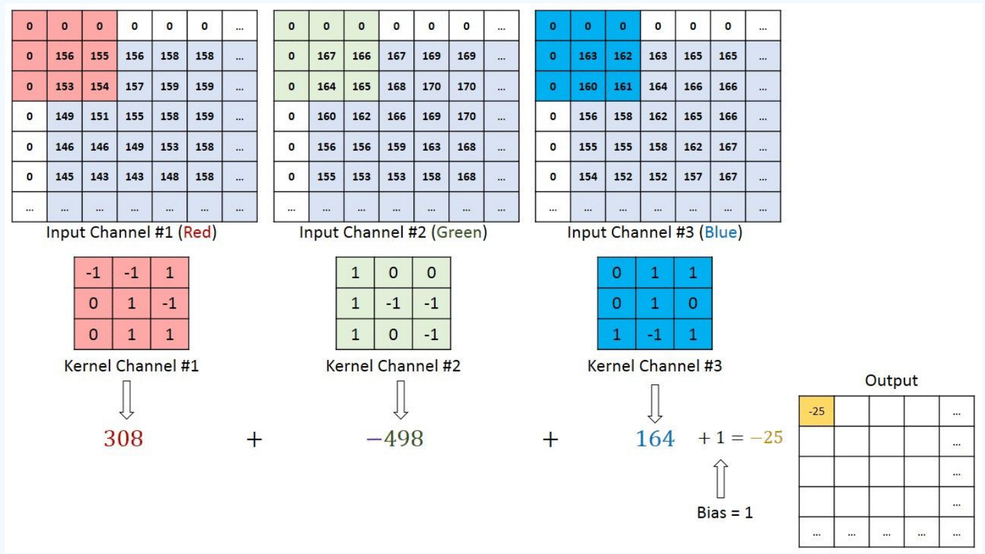
- Camadas convolucionais
- Unidade Linear Retificada (ReLU para abreviar)
- Camadas de pooling
- Camadas totalmente conectadas



PUC Minas

Camada de Convolução

- **Função Principal:** Aplicar filtros (ou kernels) que percorrem a imagem para detectar características locais, como bordas, texturas e padrões.
- **Como Funciona:** Um filtro pequeno (por exemplo, 3×3 ou 5×5) é "deslizado" sobre a imagem (ou sobre a saída da camada anterior) realizando a operação de convolução. Em cada posição, o filtro multiplica os valores dos pixels correspondentes e soma o resultado, gerando um único valor no mapa de ativação.
- **Vantagens da Convolução:** A propriedade de invariância local – ou seja, a capacidade de detectar a mesma característica independentemente da sua posição na imagem – é fundamental para o bom desempenho em tarefas de reconhecimento.



Hiperparâmetros da Camada de Convolucional

Tamanho do Kernel (Filtro):

- Comuns: 3×3, 5×5, 7×7.
- Efeito:
  - Kernels menores (3×3) capturam detalhes finos.
  - Kernels maiores (5×5) capturam características mais amplas.

Stride (Passo do Filtro):

- Define quantos pixels o kernel "pula" ao deslizar.
- Stride = 1: Filtro move-se um pixel por vez (maior feature map).
- Stride = 2: Filtro move-se dois pixels (reduz dimensionalidade).

Padding:

- Problema: Convolução reduz o tamanho da imagem.
- Solução: Adicionar bordas (zeros ou valores) para manter dimensões.
  - padding="valid": Sem preenchimento (saída menor).
  - padding="same": Saída com mesmo tamanho da entrada.

Número de Filtros:

- Cada filtro gera um feature map diferente.
- Exemplo:
  - Conv2D(32, (3,3)) → 32 filtros 3×3 → Saída com 32 feature maps.
  - Mais filtros = Maior capacidade de aprendizado (mas mais custo computacional).

Funções de Ativação em CNNs

Conceito Fundamental

As funções de ativação são componentes críticos em Redes Neurais Convolucionais que introduzem não-linearidade ao sistema, permitindo que a rede aprenda padrões complexos. Sem elas, a CNN se comportaria como uma simples regressão linear, incapaz de modelar relações complexas em dados visuais.

Propriedades Essenciais

- Não-linearidade: Permite a aproximação de funções complexas
- Diferenciabilidade: Necessária para o backpropagation
- Controle de amplitude: Limita ou normaliza a saída dos neurônios

Principais Funções de Ativação

ReLU (Rectified Linear Unit)

- Fórmula: f(x) = max(0, x)
- Características:
  - Computacionalmente eficiente
  - Evita o problema do gradiente vanishing
  - Pode causar "neurônios mortos" (Dying ReLU)

LeakyReLU

- Fórmula: f(x) = max(αx, x) (onde α ≈ 0.01)
- Vantagens sobre ReLU:
  - Resolve o problema dos neurônios mortos
  - Mantém a eficiência computacional

Softmax

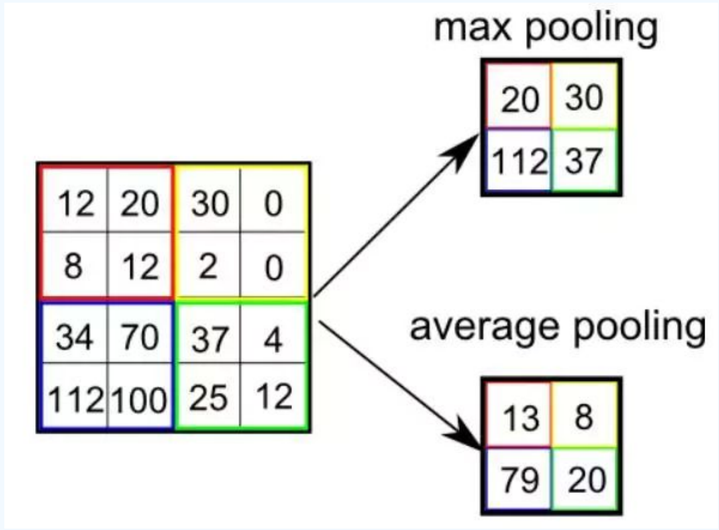
- Fórmula: f(x_i) = e^x_i / Σ(e^x_j)
- Aplicação típica:
  - Última camada em problemas de classificação multiclasse
  - Produz distribuição de probabilidades

Sigmoid

- Fórmula: $\sigma(x) = 1 / (1 + e^{-x})$
- Características Principais:
  - Última camada em problemas de classificação binária ou classificação multilabel
  - Mapeia valores reais para o intervalo (0, 1)

Camada de Pooling

- Objetivo: Reduzir a dimensionalidade dos mapas de ativação, mantendo as características relevantes.
- Tipos Comuns:
  - Max Pooling: Seleciona o valor máximo dentro de uma região especificada (por exemplo, uma janela 2×2), destacando a característica mais “forte”.
  - Average Pooling: Calcula a média dos valores na região, suavizando as ativações.
- Benefícios: Redução da carga computacional e do risco de overfitting, além de proporcionar invariância a pequenas translações na imagem.



Hiperparâmetros da Camada de Polling

Tamanho da Janela (kernel):

- Comuns: 2x2, 3×3
- Regra geral: Quanto maior o kernel, maior a redução de dimensionalidade.

Stride (Passo):

- Stride = tamanho do kernel (ex.: kernel 2×2, stride=2) ⟶ Redução pela metade.
- Se stride < tamanho do kernel, há sobreposição de janelas (overlapping pooling).

Padding:

- Raramente usado em polling (ao contrário da convolução)

Por que usar Polling?

- Sem Pooling
  - Alto custo computacional
  - Sensível a pequenas mudanças
  - Risco de overfitting
- Com Pooling
  - Redução eficiente de dados
  - Maior invariância a translações
  - Controla overfitting

PUC Minas

Camada Totalmente Conectada (Dense)

Conceito Fundamental

A camada totalmente conectada (Dense) é um componente essencial em redes neurais, incluindo CNNs, onde todos os neurônios da camada anterior são conectados a todos os neurônios da camada atual. Sua principal função é:

- Combinar features extraídas pelas camadas convolucionais/pooling
- Realizar classificação ou regressão com base nas características aprendidas

Arquitetura e Funcionamento

Estrutura Matemática

Cada neurônio recebe uma combinação linear das saídas da camada anterior, seguida por uma função de ativação:

$$y = f(\mathbf{w}^T\mathbf{x} + b)$$

w: Vetor de pesos

x: Vetor de entrada (features achatadas)

b: Viés (bias)

f: Função de ativação (ex.: ReLU, softmax)

Processo em CNNs

- Flatten (Achatamento):
  - Converte feature maps 3D (altura × largura × canais) em vetor 1D
  - Exemplo: Feature map 7×7×64 → Vetor de 3136 elementos
- Conexão Dense:
  - Cada neurônio da camada densa recebe todas as entradas do vetor achatado

PUC Minas

Camada Totalmente Conectada (Dense)

Hiperparâmetros e Configuração

- Número de Neurônios
  - Última camada:
    - Classificação binária: 1 neurônio (sigmoid)
    - Multiclasse: *N* neurônios (sof
      onde *N* = número de classes
    - Regressão: 1 neurônio (ativação linear)
  - Camadas ocultas:
    - Valores comuns: 64, 128, 256, 512
    - Trade-off: Mais neurônios → maior capacidade de aprendizado, mas risco de overfitting

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

model = models.Sequential([
    # Camadas convolucionais/pooling...
    layers.Flatten(),  # Achatamento para Dense
    layers.Dense(128, activation='relu'),  # Camada oculta
    layers.Dense(10, activation='softmax')  # Saida para 10 classes
])
```

- Funções de Ativação

Camada	Função Recomendada	Motivo
Oculta	ReLU/LeakyReLU	Não-linearidade eficiente
Saída (Binária)	Sigmoid	Probabilidade classe positiva
Saída (Multiclasse)	Softmax	Distribuição de probabilidades

Funções de Ativação na Última Camada para Diferentes Tipos de Classificação

Classificação Binária (2 Classes)

Cenário:

Problemas do tipo "sim/não" (ex.: tumor benigno/maligno)

Função Recomendada:

Sigmoid (activation='sigmoid')

Características:

- Produz um valor entre 0 e 1 (probabilidade da classe positiva)
- Fórmula: σ(x) = 1 / (1 + e^-x)

Implementação no Tensorflow

```
model.add(layers.Dense(1, activation='sigmoid'))  # Única saida
```

Função de Perda Correspondente

```
model.compile(loss='binary_crossentropy', ...)
```

PUC Minas

Funções de Ativação na Última Camada para Diferentes Tipos de Classificação

Classificação Multiclasse (Classes Exclusivas)

Cenário:

Objetos que pertencem a exatamente uma classe (ex.: MNIST - dígitos 0-9)

Função Recomendada:

Sigmoid (activation=softmax)

Características:

- Produz uma distribuição de probabilidades (soma = 1)
- Fórmula: softmax(x_i) = e^x_i / Σ(e^x_j)

Implementação no Tensorflow

```
model.add(layers.Dense(10, activation='softmax'))  # 10 classes
```

Função de Perda Correspondente

```
model.compile(loss='categorical_crossentropy', ...)
```

PUC Minas

Funções de Ativação na Última Camada para Diferentes Tipos de Classificação

Classificação Multilabel (Múltiplas Classes Simultâneas)

Cenário:

Objetos que podem pertencer a várias classes simultaneamente (ex.: imagem com "cachorro" e "grama")

Função Recomendada:

Sigmoid (activation=sigmoid)

Características:

- Cada saída é independente (valores entre 0 e 1)
- Permite múltiplas ativações simultâneas

Implementação no Tensorflow

```
model.add(layers.Dense(5, activation='sigmoid'))  # 5 possiveis labels
```

Função de Perda Correspondente

```
model.compile(loss='binary_crossentropy', ...)  # Mesmo para múltiplas saidas
```

PUC Minas

Funções de Ativação na Última Camada para Diferentes Tipos de Classificação

Tabela Comparativa

Tipo de Classificação	Função de Ativação	Nº de Neurônios	Função de Perda	Formato dos Labels
Binária	Sigmoid	1	binary_crossentropy	0 ou 1
Multiclasse	Softmax	N (classes)	categorical_crossentropy	One-hot encoded
Multilabel	Sigmoid	N (labels)	binary_crossentropy	Vetor de 0s e 1s

Como preparar os rótulos para cada caso?

- **Binária**: y = [0, 1, 0, 1] (vetor 1D)
- **Multiclasse**: y = [[0,0,1], [1,0,0]] (one-hot)
- **Multilabel**: y = [[1,0,1], [0,1,1]] (multi-hot)

Dica Pro: Para multiclasse com rótulos inteiros (não one-hot), use sparse_categorical_crossentropy!

PUC Minas

Funções de Perda (Loss Functions)

A função de perda quantifica **quão bem (ou mal)** o modelo está performando comparando suas previsões com os valores reais.

Propriedades Fundamentais

- **Diferenciabilidade:** Necessária para o cálculo de gradientes.
- **Continuidade:** Garante suavidade na otimização.
- **Sensibilidade a Erros:** Deve penalizar previsões incorretas de forma adequada.

Funções de Perda para Tarefas Específicas

Classificação Binária

- Quando usar
  - Dois estados mutuamente exclusivos (Sim/Não)
  - Amostras pertencem apenas uma categoria

```
# Modelo CNN com logits
model = tf.keras.Sequential([
    layers.Rescaling(1./255),  # Normalização
    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(1)  # Saída como logits
])

# Configuração com BinaryCrossentropy
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=['accuracy', tf.keras.metrics.AUC()]
)
```

Funções de Perda para Tarefas Específicas

Classificação Multiclasse

Categorical Cross-Entropy

- Quando usar:
  - Quando seus rótulos estão em formato one-hot encoded
  - Exemplo: *y_true = [[0,0,1], [1,0,0]]* (para 3 classes)

```
# Última camada DEVE usar softmax (ou setar from_logits=True)
model.add(layers.Dense(3, activation='softmax'))  # 3 classes

# Função de perda
loss = tf.keras.losses.CategoricalCrossentropy(
    from_logits=False,  # Se False, a última camada deve aplicar softmax
    label_smoothing=0.0  # Opcional: regularização
)

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

Sparse Categorical Cross-Entropy

- Quando usar:
  - Quando seus rótulos são inteiros (não one-hot)
  - Exemplo: *y_true = [2, 0]* (mesmas classes do exemplo anterior)

```
# Última camada PODE usar logits diretamente (recomendado)
model.add(layers.Dense(3))  # 3 classes (sem ativação)

# Função de perda
loss = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True,  # Mais estável numericamente
    reduction='auto'   # Padrão: soma sobre batch e divide por batch_size
)

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

Funções de Perda para Tarefas Específicas

Classificação Multiclasse

Tabela Comparativa

Característica	CategoricalCrossentropy	SparseCategoricalCrossentropy
Formato dos rótulos	One-hot (e.g., [0,1,0])	Inteiro (e.g., 1)
Última camada	Softmax (ou logits=True)	Logits (recomendado) ou Softmax
Eficiência de memória	Menos eficiente	Mais eficiente
Casos de uso típicos	Quando one-hot já está pré-processado	Rótulos crus de datasets (e.g., tf.data)

Funções de Perda para Tarefas Específicas

Classificação Multilabel

- Quando usar:
  - Múltiplas classes podem coexistir na mesma amostra
  - Labels não são mutuamente exclusivos
  - Exemplos:
    - Reconhecimento de objetos em imagens (cachorro, grama, sol)
    - Diagnóstico de múltiplas doenças
  - Quando seus rótulos estão em formato multi-hot encoded
  - Exemplo: *y_true = [[0,1,1], [1,0,1]]* (para 3 classes)

```
# Mesma BinaryCrossentropy mas com comportamento multilabel
loss = tf.keras.losses.BinaryCrossentropy(
    from_logits=True,  # Aplica sigmoid internamente
    reduction=tf.keras.losses.Reduction.AUTO  # Equivalente a 'sum_over_batch_size'
)

model.compile(
    optimizer='adam',
    loss=loss,
    metrics=[
        tf.keras.metrics.BinaryAccuracy(threshold=0.5),
        tf.keras.metrics.Precision(top_k=2)  # Avalia top 2 predições
    ]
)
```

Algoritmos de Otimização

Quando treinamos uma rede neural para visão computacional, estamos tentando resolver o seguinte problema matemático:

$$\min_{\theta} J(\theta) = \min_{\theta} \left(\frac{1}{N} \sum_{i=1}^{N} \ell(f(x_i; \theta), y_i) \right)$$

Onde:

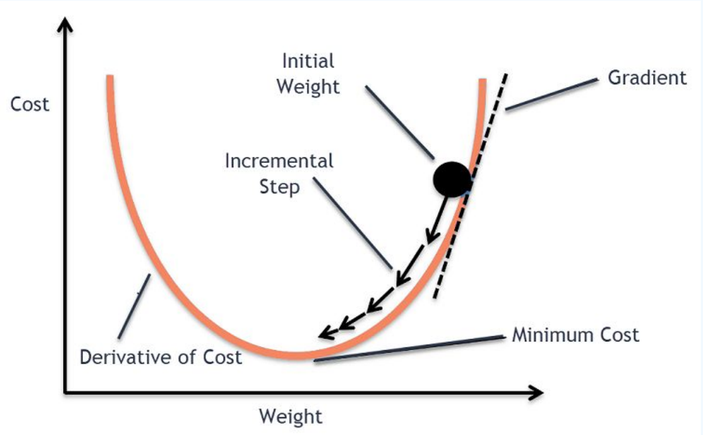
- **θ:** Representa todos os parâmetros do modelo (pesos das camadas)
- **J(θ):** Função de perda (loss) que queremos minimizar
- **f(x_i; θ)**: Predição do modelo para a entrada x_i
- **ℓ**: Função de erro para uma única amostra (ex: cross-entropy)
- **N**: Número total de amostras no dataset

PUC Minas

Gradiente Descendente - O Algoritmo Base

O gradiente descendente é o algoritmo mais básico para otimização em redes neurais. Ele funciona atualizando os parâmetros na direção oposta ao gradiente da função de perda:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t)$$

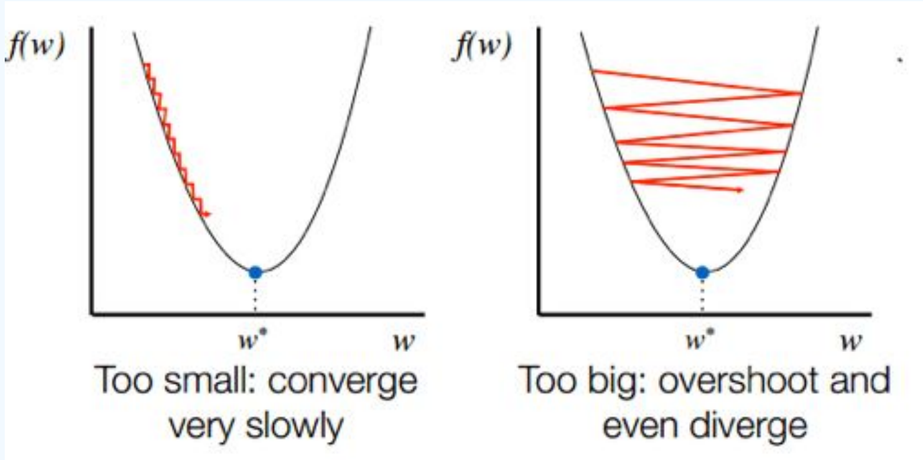


Onde:

- **θ_t:** Parâmetros na iteração t
- **η (eta):** Taxa de aprendizado (learning rate)
- **∇J(θ_t):** Gradiente da função de perda em relação a θ

Taxa de Aprendizado (η)

- Controla o tamanho do passo em cada atualização:
  - **η** muito pequeno: Convergência lenta
  - **η** muito grande: Pode divergir ou oscilar



Lista de Otimizadores para Visão Computacional

SGD (Stochastic Gradient Descent)

- Descrição: Versão básica do gradiente descendente com mini-batches.
- Melhor Para: Problemas simples ou quando combinado com momentum e learning rate decay.

```
tf.keras.optimizers.SGD(learning_rate=0.01)
```

SGD com Momentum

- Descrição: Adiciona inércia às atualizações para reduzir oscilações.
- Melhor Para: CNNs tradicionais ou superfícies de loss irregulares.

```
tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
```

Lista de Otimizadores para Visão Computacional

Adam (Adaptive Moment Estimation)

- Descrição: Combina momentum e adaptação individual de learning rates por parâmetro.
- Melhor Para: Arquiteturas modernas (Transformers, CNNs profundas) e problemas não convexos.

```
tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
```

AdamW

- Descrição: Adam com weight decay desacoplado (melhor regularização L2).
- Melhor Para: Treino de Transformers/ViTs ou quando há overfitting em datasets pequenos.

```
import tensorflow_addons as tfa

optimizer = tfa.optimizers.AdamW(learning_rate=0.001, weight_decay=0.004)
```

Lista de Otimizadores para Visão Computacional

Adagrad

- Descrição: Ajusta learning rates acumulando gradientes passados.
- Melhor Para: Problemas com features esparsas (menos usado atualmente).

```
tf.keras.optimizers.Adagrad(learning_rate=0.01)
```

RMSprop

- Descrição: Adapta learning rates dividindo pelo RMS dos gradientes recentes.
- Melhor Para: RNNs ou problemas com gradientes instáveis.

```
tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
```

PUC Minas

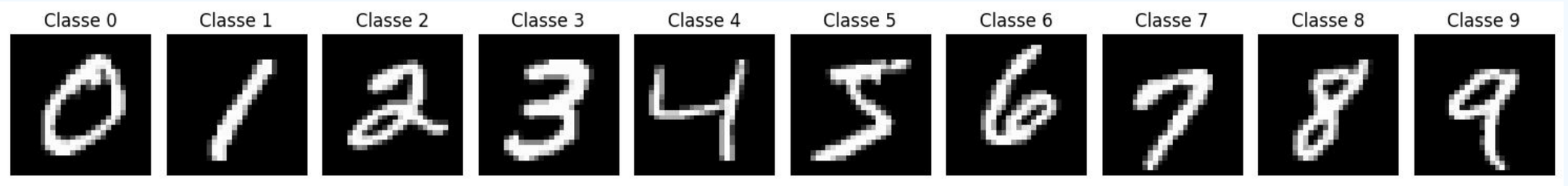
Implementação prática de uma CNN

PUC Minas

Objetivo

Neste material, vamos implementar uma **Rede Neural Convolucional (CNN)** usando **TensorFlow/Keras** para classificação de dígitos escritos à mão utilizando o dataset **MNIST**.

O MNIST é um dataset fundamental para aprendizado em visão computacional, contendo imagens de dígitos manuscritos (0-9) em escala de cinza, com 60.000 imagens de treino e 10.000 de teste.



Configuração do ambiente de desenvolvimento

Antes de começar, instale as bibliotecas necessárias:

```
pip install tensorflow numpy matplotlib seaborn scikit-learn
```

- **TensorFlow/Keras:** Para construção e treinamento da CNN.
- **NumPy:** Para manipulação de arrays.
- **Matplotlib e Seaborn:** Para visualização de imagens e resultados.
- **Scikit-learn:** Para métricas como matriz de confusão.

Carregamento e Pré-processamento do Dataset (MNIST)

Carregando o dataset MNIST

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
import numpy as np

# Carregar o dataset MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

load_data() retorna 4 arrays:

- *x_train*: 60,000 imagens 28x28 para treino
- *y_train*: 60,000 labels (0-9) correspondentes
- *x_test:* 10,000 imagens 28x28 para teste
- *y_test*: 10,000 labels para teste

Carregamento e Pré-processamento do Dataset (MNIST)

Normalizando os pixels para [0,1]

```
# Normalizar os pixels para [0, 1]
x_train = x_train / 255.0
x_test = x_test / 255.0
```

Por que dividir por 255?

- Valores de pixel originais variam de 0 (preto) a 255 (branco)
- Redes neurais trabalham melhor com valores entre 0 e 1
- Ajuda na convergência durante o treinamento

PUC Minas

Carregamento e Pré-processamento do Dataset (MNIST)

Redimensionamento para incluir o canal de cor.

```
# Redimensionar para incluir o canal de cor (1 canal, pois é grayscale)
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
```

Explicação do reshape:

- -1: Mantém o número original de amostras
- 28, 28: Dimensões altura/largura
- 1: Canal de cor (escala de cinza)
- Formato necessário para camadas Conv2D no TensorFlow

Antes do reshape

```
x_train.shape
✓ 0.0s
(60000, 28, 28)
```

Depois do reshape

```
x_train = x_train.reshape((-1, 28, 28, 1))
x_train.shape
✓ 0.0s
(60000, 28, 28, 1)
```

Carregamento e Pré-processamento do Dataset (MNIST)

Converte os labels para one-hot encoding

```
# Converter labels para one-hot encoding (opcional, mas útil para redes com softmax)
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

O que isso faz?

- Converte labels numéricos em vetores binários:
  - Ex: 3 ⟶ [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
- Necessário quando:
  - Usamos categorical_crossentropy como loss function
  - A saída da rede tem 10 neurônios (Softmax)

Alternativa (se não usar one-hot):

- Usar sparse_categorical_crossentropy e manter labels como inteiros

PUC Minas

Construção de uma CNN com Tensorflow/Keras

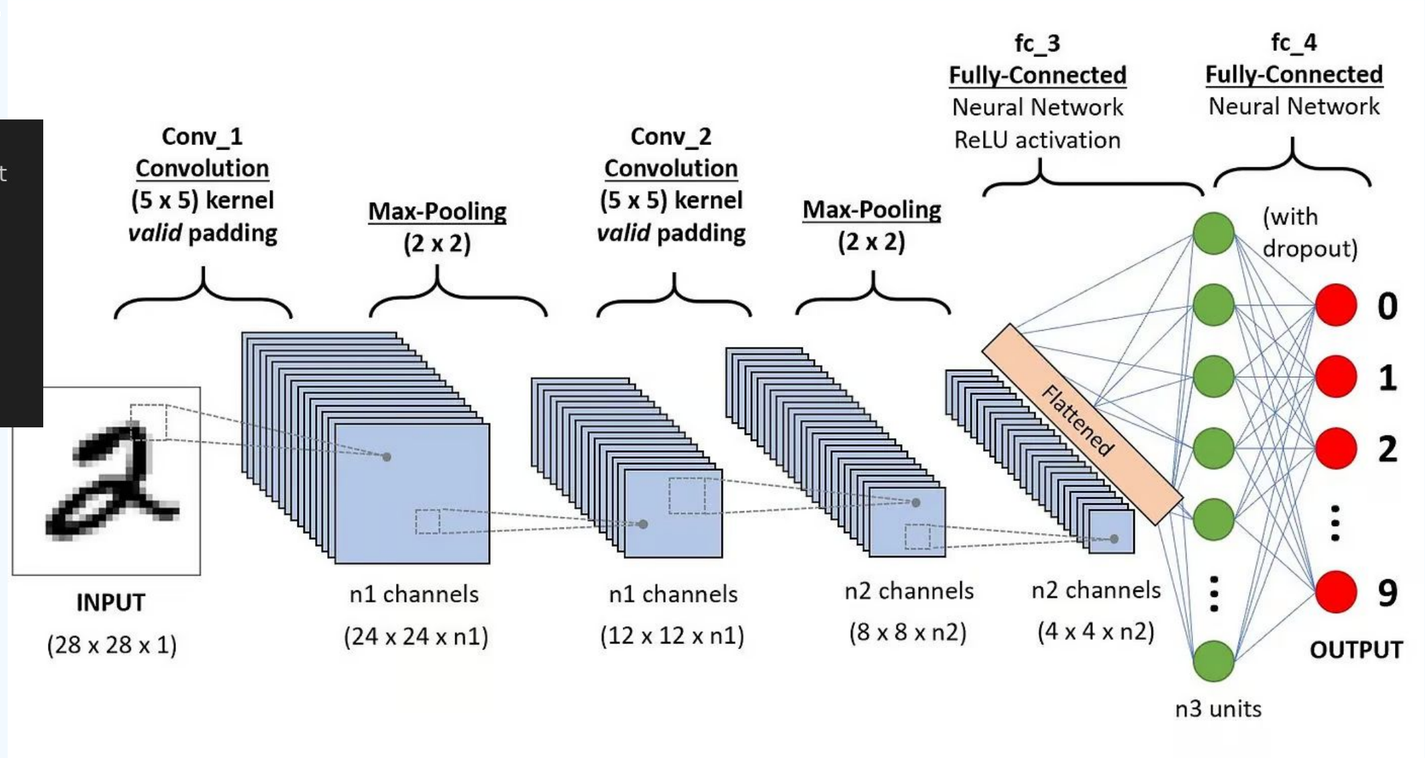
Vamos criar uma CNN simples com:

- Camadas Convolucionais (*Conv2D*) para extração de features.
- Camadas de Pooling (*MaxPooling2D*) para redução dimensional.
- Camadas Densas (*Dense*) para classificação.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Input

# Definir o modelo CNN
model = Sequential(name='MNIST_CNN')

# Camada de entrada
model.add(
    Input(shape=(28, 28, 1), name='input_layer')
)
```



PUC Minas

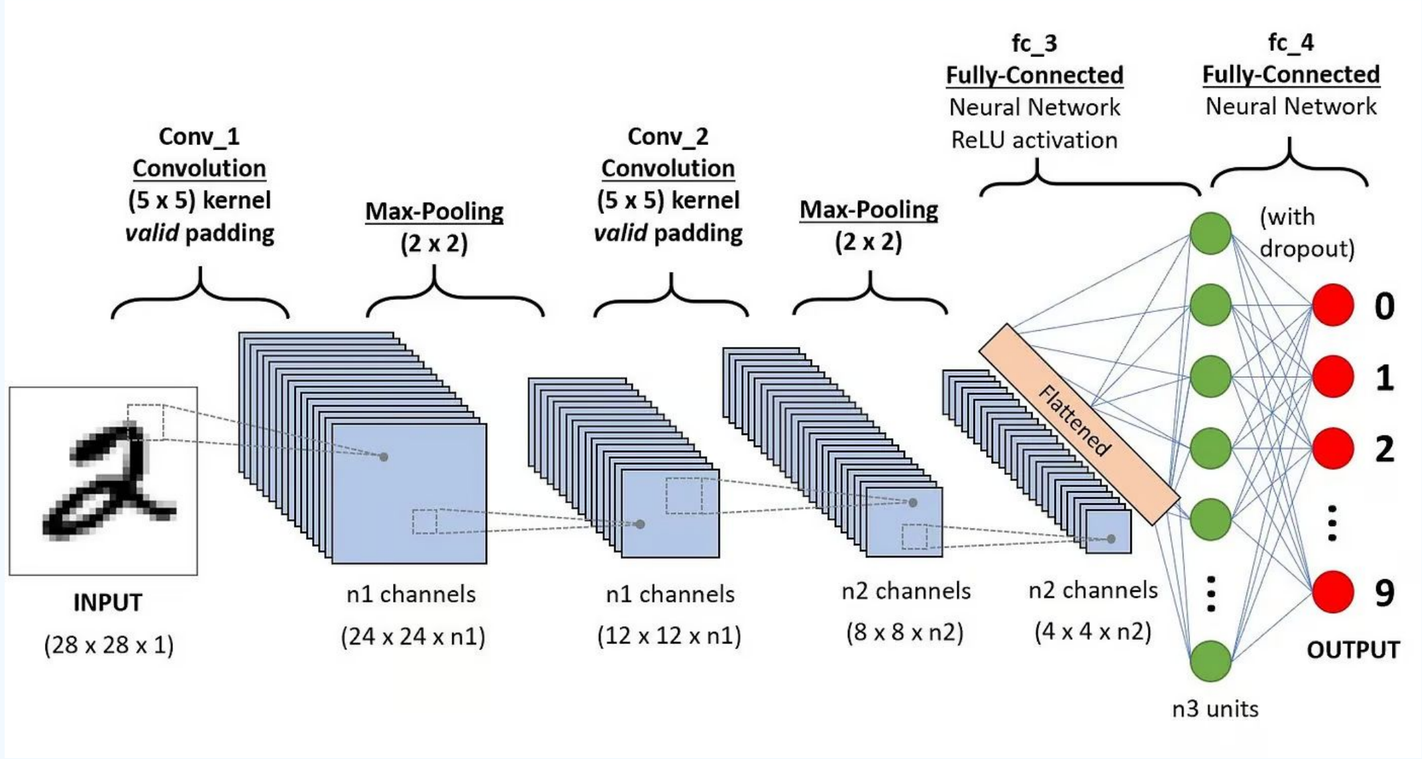
Construção de uma CNN com Tensorflow/Keras

Vamos criar uma CNN simples com:

- Camadas Convolucionais (*Conv2D*) para extração de features.
- Camadas de Pooling (*MaxPooling2D*) para redução dimensional.
- Camadas Densas (*Dense*) para classificação.

```python
# Primeira camada convolucional
model.add(
    Conv2D(
        filters=32,  # Número de filtros
        kernel_size=(5, 5),  # Tamanho do filtro
        activation='relu',  # Função de ativação
        name='conv1',  # Nome da camada
    )
)

# Camada de pooling
model.add(
    MaxPooling2D(
        pool_size=(2, 2),  # Tamanho do pooling
        name='pool1',  # Nome da camada
    )
)
```



PUC Minas

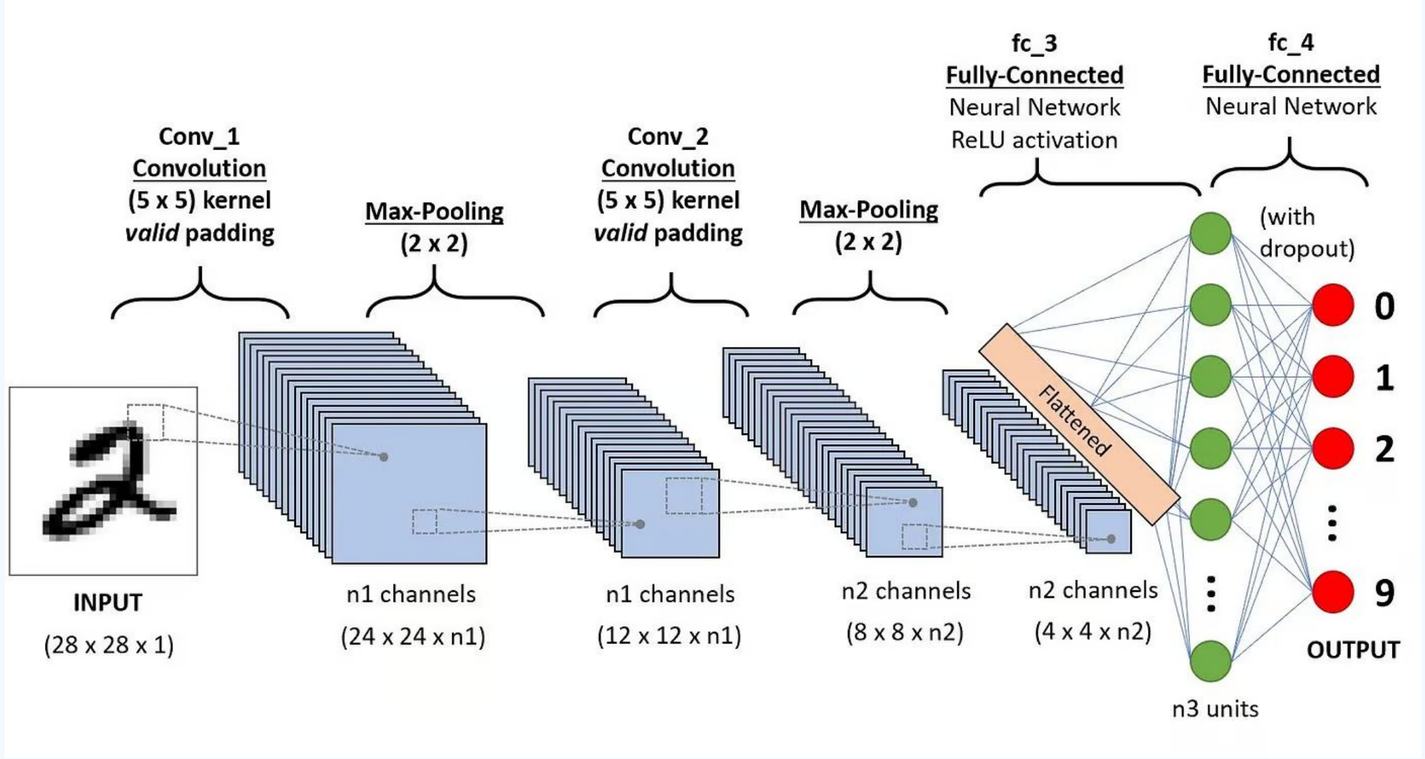
Construção de uma CNN com Tensorflow/Keras

Vamos criar uma CNN simples com:

- Camadas Convolucionais (*Conv2D*) para extração de features.
- Camadas de Pooling (*MaxPooling2D*) para redução dimensional.
- Camadas Densas (*Dense*) para classificação.

```
# Segunda camada convolucional
model.add(
    Conv2D(
            filters=64,  # Número de filtros
            kernel_size=(5, 5),  # Tamanho do filtro
            activation='relu',  # Função de ativação
            name='conv2',  # Nome da camada
        )
)

# Segunda camada de pooling
model.add(
    MaxPooling2D(
            pool_size=(2, 2),  # Tamanho do pooling
            name='pool2',  # Nome da camada
        )
)
```



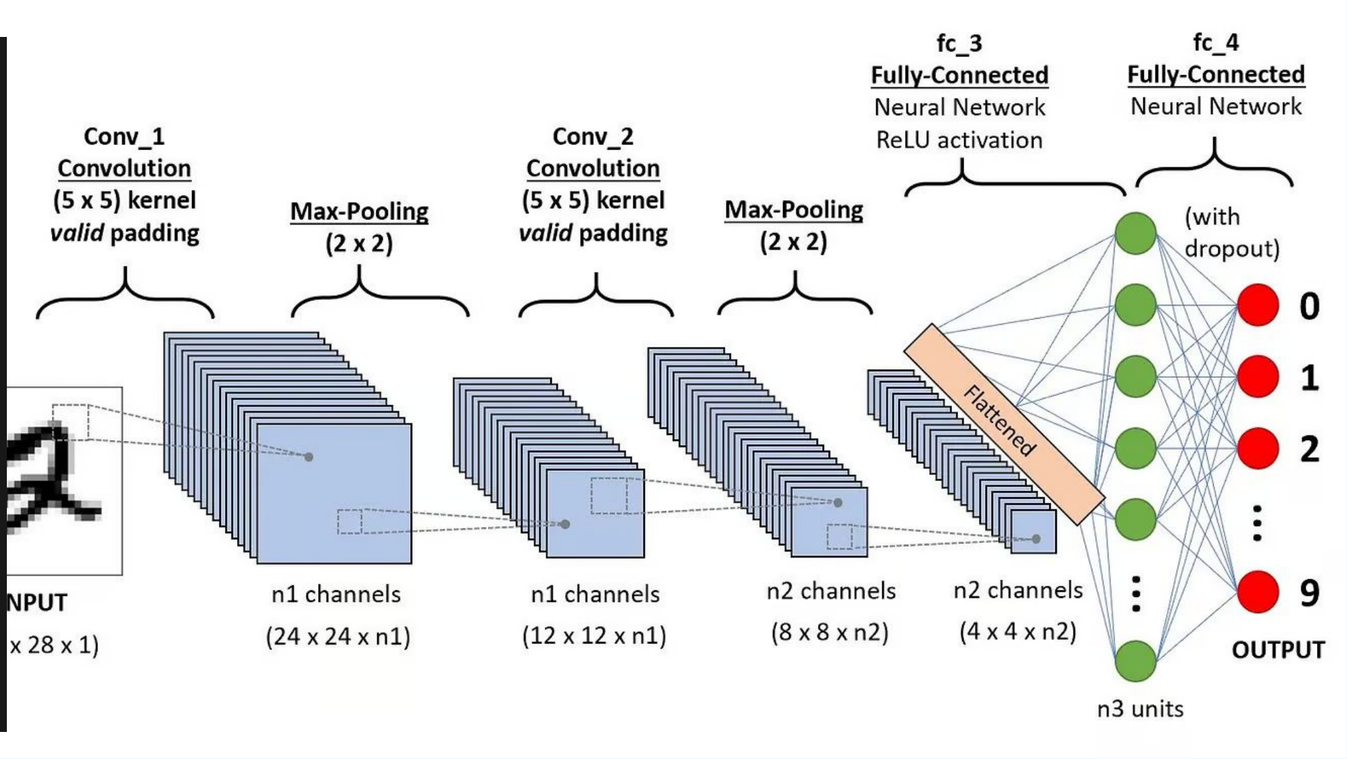
PUC Minas

Construção de uma CNN com Tensorflow/Keras

Vamos criar uma CNN simples com:

- Camadas Convolucionais (*Conv2D*) para extração de features.
- Camadas de Pooling (*MaxPooling2D*) para redução dimensional.
- Camadas Densas (*Dense*) para classificação.

```
# Flattening
model.add(
    Flatten(name='flatten')  # Achata a saída para um vetor
)
# Camada densa
model.add(
    Dense(
            units=128,  # Número de neurônios
            activation='relu',  # Função de ativação
            name='dense1',  # Nome da camada
        )
)
# Camada de saída
model.add(
    Dense(
            units=10,  # Número de classes
            activation='softmax',  # Função de ativação para classificação
            name='output_layer',  # Nome da camada
        )
)
```



PUC Minas

Treinamento e Validação do Modelo

Treinamento do modelo

```
history = model.fit(
    x_train, y_train,
    batch_size=64,
    epochs=15,
    validation_data=(x_test, y_test),
    verbose=1
)
```

Hiperparâmetros Principais e Sua Função:

- batch_size=64:
  - O que é: Número de amostras processadas antes da atualização dos pesos
  - Impacto:
    - Valores maiores (128-256): Mais estável, mas requer mais memória
    - Valores menores (32-64): Melhor generalização, mais ruído no treino
  - Por que 64: Balanceamento entre eficiência e qualidade de aprendizado para MNIST
- epochs=15:
  - O que é: Número de passagens completas pelo dataset de treino
  - Seleção:
    - MNIST geralmente converge em 10-20 épocas
- validation_data=(x_test, y_test):
  - Função: Dataset separado para avaliação durante o treino
  - Melhor prática: Nunca usar dados de teste para ajuste de hiperparâmetros
- verbose=1:
  - Controla o output durante treino (1 = barra de progresso)

Treinamento e Validação do Modelo

Plot a evolução da acurácia e perda em cada época na base de treino e validação

```
plt.figure(figsize=(12, 5))

# Gráfico de Acurácia
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Evolução da Acurácia')
plt.xlabel('Época')
plt.ylabel('Acurácia')
plt.legend()

# Gráfico de Perda
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Evolução da Função de Perda')
plt.xlabel('Época')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

- Gráfico Acurácia:
  - Treino: Aumenta suave até estabilizar
  - Validação: Segue um padrão similar ao treino
  - Observação: Diferença < 2-3%
- Gráfico Perda:
  - Treino: Diminui suave até estabilizar
  - Validação: Diminui e depois estabiliza
  - Observação: Validação deve para de melhorar quando o treino estabiliza

Avaliação de desempenho

Avaliação da Acurácia Final

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"Acurácia no Teste: {test_acc:.4f}")
print(f"Perda no Teste: {test_loss:.4f}")
```

O que está sendo feito?

- model.evaluate() calcula a loss (perda) e acurácia no conjunto de teste.
- verbose=0 evita logs desnecessários.

Avaliação de desempenho

Matriz de Confusão

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Predições no conjunto de teste
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)  # Converte probabilidades em classes (0-9)
y_true = np.argmax(y_test, axis=1)  # Labels verdadeiros

# Cria e plota a matriz de confusão
cm = confusion_matrix(y_true, y_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.xlabel("Predito")
plt.ylabel("Verdadeiro")
plt.title("Matriz de Confusão")
plt.show()
```

O que está sendo feito?

- *model.predict()* gera probabilidades para cada classe.
- *np.argmax()* seleciona a classe com maior probabilidade.
- *confusion_matrix()* compara predições (y_pred_classes) com labels reais (y_true).

PUC Minas

Avaliação de desempenho

Relatório de Classificação

```
from sklearn.metrics import classification_report

print(classification_report(y_true, y_pred_classes))
```

Saída esperada

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       0.99      0.99      0.99      1135
           2       0.98      0.99      0.99      1032
           3       0.99      0.98      0.99      1010
           4       0.99      0.98      0.99       982
           5       0.99      0.98      0.99       892
           6       0.99      0.99      0.99       958
           7       0.98      0.99      0.99      1028
           8       0.98      0.99      0.99       974
           9       0.98      0.98      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

O que cada métrica significa?

Metrica	Fórmula	Interpretação
Precision	TP / (TP + FP)	Dos que o modelo previu como X, quantos eram X de verdade?
Recall	TP / (TP + FN)	Dos que são X na verdade, quantos o modelo acertou?
F1 Score	2*(Precision*Recall) / (Precision + Recall)	Média Harmônica entre Precision e Recall
Support	-	Número de amostras por classe

Técnicas para Melhorar o desempenho da CNN

Regularização: Dropout e Weight Decay

Dropout

- O que é?
  - Técnica que "desliga" aleatoriamente neurônios durante o treinamento, evitando overfitting.
- Como usar?
  - Adicione camadas Dropout entre camadas densas ou convolucionais:

```
model.add(layers.Dropout(0.5))  # 50% dos neurônios são desativados
```

- Por que funciona?
  - Impede que a rede dependa excessivamente de neurônios específicos, promovendo robustez.

Regularização: Dropout e Weight Decay

Weight Decay (L2 Regularization)

- O que é?
  - Penaliza pesos grandes adicionando um termo à função de perda (L2 norm).
- Como usar?
  - Aplique em camadas convolucionais ou densas:

```
from tensorflow.keras.regularizers import l2

model.add(
    Conv2D(64, (3,3), activation='relu', kernel_regularizer=l2(0.01))
)
```

- Efeito
  - Reduz a complexidade do modelo, evitando overfitting.

Callbacks

Exemplo de Callbacks

```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

callbacks = [
    EarlyStopping(monitor='val_loss', patience=5),
    ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3)
]

history = model.fit(...,
                    callbacks=callbacks,
                    shuffle=True)  # Embaralha os dados a cada época
```

- **EarlyStopping:**
  - Interrompe treino se val_loss não melhorar por 'patience' épocas
  - Evita overfitting e economiza tempo
- **ReduceLROnPlateau:**
  - Reduz learning rate automaticamente quando estagnado
  - Ajuda a refinar os pesos no final do treino

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks

Ajuste de Hiperparâmetros

Taxa de aprendizado (learning rate)

A taxa de aprendizado (η) controla o tamanho do passo que o otimizador (ex: SGD, Adam) dá na direção do gradiente durante o treinamento.

- Se η for muito alto:
  - Pode causar oscilações ou divergência (a loss "explode").
  - Exemplo: η = 0.1 pode ser alto para muitas CNNs.
- Se η for muito baixo:
  - O modelo converge lentamente ou fica preso em mínimos locais.
  - Exemplo: η = 1e-5 pode exigir muitas épocas.

Estratégias de Ajuste

1. Escolha inicial
   - Valor inicial
     - Adam: η = 0.001 (default).
     - SGD: η = 0.01 ou 0.001 (com momentum).
   - Teste em escala logarítmica: [0.1, 0.01, 0.001, 0.0001].

PUC Minas

Ajuste de Hiperparâmetros

Estratégias de Ajuste

2. Learning Rate Scheduling
   - Reduz η dinamicamente durante o treinamento:

```python
from tensorflow.keras.optimizers.schedules import ExponentialDecay

# Define um decay exponencial (η = η₀ * decay_rate^(epoch / decay_steps))
lr_schedule = ExponentialDecay(
    initial_learning_rate=0.01,
    decay_steps=10000,
    decay_rate=0.9
)

optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule)
```

3. Callbacks - ReduceLROnPlateau
   - Reduz η automaticamente se a loss parar de melhorar:

```python
from tensorflow.keras.callbacks import ReduceLROnPlateau

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.1,      # Reduz η por 10x
    patience=5,      # Espera 5 épocas sem melhoria
    min_lr=1e-6      # Valor mínimo de η
)

model.fit(..., callbacks=[reduce_lr])
```

Ajuste de Hiperparâmetros

Número de Épocas (Epochs)

- Uma época corresponde a uma passagem completa pelo dataset de treino.
- Poucas épocas: Underfitting (modelo não aprende).
- Muitas épocas: Overfitting (modelo memoriza os dados).

Estratégias de Ajuste

1. Early Stopping
   - Interrompe o treinamento se o modelo não melhora:

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,    # Nº de épocas sem melhoria
    restore_best_weights=True  # Restaura os melhores pesos
)

model.fit(..., epochs=100, callbacks=[early_stopping])
```

Ajuste de Hiperparâmetros

Tamanho do Batch (Batch Size)

Batch size define quantos exemplos são processados antes de uma atualização dos pesos.

Trade-offs:

- Batch pequeno (ex: 16, 32):
  - Mais ruído (ajuda a escapar de mínimos locais).
  - Requer menos memória, mas é mais lento (mais atualizações por época).
- Batch grande (ex: 256, 512):
  - Gradientes mais estáveis, mas pode convergir para mínimos subóptimos.
  - Consome mais memória (risco de OOM Error em GPUs).

Observações Importantes

- Batch size vs. Learning Rate:
  - Se aumentar o batch size, aumente η proporcionalmente (ex: dobrar o batch $\rightarrow$ dobrar η).
  - Estudo: Accurate, Large Minibatch SGD (Facebook AI, 2017).

Aumento de dados (Data Augmentation)

É uma técnica essencial para melhorar o desempenho de CNNs, especialmente quando o dataset é pequeno. Ele cria variações artificiais dos dados de treinamento, ajudando a evitar overfitting e melhorando a generalização do modelo.

Porque usar data augmentation?

- Problema:
  - CNNs precisam de muitos dados para aprender padrões robustos.
  - Datasets pequenos levam a overfitting (modelo memoriza os dados de treino).
- Solução:
  - Gera novas amostras sintéticas a partir das imagens originais.
  - Benefícios:
    - Aumenta o tamanho efetivo do dataset.
    - Torna o modelo invariante a transformações comuns (rotação, zoom, etc.).
    - Melhora a generalização em dados não vistos (teste).

Técnica de Data Augmentation

Transformações Geométricas

Técnica	Descrição	Exemplo em Código
Rotação	Gira a imagem em um ângulo aleatório.	rotation_range=30
Translação	Desloca a imagem horizontal/verticalmente.	width_shift_range=0.2, height_shift_range=0.2
Zoom	Aplica zoom aleatório (afastamento/aproximação).	zoom_range=0.3
Flip Horizontal	Inverte a imagem horizontalmente (útil para objetos simétricos).	horizontal_flip=True
Flip Vertical	Inverte a imagem verticalmente (menos comum).	vertical_flip=True
Shear (Cisalhamento)	Distorce a imagem inclinando-a.	shear_range=0.2

Técnica de Data Augmentation

Transformações Geométricas

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=30,            # Rotação de ±30 graus
    width_shift_range=0.2,        # Translação horizontal (20% da largura)
    height_shift_range=0.2,       # Translação vertical (20% da altura)
    zoom_range=0.3,               # Zoom aleatório (70% a 130%)
    horizontal_flip=True,         # Flip horizontal
    shear_range=0.2,              # Cisalhamento
    fill_mode='nearest'           # Preenche pixels novos com valores próximos
)

# Aplica aumento de dados durante o treinamento
train_generator = datagen.flow(x_train, y_train, batch_size=32)
model.fit(train_generator, epochs=50)
```

Técnica de Data Augmentation

Transformações de Cor e Contraste

Além de transformações geométricas, ajustes de cor e iluminação podem ajudar a generalizar melhor.

Técnica	Descrição	Exemplo em Código
Brilho (Brightness)	Altera aleatoriamente o brilho da imagem.	brightness_range=[0.5, 1.5]
Canais RGB	Multiplica valores dos canais por fatores aleatórios.	channel_shift_range=50

Exemplo de código no Tensorflow/Keras

```
datagen = ImageDataGenerator(
    brightness_range=[0.7, 1.3],   # Escurece ou clareia a imagem
    channel_shift_range=50,        # Perturba canais RGB
)
```

Técnica de Data Augmentation

Boas Práticas e Dicas

- Quando Usar Data Augmentation?
  - Datasets pequenos (< 10k imagens).
  - Problemas com overfitting.
  - Objetos com variações naturais (ex: rotação em imagens médicas).
- Quando Evitar?
  - Se o dataset já é grande (pode aumentar tempo de treinamento sem benefício).
  - Se as transformações não fazem sentido (ex: flip vertical em rostos).

Validação do Aumento de Dados

Sempre visualize amostras aumentadas para garantir que as transformações fazem sentido:

```
import matplotlib.pyplot as plt

augmented_images, _ = next(train_generator)
plt.imshow(augmented_images[0].astype('uint8'))
plt.show()
```

Conclusão

Nesta aula, exploramos os **fundamentos das Redes Neurais Convolucionais (CNNs)** e sua aplicação em visão computacional, cobrindo desde conceitos teóricos até implementação prática. Os principais tópicos abordados foram:

- Fundamentos das CNNs e Sua Aplicação em Visão Computacional:
  - Compreendemos como as **CNNs revolucionaram o processamento de imagens**, superando métodos tradicionais em tarefas como classificação, detecção e segmentação.
  - Discutimos a **inspiração biológica** (córtex visual) e como as CNNs aprendem **hierarquias de características** (bordas → texturas → partes de objetos → objetos completos).
- Principais Componentes de uma CNN - Analisamos as três camadas essenciais de uma CNN:
  - **Camadas Convolucionais**
    - Filtros (kernels) que extraem características espaciais (bordas, texturas, padrões).
    - Uso de *padding* e *strides* para controlar o tamanho da saída.
  - **Camadas de Pooling** (Max Pooling, Average Pooling)
    - Redução de dimensionalidade, mantendo as características mais relevantes.
    - Aumento da eficiência computacional e redução de overfitting.
  - **Camadas Totalmente Conectadas (Dense)**
    - Combinação das features extraídas para classificação final.
- Implementação de uma CNN Básica com TensorFlow/Keras
- Avaliação e Melhoria do Desempenho