Redes Neurais Convolucionais (CNNs) e Transfer Learning

Eduardo Adame

Redes Neurais

15 de outubro de 2025

Agenda da Aula

Parte 1: CNNs

- Revisão: Limitações das MLPs
- Operação de Convolução
- Arquitetura LeNet-5
- Cálculo de dimensões
- Pooling e seus tipos

Parte 2: Transfer Learning

- Motivação e conceitos
- Fine-tuning estratégico
- Quando usar cada abordagem
- Implementação prática
- Aplicações modernas

Revisão: Por que CNNs?

Problemas das MLPs com Imagens:

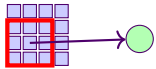
- Explosão paramétrica
  - Imagem 224×224 RGB = 150.528 entradas
  - Uma camada oculta (1000 neurônios):
  - 150 milhões de parâmetros!
- Perda de localidade espacial
- Sem invariância translacional

Solução: Convolução

- Compartilhamento de pesos
- Conectividade local
- Hierarquia de características

MLP: Conexão Total

CNN: Conexão Local



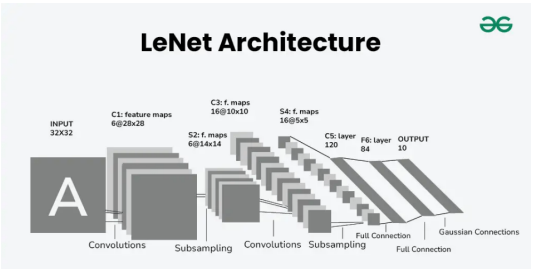
LeNet-5: Pioneira das CNNs

Criada por Yann LeCun (1998)

- Dataset MNIST: dígitos manuscritos
- 60.000 imagens treino / 10.000 teste
- Imagens 28×28 em escala de cinza
- Padding para 32×32

Inovação Principal:

- Usar convoluções para aprender características
- Redução progressiva de dimensões
- Extração hierárquica de features



Arquitetura: Conv → Pool → Conv → Pool → FC → FC → Saída

Operação de Convolução: Matemática

Definição Matemática:
Para entrada X e kernel W:

$$Y_{i,j} = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} X_{i+m,j+n} \cdot W_{m,n} + b \qquad (1)$$

onde:

- $Y_{i,j}$: pixel da saída na posição (i, j)
- k: tamanho do kernel (ex: 3×3, 5×5)
- b: bias

Cálculo de Dimensão de Saída:

$$\text{Saída} = \left\lfloor \frac{N - K + 2P}{S} \right\rfloor + 1 \qquad (2)$$

- N: dimensão entrada
- K: dimensão kernel
- P: padding
- S: stride

Operação de Convolução: Matemática

Exemplo Numérico:

Entrada 5x5, kernel 3x3, com stride=1 e padding=0:

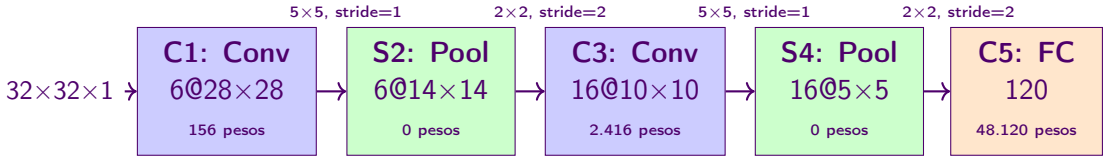
Operação de Convolução: Matemática

Exemplo Numérico:

Entrada 5x5, kernel 3x3, com stride=1 e padding=0:

$$\text{Saída} = \frac{5 - 3 + 0}{1} + 1 = 3$$

LeNet-5: Análise Camada por Camada



Cálculo de Pesos - C1:

- Kernel: $5 \times 5 \times 1 = 25$ pesos
- Bias: 1 peso
- Total por filtro: 26
- 6 filtros: $6 \times 26 = 156$

Cálculo de Pesos - C3:

- Kernel: $5 \times 5 \times 6 = 150$ pesos
- Bias: 1 peso
- Total por filtro: 151
- 16 filtros: $16 \times 151 = 2.416$

Pooling: Redução de Dimensionalidade

Objetivos do Pooling:

- Reduzir dimensões espaciais
- Diminuir parâmetros/computação
- Adicionar invariância local
- Controlar overfitting

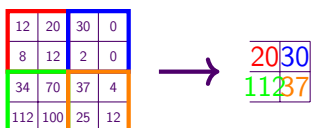
Tipos Principais:

- Max Pooling: $\max$(região)
- Average Pooling: média(região)
- Global Average Pooling (GAP)
- Stochastic Pooling

Observação: Pooling não tem pesos treináveis!

Exemplo: Max Pooling 2×2, stride=2

Entrada 4×4 Saída 2×2



Redução: 75% dos dados!

Contagem Total de Parâmetros

Camada	Cálculo	Pesos	Acumulado
Conv1 (C1)	$1 \times 6 \times 5 \times 5 + 6$	156	156
Pool1 (S2)	Sem parâmetros	0	156
Conv2 (C3)	$6 \times 16 \times 5 \times 5 + 16$	2.416	2.572
Pool2 (S4)	Sem parâmetros	0	2.572
FC1 (C5)	$400 \times 120 + 120$	48.120	50.692
FC2 (F6)	$120 \times 84 + 84$	10.164	60.856
Output	$84 \times 10 + 10$	850	61.706

Comparação com MLP

- MLP equivalente (32×32 → 120 → 84 → 10): ≈ 130.000 parâmetros
- LeNet-5: 61.706 parâmetros
- **Redução de 53%** com melhor performance!

Implementação em PyTorch


```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        # Camadas convolucionais
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)

        # Camadas totalmente conectadas
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

Implementação em PyTorch


```python
def forward(self, x):
    # C1: 32x32x1 -> 28x28x6
    x = self.pool(F.relu(self.conv1(x)))  # -> 14x14x6
    # C3: 14x14x6 -> 10x10x16
    x = self.pool(F.relu(self.conv2(x)))  # -> 5x5x16
    # Flatten
    x = x.view(-1, 16 * 5 * 5)
    # Fully connected
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

Treinamento no MNIST


```python
import torchvision
import torchvision.transforms as transforms
from torch.optim import Adam

# Preparar dados
transform = transforms.Compose([
    transforms.Pad(2),  # 28x28 -> 32x32
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

trainset = torchvision.datasets.MNIST(root='./data',
                                      train=True,
                                      download=True,
                                      transform=transform)
trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=64,
                                          shuffle=True)
```

Treinamento no MNIST


```python
# Modelo, loss e otimizador
model = LeNet5()
criterion = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters(), lr=0.001)

# Loop de treinamento
for epoch in range(10):
    for i, (images, labels) in enumerate(trainloader):
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Motivação para Transfer Learning

Por que Transfer Learning?

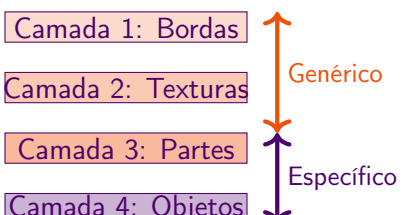
- Camadas iniciais aprendem features genéricas
  - Bordas, texturas, formas
  - Úteis para várias tarefas
- Camadas finais são específicas da tarefa
- Treinar do zero é caro:
  - ImageNet: 1.2M imagens
  - Semanas de GPU
  - Ajuste de hiperparâmetros

Vanishing Gradient:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_n} \prod_{i=2}^{n} \frac{\partial a_i}{\partial a_{i-1}}$$

Gradientes diminuem exponencialmente!

Hierarquia de Features



Estratégias de Transfer Learning

Rede Pré-treinada (ex: ResNet-50)

Estratégia 1:
Feature Extractor

Estratégia 2:
Fine-tuning

Estratégia 3:
Full training

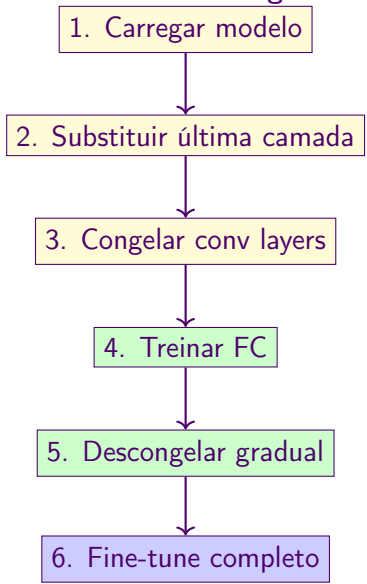
Princípios do Fine-tuning

1. **Taxa de Aprendizado Diferenciada**
   - Camadas iniciais: $lr = 10^{-5}$
   - Camadas finais: $lr = 10^{-3}$
   - Evita "catastrophic forgetting"

2. **Descongelamento Gradual**
   - Época 1-5: só treina FC
   - Época 6-10: descongela últimas convs
   - Época 11+: treina tudo

3. **Regularização**
   - Dropout mais agressivo (0.5 $\rightarrow$ 0.7)
   - Data augmentation pesada
   - Early stopping

Fluxo de Fine-tuning:



Implementação: Feature Extractor

Listing 5: CNN Simples para MNIST

```python
import torch
import torchvision.models as models

# Carregar ResNet-18 pré-treinada
resnet = models.resnet18(pretrained=True)

# Congelar todos os parâmetros
for param in resnet.parameters():
    param.requires_grad = False

# Substituir última camada (1000 -> 10 classes)
num_features = resnet.fc.in_features
resnet.fc = nn.Linear(num_features, 10)

# Apenas a última camada será treinada
trainable_params = sum(p.numel() for p in resnet.parameters()
                       if p.requires_grad)
print(f"Parâmetros treináveis: {trainable_params}")  # 5,130
```

Implementação: Feature Extractor

Listing 6: CNN Simples para MNIST

```
# Otimizador só para parâmetros treináveis
optimizer = torch.optim.Adam(resnet.fc.parameters(), lr=0.001)

# Treinar normalmente
for epoch in range(10):
    for inputs, labels in dataloader:
        outputs = resnet(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Implementação: Fine-tuning Gradual

```python
# Fine-tuning com descongelamento gradual
def unfreeze_layers(model, layer_name):
    """Descongela camadas a partir de layer_name"""
    found = False
    for name, param in model.named_parameters():
        if layer_name in name:
            found = True
        if found:
            param.requires_grad = True

# Estratégia de descongelamento
resnet = models.resnet18(pretrained=True)

# Fase 1: Apenas FC (épocas 1-5)
for param in resnet.parameters():
    param.requires_grad = False
resnet.fc = nn.Linear(resnet.fc.in_features, 10)
optimizer = Adam(resnet.fc.parameters(), lr=1e-3)
train_epochs(5)
```

Implementação: Fine-tuning Gradual


```python
# Fase 2: Descongelar layer4 (épocas 6-10)
unfreeze_layers(resnet, 'layer4')
optimizer = Adam(filter(lambda p: p.requires_grad,
                        resnet.parameters()), lr=1e-4)
train_epochs(5)

# Fase 3: Treinar tudo (épocas 11-15)
for param in resnet.parameters():
    param.requires_grad = True
optimizer = Adam(resnet.parameters(), lr=1e-5)
train_epochs(5)
```

Taxa de Aprendizado Diferenciada

```python
# Diferentes learning rates para diferentes camadas
def get_parameter_groups(model):
    """Agrupa parâmetros com diferentes LRs"""
    params = []

    # Camadas iniciais - LR muito baixo
    params.append({
        'params': model.layer1.parameters(),
        'lr': 1e-5
    })

    # Camadas intermediárias - LR baixo
    params.append({
        'params': model.layer2.parameters(),
        'lr': 5e-5
    })
    params.append({
        'params': model.layer3.parameters(),
        'lr': 1e-4
    })
```

Taxa de Aprendizado Diferenciada

```
    # Camadas finais - LR normal
    params.append({
        'params': model.layer4.parameters(),
        'lr': 5e-4
    })
    params.append({
        'params': model.fc.parameters(),
        'lr': 1e-3
    })

    return params

# Criar otimizador com grupos
param_groups = get_parameter_groups(resnet)
optimizer = Adam(param_groups)
```

Modelos Pré-treinados Populares (2025)

Modelo	Ano	Parâmetros	Top-1 Acc	Uso
AlexNet	2012	61M	56.1%	Histórico
VGG-16	2014	138M	71.5%	Segmentação
ResNet-50	2015	25M	76.1%	Padrão
MobileNet V3	2019	5.4M	75.2%	Mobile
EfficientNet-B7	2019	66M	84.3%	SOTA CNN
Vision Transformer	2020	86M	85.8%	Trending
ConvNeXt	2022	89M	87.8%	Moderna
DINOv2	2023	1B	88.3%	Self-supervised
EVA-02	2024	1B	90.1%	SOTA atual

Observações:

- ResNet-50 ainda é excelente para transfer learning
- EfficientNet oferece melhor trade-off tamanho/performance
- Vision Transformers dominam benchmarks mas precisam mais dados

Aplicações Práticas de Transfer Learning

Medicina:

- Detecção de câncer em mamografias
- COVID-19 em raios-X
- Retinopatia diabética
- Base: ImageNet $\rightarrow$ Fine-tune médico

Agricultura:

- Identificação de pragas
- Estimativa de colheita
- Análise de solo via drone
- Base: ImageNet $\rightarrow$ Dados agrícolas

Indústria:

- Detecção de defeitos
- Controle de qualidade
- Manutenção preditiva
- Base: ImageNet $\rightarrow$ Imagens industriais

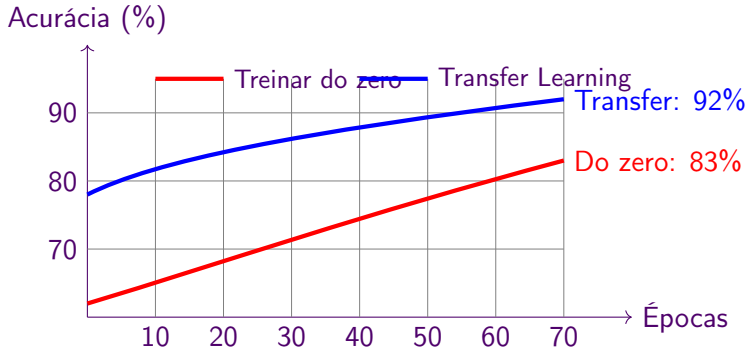
Varejo/E-commerce:

- Busca visual de produtos
- Classificação automática
- Detecção de falsificações
- Base: Fashion-MNIST $\rightarrow$ Catálogo

Sucesso do Transfer Learning

90% das aplicações de visão computacional em produção usam transfer learning!

Comparação: Treinar do Zero vs Transfer Learning



Treinar do Zero:

- Precisa ∼100k imagens
- 100+ épocas
- Convergência lenta
- GPU por dias/semanas

Transfer Learning:

- Funciona com ∼1k imagens
- 10-30 épocas
- Começa em 70-80%
- Horas de GPU

Exemplo Completo: Cães vs Gatos

Dataset:

- 25.000 imagens (Kaggle)
- 12.500 cães / 12.500 gatos
- Split: 80/10/10

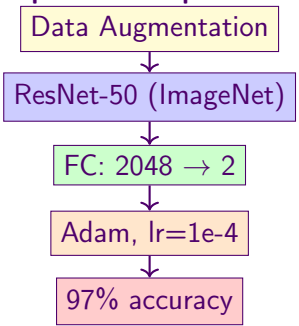
Resultados:

- Do zero: 78% (200 épocas)
- VGG-16 frozen: 92% (10 épocas)
- ResNet-50 fine-tune: 97% (25 épocas)
- EfficientNet-B0: 98.5% (20 épocas)

Tempo de treino (RTX 4090):

- Do zero: 8 horas
- Transfer: 30 minutos

Pipeline Completo:



Matriz de Confusão:

	Cão	Gato
Cão	97.2%	2.8%
Gato	3.1%	96.9%

Tendências e Futuro

Foundation Models:

- CLIP (texto + imagem)
- DINO (self-supervised)
- SAM (Segment Anything)
- Transfer learning multimodal

Efficient Transfer:

- LoRA (Low-Rank Adaptation)
- Adapter layers
- Prompt tuning visual
- Parameter-efficient fine-tuning

Novos Paradigmas:

- Zero-shot: sem fine-tuning!
- Few-shot: 5-10 exemplos
- Cross-domain: médico $\rightarrow$ industrial
- Continual learning: sem esquecer

Aplicações Emergentes:

- Visão 3D (NeRFs)
- Vídeo understanding
- Robótica embodied AI
- AR/VR em tempo real

Mensagem Final

Transfer Learning democratizou Deep Learning: qualquer um pode criar modelos state-of-the-art com recursos limitados!

Resumo da Aula

CNNs:

- Convoluções preservam localidade
- Compartilhamento de pesos
- Pooling reduz dimensões
- Hierarquia de features
- LeNet-5: 61k parâmetros

Conceitos-Chave:

- Stride e padding
- Cálculo de dimensões
- Receptive field
- Feature maps

Transfer Learning:

- Features genéricas vs específicas
- Feature extraction vs fine-tuning
- Descongelamento gradual
- Learning rate diferenciada
- 10-100x mais rápido!

Na Prática:

- Use modelos pré-treinados
- Comece com feature extraction
- Fine-tune se tiver dados
- PyTorch/TensorFlow facilitam

Próxima aula: Arquiteturas Modernas de CNNs (ResNet, Transformers)

Material Complementar e Exercícios

Leitura Recomendada:

- Paper original LeNet-5 (LeCun et al., 1998)
- "Deep Learning" - Goodfellow, Cap. 9
- CS231n Stanford - Lecture Notes
- Papers With Code - Transfer Learning

Obrigado!

Dúvidas?

Próxima aula: 22/10/2025
Tema: Arquiteturas Modernas de CNNs