PROJETO DE CURSO

SEM5952 REDES NEURAIS E APRENDIZAGEM DE MÁQUINA

2025-I

Resumo

Aplicação e Avaliação de Redes Neurais para Reconhecimento de Células Sanguíneas com a Base de Dados BloodMNIST.

Este projeto envolve a aplicação e avaliação de redes neurais para o reconhecimento de células sanguíneas periféricas usando a base de dados BloodMNIST, com 17.092 imagens coloridas classificadas em oito categorias. A primeira tarefa é implementar uma MLP com uma camada intermediária e avaliar sua acurácia e matriz de confusão. A segunda tarefa requer a construção de uma CNN simples e a avaliação da acurácia em função da quantidade e do tamanho dos kernels. Na terceira tarefa, a melhor configuração da CNN é re-treinada, e sua matriz de confusão, acurácia global e erros de classificação são analisados. Por fim, a quarta tarefa explora uma CNN mais profunda, avaliando seu desempenho e comparando os modelos estudados. A atividade exige justificativas detalhadas das escolhas feitas para garantir a reprodutibilidade da metodologia.

Oscar Daniel Veloz Segarra
NUSP: 16465899

Catalog

Descrição do projeto a resolver

Introdução

Nesta atividade, vamos abordar o problema de reconhecimento de células sanguíneas periféricas utilizando a base de dados BloodMNIST [Acevedo et al., 2020, Yang et al., 2021] (https://medmnist.com/), a qual possui 17.092 imagens microscópicas coloridas (3 canais de cor). A Figura 1 exibe uma amostra de cada classe existente na base de dados considerando a versão com resolução de 28 × 28 pixels. O mapeamento entre os identificadores das classes e os rótulos está indicado na Tabela 1.

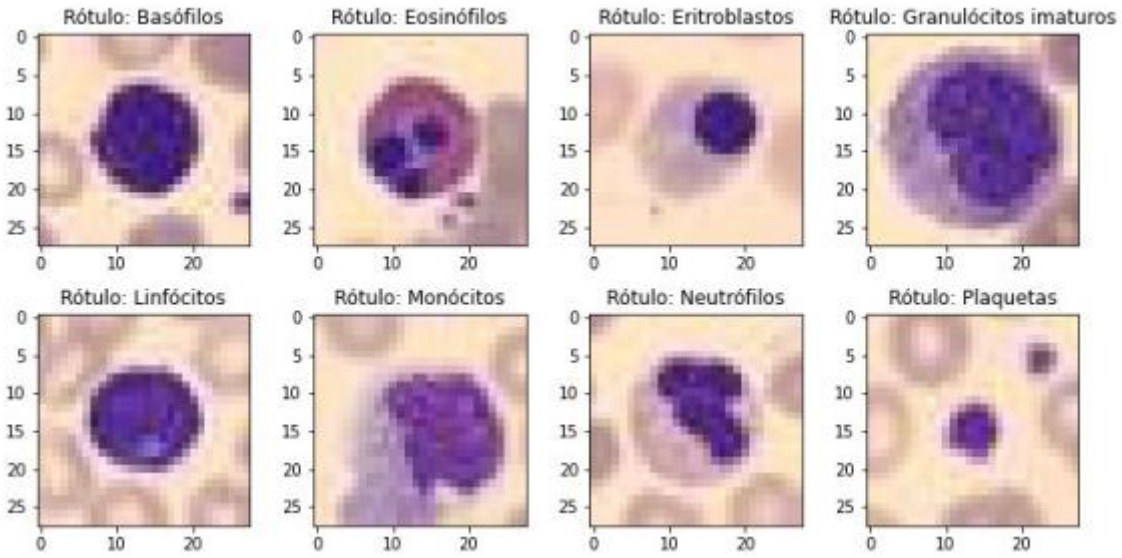


Figura 1: Amostras da base de dados BloodMNIST.

ID	Rótulo
0	Basófilos
1	Eosinófilos
2	Eritroblastos
3	Granulocitos imaturos
4	Linfocitos
5	Monocitos
6	Neutrófilos
7	Plaquetas

Tabela 1: Correspondência entre os identificadores numéricos das classes e os tipos de células sanguíneas.

Atividades

a) Aplique uma rede MLP com uma camada intermediária e análise (1) a acurácia e (2) a matriz de confusão para os dados de teste obtidas pela melhor versão desta rede. Descreva a metodologia e a arquitetura empregada, bem como todas as escolhas feitas.

b) Monte uma CNN simples contendo: (*i*) uma camada convolucional com função de ativação não-linear; (*ii*) uma camada de *pooling*; (*iii*) uma camada de saída do tipo *softmax*. Avalie a progressão da acurácia junto aos dados de validação em função ao:

- Da quantidade de *kernels* utilizados na camada convolucional;
- Do tamanho do *kernel* de convolucão.

c) Escolhendo, então, a melhor configuração para a CNN simples, refaça o treinamento do modelo e apresente:

- A matriz de confusão para os dados de teste;
- A acurácia global;
- Cinco padrões de teste que foram classificados incorretamente, indicando a classe esperada e as probabilidades estimadas pela rede.

Discuta os resultados obtidos.

d) Explorar, agora, uma CNN um pouco mais profunda. Descrever a arquitetura utilizada e apresente os mesmos resultados solicitados no item (c) para o conjunto de teste. Por fim, façer uma breve comparação entre os modelos estudados neste exercício. Pode ser interessante explorar ideias ou elementos característicos de algumas CNNs famosas (como as *ResNets* ou as *DenseNets*).

Referências

[Yang et al., 2021] J. Yang, R. Shi, D. Wei, z. Liu, L. Zhao, B. Ke, H. Pfister, B. Ni, *MedMNIST v2: A Large-Scale Lightweight Benchmark for 2D and 3D Biomedical Image Classification*. arXiv preprint arXiv:2110.14795, 2021.

[Acevedo et al., 2020] A. Acevedo, A. Merino, S. Alférez, A. Molina, L. Boldú, J. Rodellar, *A dataset of microscopic peripheral blood cell images for development of automatic recognition systems*, Data in Brief, vol. 30, 2020.

Coleta e Preparação dos Dados

Para o projeto de reconhecimento de células sanguíneas periféricas, utilizamos a base de dados **BloodMNIST**, que contém 17.092 imagens microscópicas coloridas, cada uma com resolução de 28x28 pixels. As imagens estão classificadas em oito categorias distintas de células sanguíneas. A seguir, descrevemos o processo de coleta, preparação e divisão dos dados.

Estratégia de Aquisição de Dados

Os dados foram adquiridos utilizando a biblioteca **medmnist**, que fornece uma interface para acesso e uso da base de dados **BloodMNIST**. A biblioteca automatiza o download e a verificação dos dados, de forma a garantir que todas as imagens sejam corretamente obtidas e estejam prontas para uso. Os dados foram armazenados localmente em um diretório de *cache* para facilitar o acesso subsequente sem necessidade de novos downloads.

Divisão dos Dados

A base de dados foi dividida em três conjuntos: treino, validação e teste e a seguinte proporção foi utilizada para a divisão dos dados:

Treinamento (70%): Utilizado para ajustar os parâmetros dos modelos durante o processo de aprendizado.

Validação (10%): Empregado para validar o desempenho do modelo durante o treinamento, permitindo ajustes nos hiperparâmetros.

Teste (20%): Usado para avaliar o desempenho final do modelo, representando dados novos e não vistos anteriormente.

Redimensionamento das Imagens

Com a finalidade de melhorar e garantir maior consistência e facilitar o processamento pelas redes neurais, todas as imagens foram redimensionadas para uma resolução de 32x32 pixels. Esse redimensionamento foi realizado utilizando a biblioteca PIL (Python Imaging Library), que oferece métodos para manipulação de imagens.

Resolução do item a)

Arquitetura da Rede

Camada de Entrada: Flatten para converter as imagens 2D em vetores 1D.

Camada Intermediária: Uma camada densa com 128 neurônios e ativação ReLU.

Camada de Saída: Uma camada densa com 8 neurônios e ativação *softmax* para classificar as imagens em uma das 8 classes.

Processo de Treinamento

Dataset: Utilizamos o **dataset BloodMNIST**, contendo 8 classes de tipos de células sanguíneas.

Pré-processamento: As imagens foram redimensionadas para 32x32 pixels e normalizadas.

Divisão dos Dados: Os dados foram divididos em 70% para treino, 10% para validação e 20% para teste.

Treinamento: A rede foi treinada por 50 épocas utilizando a função de perda CrossEntropyLoss e o otimizador Adam com taxa de aprendizado de 0,001.

Escolhas Feitas

Função de Ativação: ReLU na camada intermediária para introduzir não-linearidade.

Otimizador: Adam, com o propósito de alcançar eficácia e eficiência em rápida convergência.

Função de Perda: CrossEntropyLoss, adequada para problemas de classificação multi-classe.

Número de Épocas: 50 épocas para garantir que a rede tivesse tempo suficiente para aprender os padrões dos dados.

Análise dos Resultados

Os resultados obtidos pela rede neural MLP no dataset BloodMNIST são evidenciados pela matriz de confusão (Figura 1A) e pelo relatório de classificação apresentados (Figura 1B). A matriz de confusão mostra a distribuição das predições corretas e incorretas para cada uma das classes de células sanguíneas, enquanto o relatório de classificação fornece uma visão detalhada das métricas de desempenho: precisão, recall e F1-score para cada classe.

Na matriz de confusão, podemos observar que a rede neural MLP apresenta um bom desempenho geral, com a maioria das classes apresentando um alto número de predições corretas ao longo da diagonal principal. Com destaque, as classes "Eosinófilos", "Neutrófilos" e "Plaquetas" apresentam uma alta taxa de acertos, com 631, 543 e 442 predições corretas, respectivamente. Isso é corroborado pelos altos valores de precisão (0,98 para Eosinófilos, 0,94 para Neutrófilos e 0.99 para Plaquetas), recall (0,91 para Eosinófilos, 0,87 para Neutrófilos e 0,94 para Plaquetas) e F1-score (0,95 para Eosinófilos, 0,90 para Neutrófilos e 0,97 para Plaquetas), conforme indicado no relatório de classificação.

Por outro lado, a classe "Monócitos" apresenta desempenho substancialmente inferior, com uma precisão de 0,79, recall de 0,38 e F1-score de 0,52. A baixa taxa de recall indica que a rede neural está tendo dificuldades em identificar corretamente as células monócitos, frequentemente classificando-as erroneamente como outras classes, como indicado pela dispersão de predições incorretas na matriz de confusão. Este problema pode ser decorrente da similaridade visual entre monócitos e outras células ou da representação insuficiente desta classe no conjunto de dados de treinamento.

Classes como "Granulócitos" e "Linfócitos" também mostram desempenhos que podem ser melhorados, com F1-scores de 0,69 e 0,75, respectivamente. Embora os valores de recall sejam razoavelmente altos (0,82 para

Granulócitos e 0,65 para Linfócitos), a precisão para estas classes não é ideal, indicando uma quantidade considerável de falsos positivos.

A **acurácia global do modelo é de 82%**, que é uma boa métrica, porém pode ser melhorada ao abordar as classes que apresentam baixo desempenho, especialmente os Monócitos. A média macro dos F1-scores é de 0,78, indicando uma ligeira queda de desempenho quando se considera o equilíbrio entre precisão e recall de todas as classes de forma igualitária.

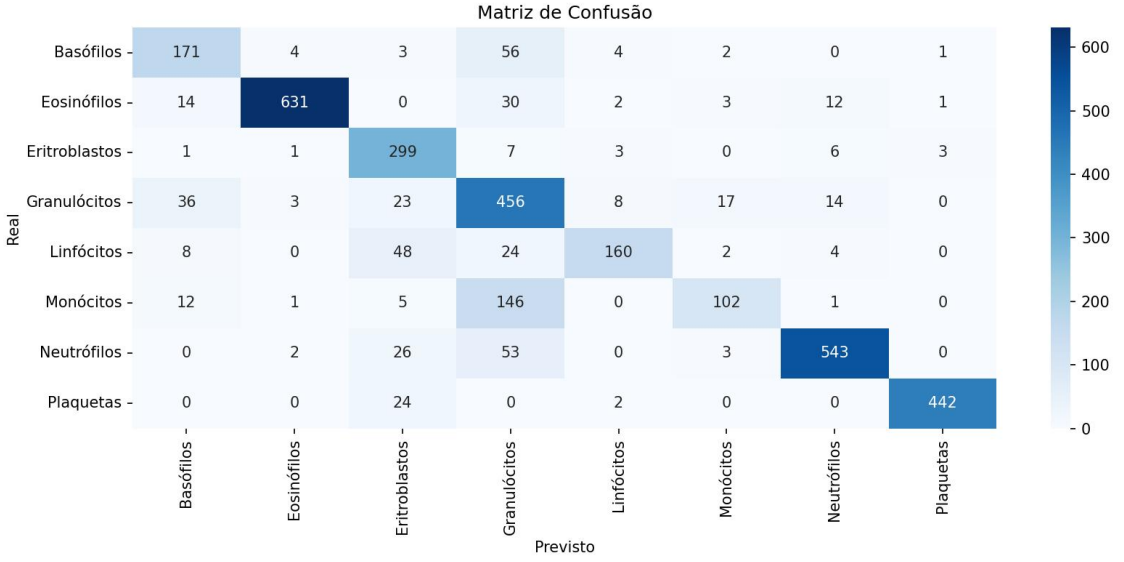


Figura 1A

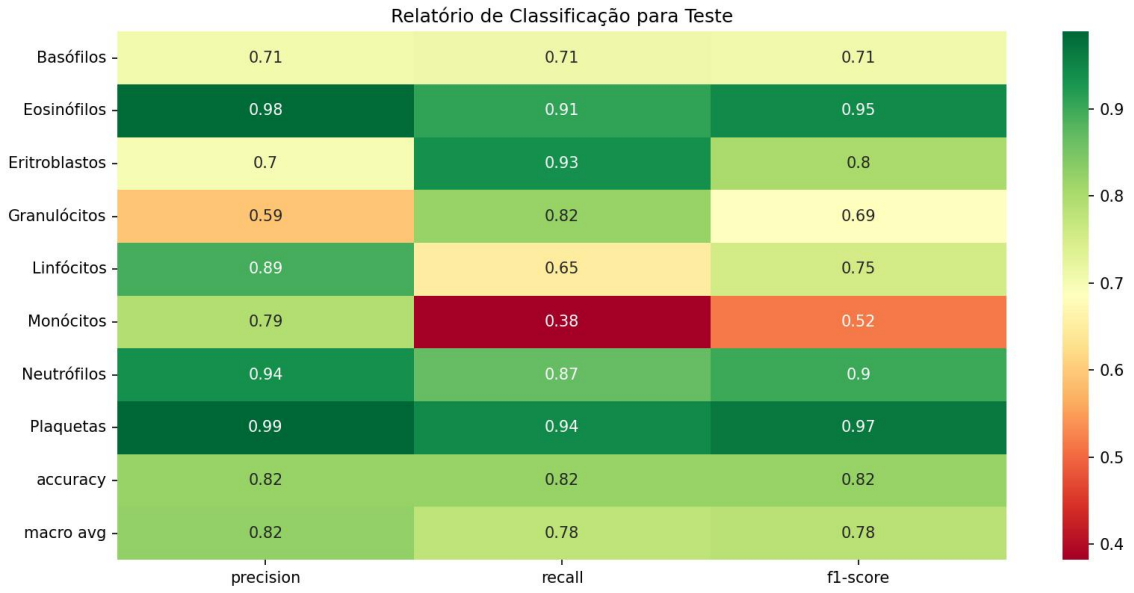


Figura 1B

Conclusão

Os resultados obtidos mostram que a rede MLP foi capaz de classificar as imagens de células sanguíneas com boa precisão. A matriz de confusão destaca áreas onde o modelo desempenha bem e onde pode ser necessário mais ajuste ou treinamento. A abordagem metodológica e as escolhas de arquitetura e parâmetros mostraram-se razoavelmente eficazes para este problema de classificação.

A rede mostra um desempenho aceitável com uma acurácia geral de 82%. Experimentamos técnicas de regularização como Dropout e Early Stopping, mas não obtivemos melhorias significativas no desempenho. Assim, optamos por manter este modelo. Observamos que há espaço para melhorias, especialmente nas classes com desempenho inferior. Técnicas adicionais, como aumento de dados e ajuste fino de hiperparâmetros, poderiam melhorar a precisão e reduzir a confusão entre as classes. No entanto, dentro das configurações atuais, este modelo representou a melhor versão que conseguimos.

Resolução do item b)

Introdução

Este item da atividade envolve a montagem de uma rede neural convolucional (CNN) simples para análise de desempenho com diferentes configurações. O modelo deve incluiu uma camada convolucional com função de ativação não-linear (ReLU), uma camada de pooling (MaxPool2d), e uma camada de saída do tipo softmax. Avaliamos a progressão da acurácia de validação em função da quantidade de kernels e do tamanho do kernel na camada convolucional

Os resultados foram obtidos utilizando o dataset BloodMNIST. A acurácia de validação foi medida para três diferentes combinações de quantidade de kernels (8, 16, 32) e tamanhos de kernel (3, 5, 7). Os gráficos gerados mostram a progressão da acurácia de validação ao longo de 50 épocas.

Análise dos resultados

8 kernels

O gráfico da Figura 2 ilustra a progressão da acurácia de validação ao longo de 50 épocas para três diferentes tamanhos de kernel (3, 5 e 7) utilizando **8 kernels** na camada convolucional.

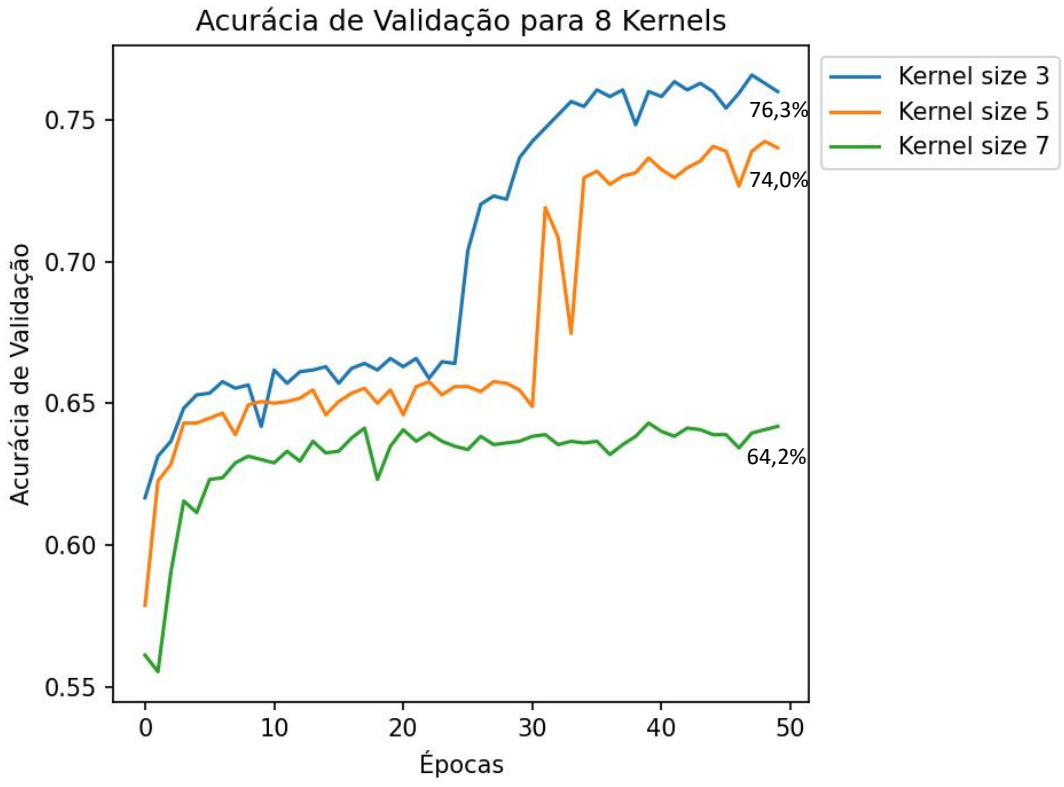


Figura 2

- *Kernel size (Azul)*
  Inicia com um desempenho próximo a 60%, com um aumento constante na acurácia até ultrapassar 75% nas últimas épocas, o que indica uma aprendizagem eficiente e estável ao longo do tempo.

- *Kernel size 5 (Laranja)*
  Apresenta a trajetória semelhante ao *size* 3, porém menos eficiente com acurácia final em 74%.

- *Kernel size 7 (Verde)*
  Atinge a estabilidade da acurácia perto da 15ª época mantendo-se próximo aos 64%.

16 kernels

Seguindo o padrão anterior, o gráfico da Figura 3 apresenta a acurácia de validação ao longo de 50 épocas para agora para 16 kernels. Aqui estão algumas observações detalhadas sobre o desempenho de cada configuração:

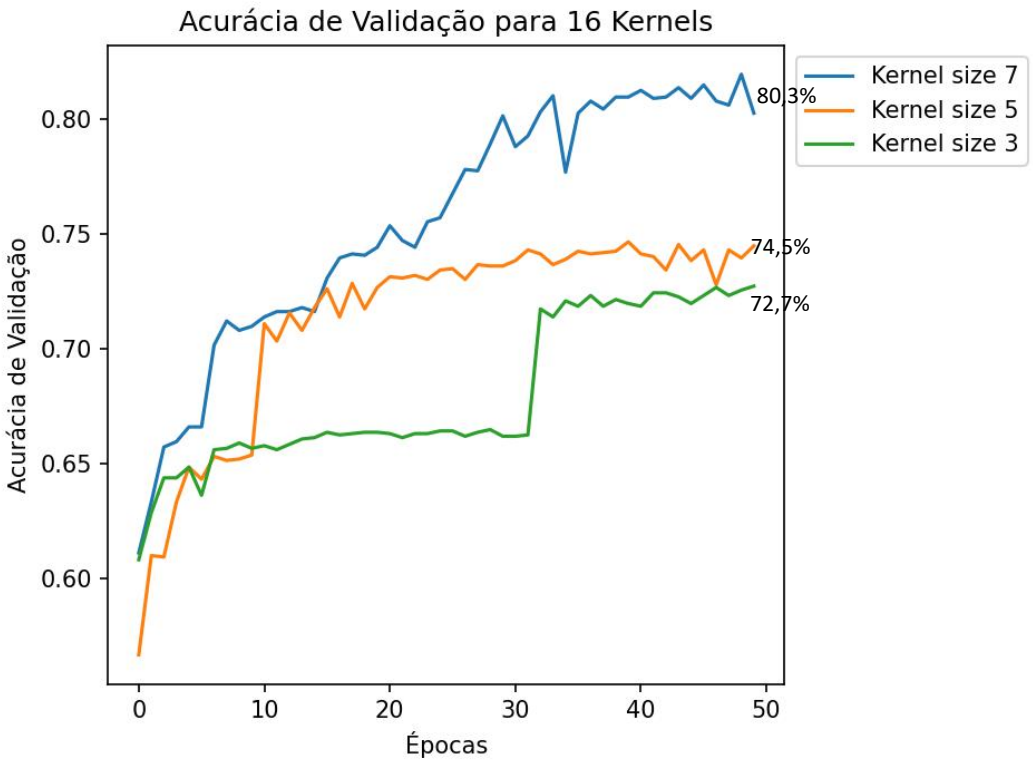


Figura 3

- *Kernel Size 3 (Verde)*
  Começa com a mesma acurácia que o tamanho 5, mas seu desempenho é o mais limitado com aumento em degrau na 30ª época e atingindo e mantendo uma acurácia em torno de 72% e finalizando na 50ª época em 72,7%.

- *Kernel Size 5 (Laranja)*
  Inicia com uma acurácia similar ao kernel de tamanho 3, mas rapidamente se distingue e segue uma trajetória de melhora constante, estabilizando-se em torno de 74,5%. Embora não atinja a eficácia do kernel de tamanho 7, mostra um equilíbrio entre desempenho e complexidade.

- *Kernel Size 7 (Azul)*
  Este kernel demonstra o melhor desempenho geral. A acurácia começa relativamente alta quando atinge a 10ª época e mostra uma tendência de aumento consistente ao longo das demais épocas,

alcançando um pico de **82,0%** na 49ª época e finalizando em 80,3%. Este comportamento indica uma boa eficiência na captura de características relevantes nos dados.

32 kernels

O gráfico da Figura 4 mostra a acurácia de validação para modelos com 32 kernels utilizando dentro dos mesmos padrões anteriores.

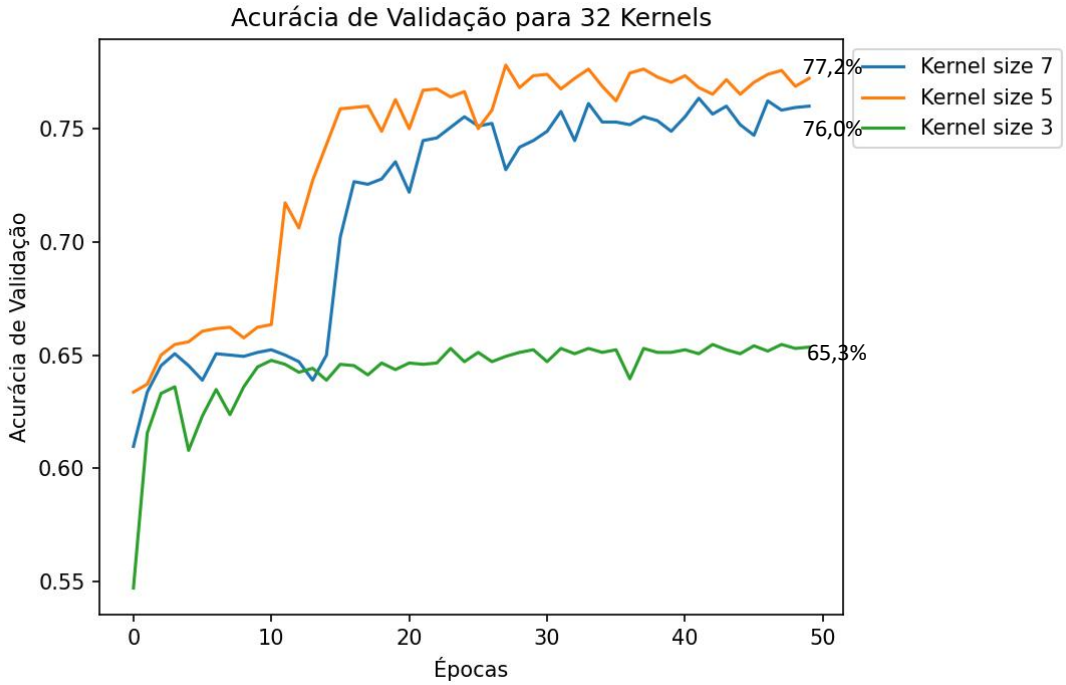


Figura 4

- *Kernel Size 7 (Azul)*
  Este tamanho de kernel demonstra uma evolução rápida em termos de acurácia, alcançando cerca de 77,2%. Apesar de algumas flutuações, ele mantém um desempenho estável a partir da 10ª época até o final.

- *Kernel Size 5 (Laranja)*
  Iniciando com uma acurácia comparável ao kernel de tamanho 7, este apresenta um crescimento até estabilizar em torno de 76%. Embora ligeiramente inferior ao kernel de tamanho 7 em termos de pico de acurácia, ele mantém um desempenho consistente a partir da 15ª época.

- *Kernel Size 3 (Verde)*
  O desempenho deste kernel é significativamente mais baixo, iniciando com acurácia em torno de 55% e alcançando um platô em torno de 65% encerrando com maior acurácia em 65,3%. Este comportamento indica uma capacidade menor para processar e aprender eficazmente as características dos dados.

Resolução do item c)

Este item propõe a escolha da melhor configuração de CNN simples entre os modelos testados no item anterior. Dentre todos os modelos, o que mais se destacou foi o com 16 kernels de tamanho 7x7, apresentando o melhor desempenho. Como pode ser observado no gráfico a seguir, a acurácia deste modelo atingiu 80% na 50ª época. Esta configuração mostrou-se mais eficaz em capturar as características distintivas das células, resultando em uma classificação mais precisa em comparação com outras configurações testadas.

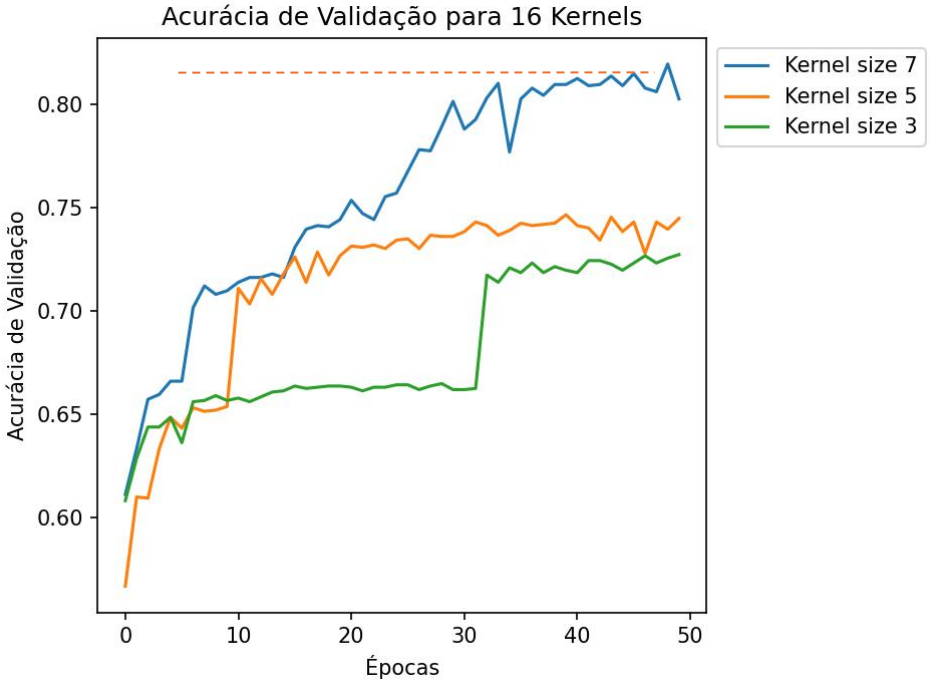


Figura 5

Refeito o treinamento para o modelo acima (agora com 70 épocas) obtivemos a seguinte matriz de confusão, com acurácia global de **87,16%**, representada na Figura 6.

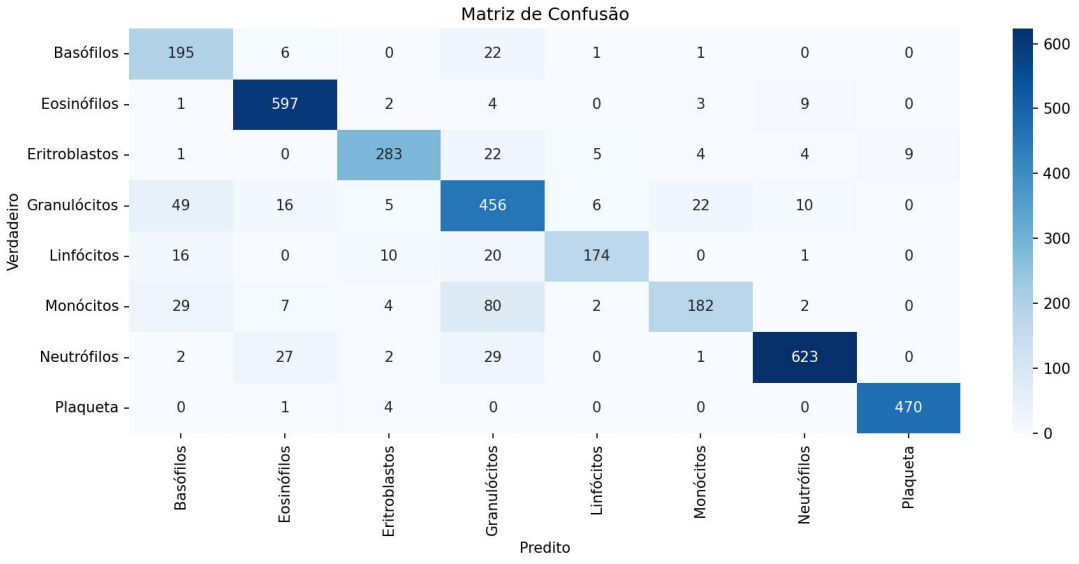


Figura 6

Observações acerca da Matriz de Confusão

A matriz de confusão e a acurácia global de **87,16%** indicam um bom desempenho geral do modelo na classificação das diferentes classes de células sanguíneas. A alta acurácia global indicam também que o modelo é eficaz em diferenciar a maioria das classes, embora ainda existam áreas de confusão entre certas classes. Isso fica evidenciado na matriz, onde podemos ver que, embora algumas classes, como Eosinófilos e Neutrófilos, sejam bem classificadas, outras ainda apresentam confusões. Esses resultados demonstram a capacidade do modelo em captar características distintivas para a maioria das classes, mas também revelam a necessidade de um exame mais detalhado das confusões específicas para entender melhor as limitações e os desafios na classificação.

Cinco padrões classificados incorretamente

Este item também propõe que sejam apresentados cinco padrões de teste que foram classificados incorretamente, indicando a classe esperada e as probabilidades. Estes padrões estão representados na Figura 7, onde "V" indica a classe verdadeira, "P" a classe predita e "Prob" a probabilidade associada à predição. A análise desses padrões revela que, apesar do modelo apresentar uma boa acurácia global, ele ainda enfrenta dificuldades em diferenciar certas classes de células, especialmente aquelas com características visuais semelhantes. Essa observação é importante para identificar áreas de melhoria no modelo de forma a permitir aprimoramentos específicos que possam aumentar ainda mais sua precisão e confiabilidade.

Figura 7

Os cinco padrões de teste apresentados que foram classificados incorretamente pelo modelo ilustram a confiança do modelo em suas predições erradas, revelando uma área importante para melhoria. A análise desses erros nos mostra que, mesmo com uma alta acurácia global, o modelo ainda enfrenta desafios significativos na diferenciação de certas classes de células. Este fenômeno sugere que há semelhanças visuais que enganam o modelo ou que pode haver um desequilíbrio nos dados de treinamento. A capacidade do modelo de fazer predições errôneas com alta confiança indica que, embora tenha aprendido a distinguir características de muitas classes com eficácia, há espaço para melhorar a discriminação entre as classes mais semelhantes. Este entendimento nos ajuda a focar em técnicas de aprimoramento, como aumento de dados específicos ou ajustes nos hiperparâmetros, para aumentar a precisão e a confiabilidade do modelo em aplicações práticas.

Resolução do item d)

Planejamento da Implementação

Iremos começar projetando uma CNN mais profunda inspirada nas ResNets, uma escolha adequada para muitas tarefas de visão computacional devido à sua capacidade de treinar redes muito profundas sem degradar o desempenho, utilizando blocos residuais.

Arquitetura Proposta para a CNN Profunda

Camada Convolutiva Inicial:

Conv2D com 32 filtros de tamanho 7x7, stride de 2, padding de 3, seguida por uma ReLU e MaxPooling de tamanho 3x3 com stride 2.

Blocos Residuais:

Vários blocos residuais que contêm duas camadas convolutivas com ativação ReLU. Cada bloco usará conexões residuais para adicionar a entrada do bloco à saída das camadas convolutivas.

Camada de Achatamento e Dense:

Flatten para transformar a saída 2D em um vetor 1D; uma camada densa com 128 unidades e ativação ReLU; camada de saída densa com 8 unidades (uma para cada classe de célula no MedMNIST).

Softmax:

A camada final Softmax para a classificação.

Matriz de Confusão: com a configuração acima, obtivemos a matriz de confusão (Figura 8ª) cuja acurácia global atingiu **93,03%** em 50 épocas.

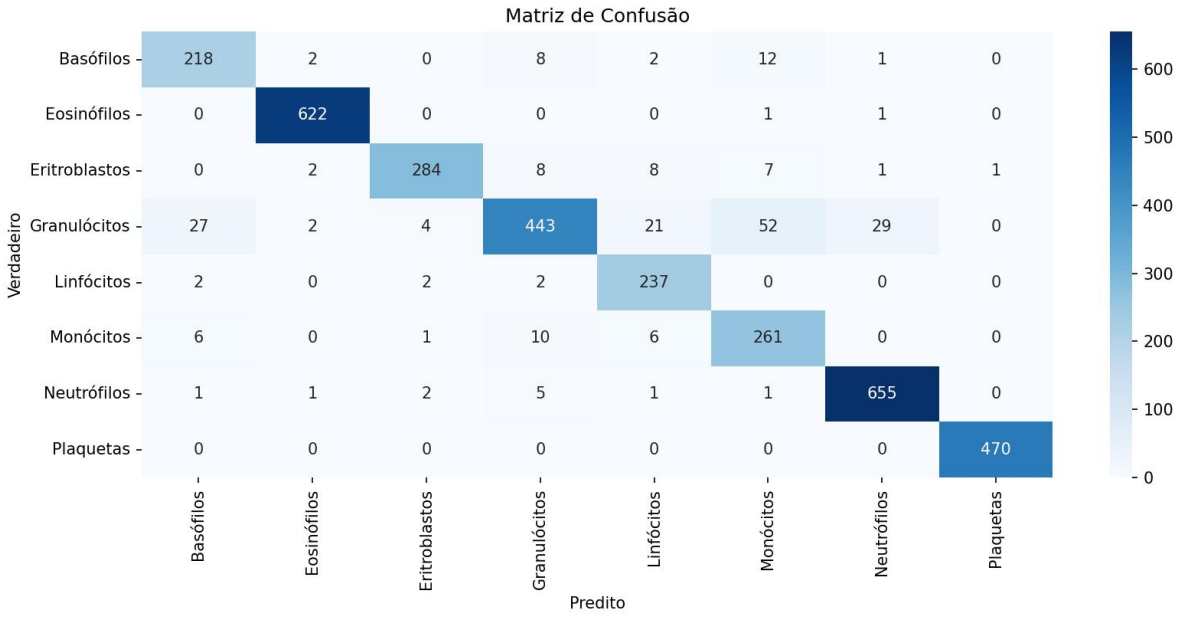


Figura 8A

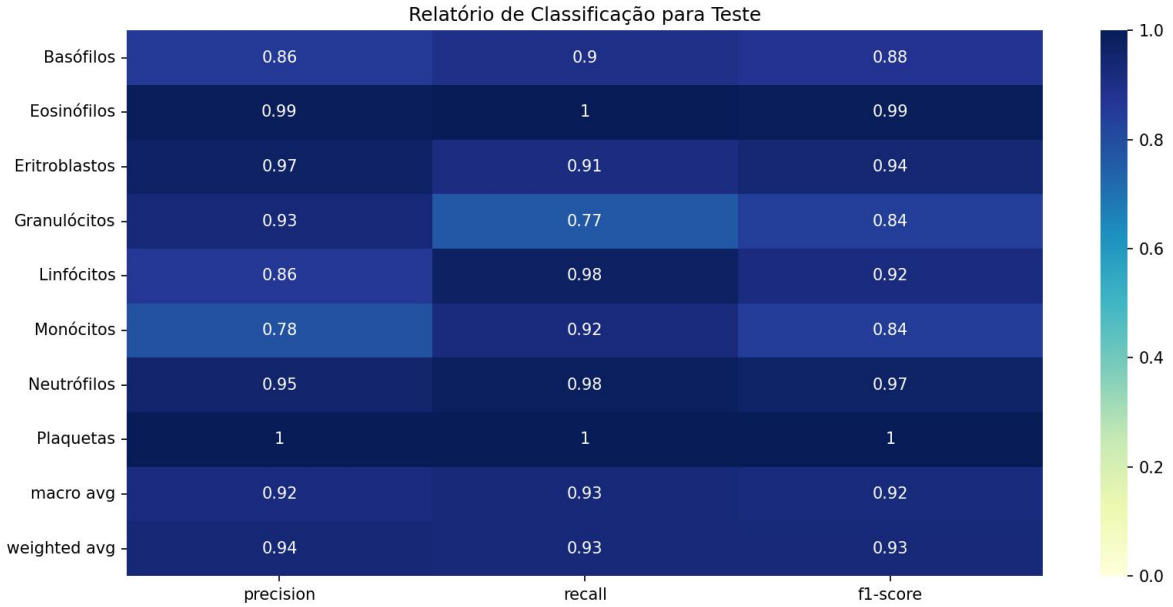


Figura 8B

Análise dos resultados

Os resultados apresentados pela rede neural MLP, baseada na arquitetura ResNet, para a classificação das imagens do dataset BloodMNIST estão evidenciados pela matriz de confusão da Figura 8A e pelo relatório de classificação na Figura 8B. A análise detalhada destes resultados revela tanto a eficácia geral do modelo quanto áreas específicas que podem necessitar de melhorias.

A matriz de confusão demonstra uma alta taxa de acertos, com a maioria das predições corretas concentradas na diagonal principal. As classes "Eosinófilos", "Neutrófilos" e "Plaquetas" exibem um desempenho excelente, com 622, 655 e 470 predições corretas, respectivamente. Este alto desempenho é corroborado pelos valores de precisão (0.99 para Eosinófilos, 0.95 para Neutrófilos e 1.0 para Plaquetas), recall (1.0 para Eosinófilos, 0.98 para Neutrófilos e 1.0 para Plaquetas) e F1-score (0.99 para Eosinófilos, 0.97 para Neutrófilos e 1.0 para Plaquetas) no relatório de classificação.

A acurácia geral do modelo é alta, refletida por um valor médio macro de precisão de 0.92, recall de 0.93 e F1-score de 0.92, indicando um desempenho equilibrado através das classes. A média ponderada das métricas reflete valores similares, com precisão de 0.94, recall de 0.93 e F1-score de 0.93, indicando que o modelo lida bem com o desbalanceamento de classes.

Em suma, a rede neural MLP baseada na ResNet mostrou-se bastante eficaz na classificação de imagens do BloodMNIST, com excelentes resultados para a maioria das classes quando comparado com as outras modelagens dos itens anteriores.

Padrões de teste

A Figura 9 mostra cinco padrões de teste que foram classificados incorretamente, indicando a classe esperada (V) a predita (P) e a probabilidade (Prob) estimada pela rede para cada uma delas.

Figura 9

Os resultados obtidos com a aplicação da ResNet, ilustrados na Figura 9, destacam desafios na discriminação de certas classes de células com características visuais semelhantes. As classificações incorretas evidenciam dificuldades específicas na identificação de padrões precisos para a diferenciação entre essas células. Uma rápida inspeção visual dos exemplos selecionados aleatoriamente revela a complexidade de distinguir entre classes que apresentam sobreposição significativa de características visuais, indicando áreas onde o modelo ainda enfrenta limitações.

Comparação entre o modelo MLP com uma camada intermediária e a arquitetura ResNet.

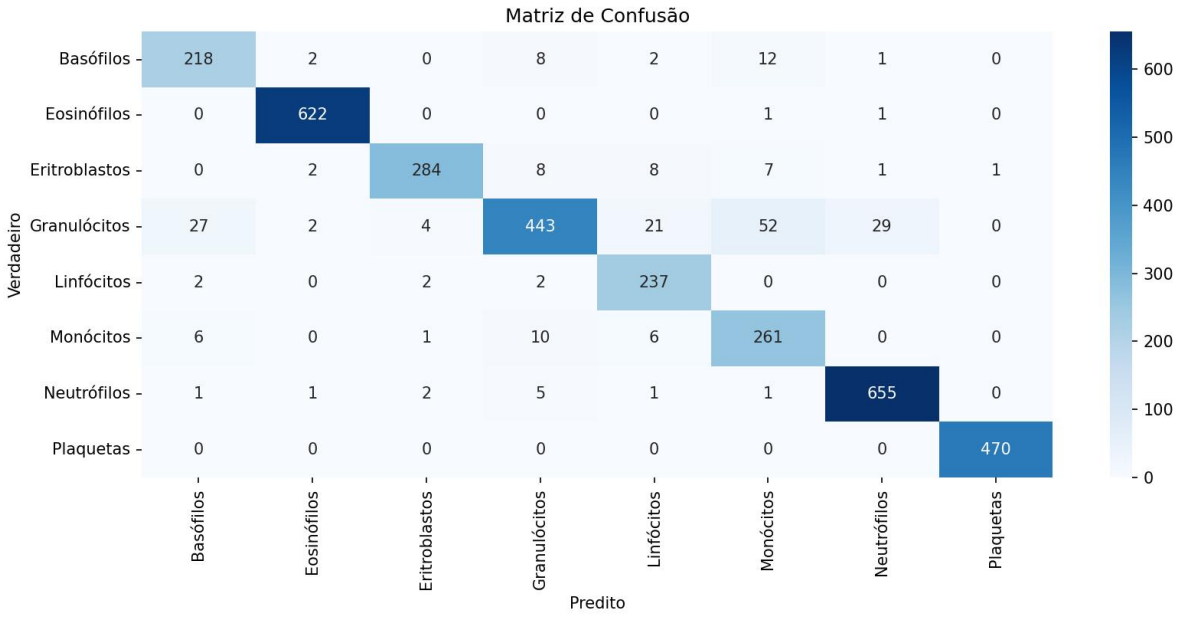


Figura 10

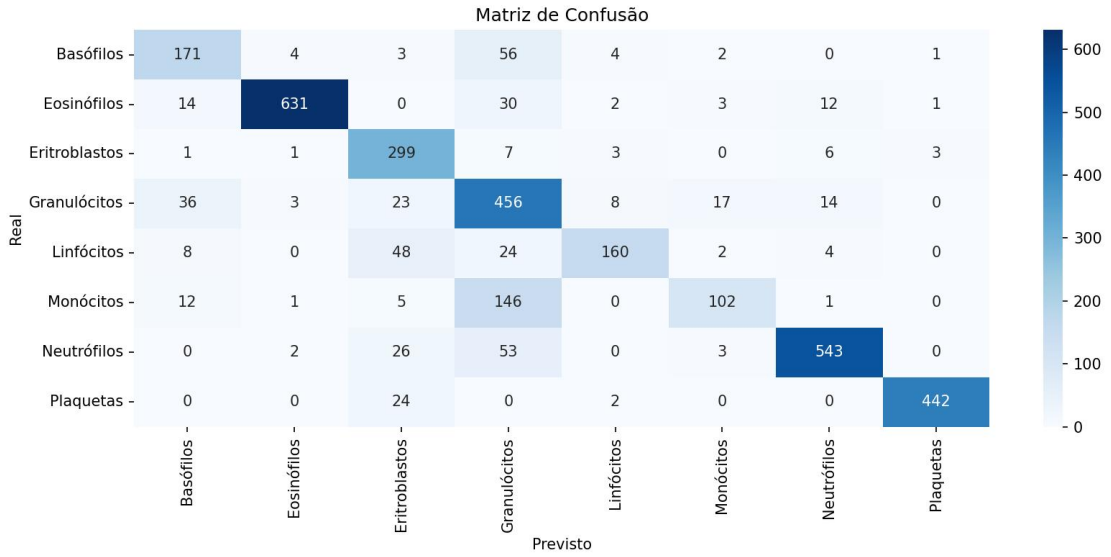


Figura 11

A comparação entre os resultados obtidos pelo modelo MLP com uma camada intermediária (Figura 10) e a arquitetura ResNet (Figura 11) revela uma clara superioridade da última em termos de desempenho na tarefa de classificação de imagens do dataset BloodMNIST. Ambos os modelos foram avaliados utilizando uma matriz de confusão e um relatório de classificação, fornecendo uma visão detalhada sobre a precisão, recall e F1-score para cada classe.

A acurácia global do modelo MLP foi de 82%, enquanto a ResNet alcançou uma acurácia significativamente maior, de 93%. Esta diferença de mais de 11 pontos percentuais na acurácia geral destaca a eficácia aprimorada da ResNet na classificação das diferentes classes de células sanguíneas.

Analisando as matrizes de confusão, observa-se que o modelo ResNet apresenta uma distribuição mais concentrada de predições corretas ao longo da diagonal principal, indicando uma maior precisão nas classificações

Em conclusão, a adoção da ResNet, uma arquitetura mais profunda e complexa, proporciona melhorias significativas em termos de precisão, recall e F1-score para quase todas as classes, resultando em uma acurácia global significativamente maior. A capacidade da ResNet de capturar características mais detalhadas e discriminativas das imagens do dataset BloodMNIST torna-a uma escolha superior para tarefas de classificação de células sanguíneas, destacando-se especialmente em classes com maior complexidade visual.

APÊNDICE

Código Python: "Carrega Dados Manipulados – V4.py"

```
import numpy as np
from PIL import Image
import pandas as pd
import medmnist
from medmnist import INFO
from sklearn.model_selection import train_test_split
import openpyxl
import os

# Use raw string para o caminho para evitar problemas com barras invertidas
# base_path = r'C:\Users\charles\PycharmProjects\IA048 - Redes Neurais'
base_path = r'D:\+USP\+MATERIAS USP\SEM5952-25i Neural Networks and Machine
Learning (Glauco Caurin)\PROJETO - ANNs. MLPs e CNNs'
```

```
def resize_images(images, new_size=(32, 32)):
    """Redimensiona imagens para o novo tamanho especificado."""
    resized_images = np.zeros((images.shape[0], new_size[0], new_size[1],
images.shape[3]), dtype=np.uint8)
    for i in range(images.shape[0]):
        img = Image.fromarray(images[i])
        img = img.resize(new_size, Image.Resampling.LANCZOS)
        resized_images[i] = np.array(img)
    return resized_images
```

```
def save_to_excel(data, filename='Tabela_Distribuicao_Dados.xlsx'):
    """Salva os dados em uma planilha Excel."""
    file_path = os.path.join(base_path, filename)
    df = pd.DataFrame(data, columns=['Conjunto', 'Número de Imagens',
'Dimensões', 'Proporção do Total'])
    df.to_excel(file_path, index=False, engine='openpyxl')
    print("Dados salvos no Excel em:", file_path)
```

```
def save_numpy_arrays(train_images, train_labels, val_images, val_labels,
test_images, test_labels):
    """Salva os arrays como arquivos .npy para uso futuro."""
    np.save(os.path.join(base_path, 'train_images.npy'), train_images)
    np.save(os.path.join(base_path, 'train_labels.npy'), train_labels)
    np.save(os.path.join(base_path, 'val_images.npy'), val_images)
    np.save(os.path.join(base_path, 'val_labels.npy'), val_labels)
    np.save(os.path.join(base_path, 'test_images.npy'), test_images)
    np.save(os.path.join(base_path, 'test_labels.npy'), test_labels)
    print("Arquivos .npy foram salvos em:", base_path)
```

```
def main():
```

```
    data_flag = 'bloodmnist'
    info = INFO[data_flag]
    DataClass = getattr(medmnist, info['python_class'])
```

```
    # Carrega todos os conjuntos de dados disponíveis, especificando onde
devem ser baixados
    train_data = DataClass(split='train', download=True, root=base_path)
    val_data = DataClass(split='val', download=True, root=base_path)
    test_data = DataClass(split='test', download=True, root=base_path)
```

```
    # Concatenação de todos os conjuntos de dados
    images = np.concatenate((train_data.imgs, val_data.imgs, test_data.imgs),
axis=0)
    labels = np.concatenate((train_data.labels, val_data.labels,
test_data.labels), axis=0)
```

```
    # Redimensionamento das imagens
    images_resized = resize_images(images, new_size=(32, 32))
```

```
    # Divisão dos dados com proporções especificadas
    train_images, test_images, train_labels, test_labels = train_test_split(
        images_resized, labels, test_size=0.2, random_state=42)  # 20% para
teste
    train_images, val_images, train_labels, val_labels = train_test_split(
        train_images, train_labels, test_size=0.125, random_state=42)  # 12.5%
de 80% é 10% do total para validação
```

```
    # Salva os conjuntos de dados divididos
    save_numpy_arrays(train_images, train_labels, val_images, val_labels,
test_images, test_labels)
```

```
    # Preparando os dados para salvar no Excel
    data = [
        ['Treino', train_images.shape[0], str(train_images.shape), f"{100 *
train_images.shape[0] / images.shape[0]:.2f}%"],
        ['Validação', val_images.shape[0], str(val_images.shape), f"{100 *
val_images.shape[0] / images.shape[0]:.2f}%"],
        ['Teste', test_images.shape[0], str(test_images.shape), f"{100 *
test_images.shape[0] / images.shape[0]:.2f}%"]
    ]
```

```
    save_to_excel(data)
```

```
if __name__ == '__main__':
    main()
```

Código Python do item a)

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report
import numpy as np
from PIL import Image
import medmnist
from medmnist import INFO
import pandas as pd
import openpyxl
import seaborn as sns
import matplotlib.pyplot as plt

class MedMNISTDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform
```

```
    def __len__(self):
        return len(self.images)
```

```
    def __getitem__(self, idx):
        img, label = self.images[idx], self.labels[idx]
        img = Image.fromarray(img)
        if self.transform:
            img = self.transform(img)
        return img, label[0]
```

```
class SimpleMLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(SimpleMLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)
```

```
    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

```
def load_and_prepare_data():
    data_flag = 'bloodmnist'
```

```
    info = INFO[data_flag]
    DataClass = getattr(medmnist, info['python_class'])
```

```
    train_data = DataClass(split='train', download=True)
    val_data = DataClass(split='val', download=True)
    test_data = DataClass(split='test', download=True)
```

```
    images = np.concatenate((train_data.imgs, val_data.imgs, test_data.imgs),
axis=0)
    labels = np.concatenate((train_data.labels, val_data.labels,
test_data.labels), axis=0)
```

```
    transform = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor()
    ])
```

```
    dataset = MedMNISTDataset(images, labels, transform=transform)
    train_size = int(0.7 * len(dataset))
    val_size = int(0.1 * len(dataset))
    test_size = len(dataset) - train_size - val_size
    train_dataset, val_dataset, test_dataset =
torch.utils.data.random_split(dataset, [train_size, val_size, test_size])
```

```
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

```
    return train_loader, val_loader, test_loader
```

```
def plot_confusion_matrix(cm, classes):
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=classes,
yticklabels=classes)
    plt.xlabel('Previsto')
    plt.ylabel('Real')
    plt.title('Matriz de Confusão')
    plt.show()
```

```
def plot_classification_report(cr, title='Relatório de Classificação',
cmap='RdYlGn'):
    df_cr = pd.DataFrame(cr).T
    sns.heatmap(df_cr.iloc[:-1, :].drop(['support'], axis=1), annot=True,
cmap=cmap)
    plt.title(title)
    plt.show()
```

```
def train_and_evaluate_model(train_loader, val_loader, test_loader):
    input_size = 32 * 32 * 3  # RGB images of 32x32
    hidden_size = 128
    num_classes = 8
```

```
    model = SimpleMLP(input_size, hidden_size, num_classes)
```

```
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
    num_epochs = 50
    for epoch in range(num_epochs):
        model.train()
        for images, labels in train_loader:
            images = images.view(images.size(0), -1)
            outputs = model(images)
            loss = criterion(outputs, labels.long())
```

```
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
    model.eval()
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for images, labels in test_loader:
            images = images.view(images.size(0), -1)
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.numpy())
            all_preds.extend(preds.numpy())
```

```
    acc = accuracy_score(all_labels, all_preds)
    cm = confusion_matrix(all_labels, all_preds)
    cr = classification_report(all_labels, all_preds,
target_names=["Basófilos", "Eosinófilos", "Eritroblastos", "Granulócitos",
"Linfócitos", "Monócitos", "Neutrófilos", "Plaquetas"], output_dict=True)
```

```
    plot_confusion_matrix(cm, classes=["Basófilos", "Eosinófilos",
"Eritroblastos", "Granulócitos", "Linfócitos", "Monócitos", "Neutrófilos",
"Plaquetas"])
    plot_classification_report(cr, title='Relatório de Classificação para
Teste')
```

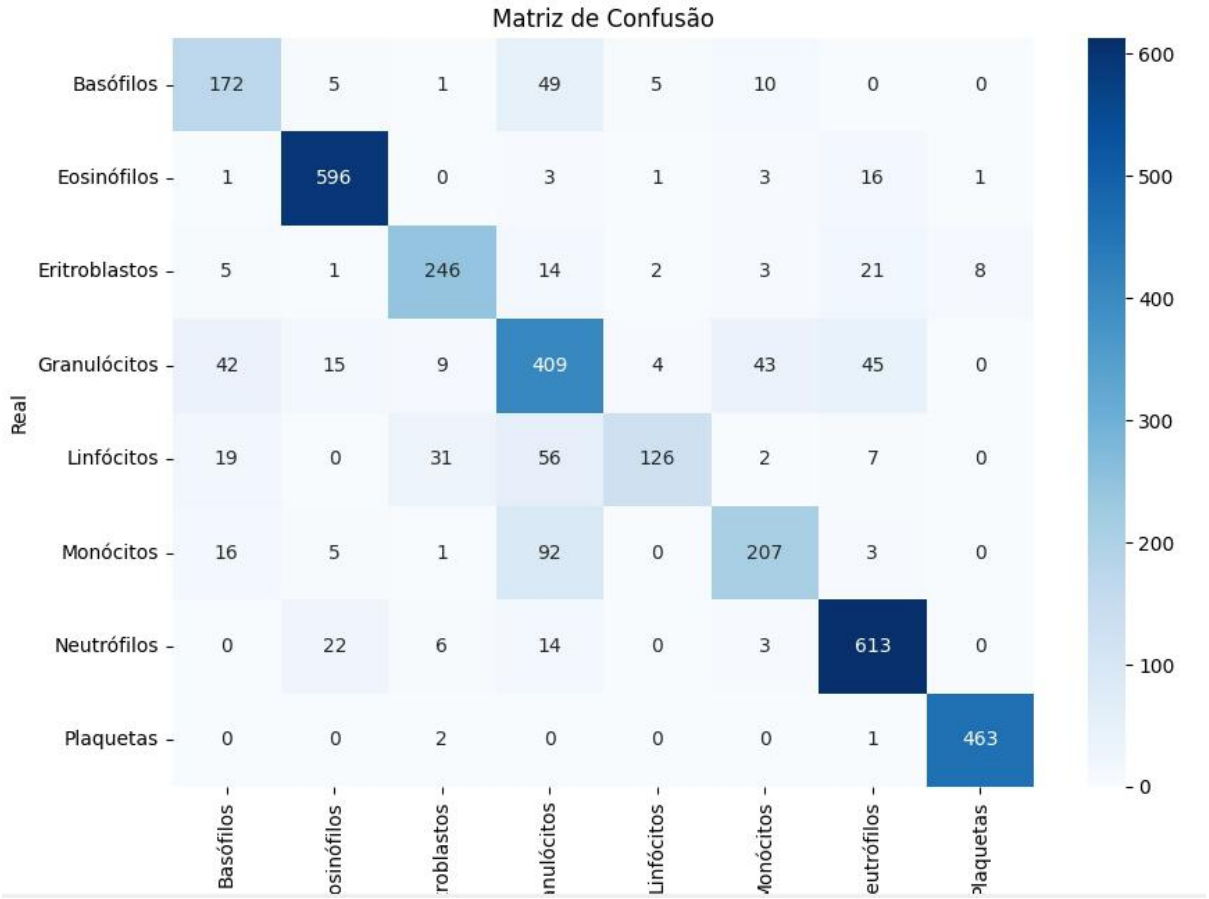
```
    return acc, cm, cr
```

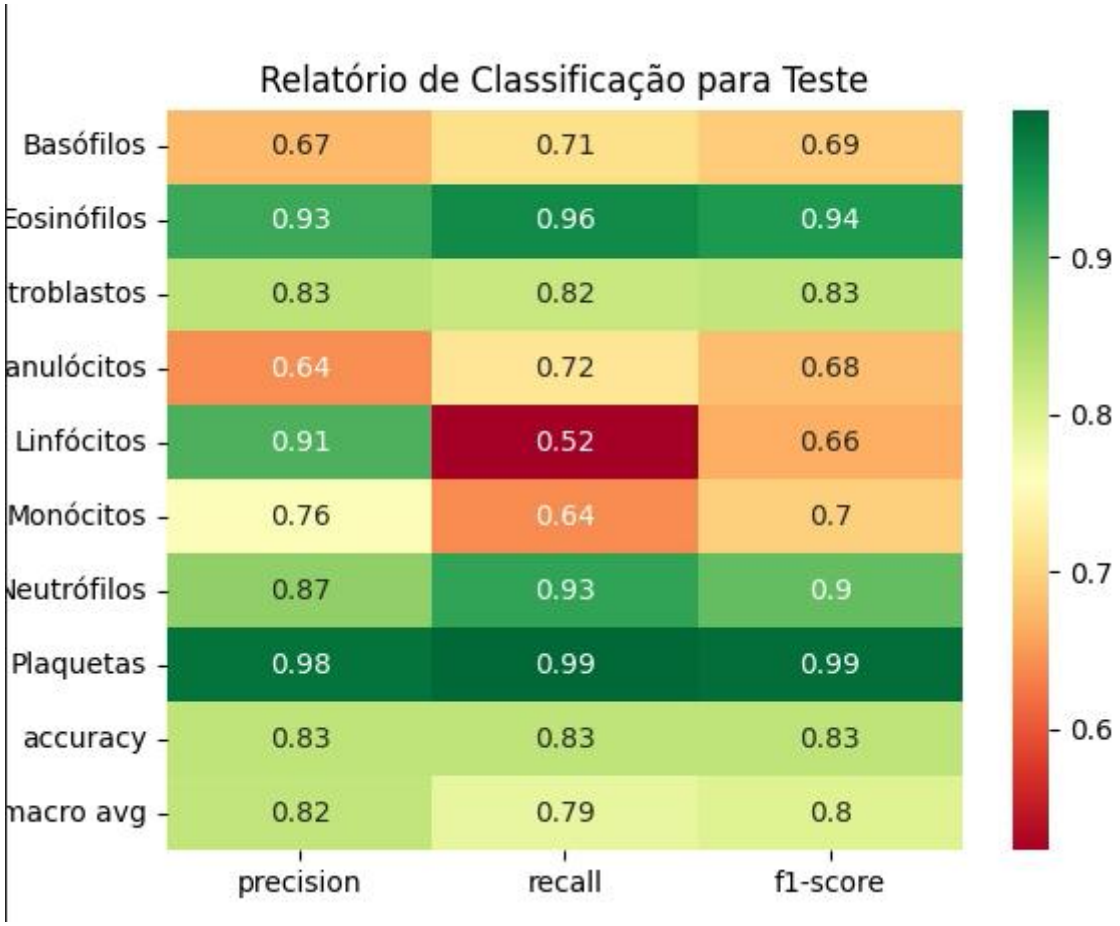
```
def main():
    train_loader, val_loader, test_loader = load_and_prepare_data()
    test_accuracy, confusion_matrix, classification_report =
train_and_evaluate_model(train_loader, val_loader, test_loader)
    print(f"Acurácia nos dados de teste: {test_accuracy * 100:.2f}%")
```

```
if __name__ == '__main__':
    main()
```

Saída do Terminal do item a)

Acurácia nos dados de teste: 82.83%




Relatório de Classificação para Teste
Basófilos
Eosinófilos
troblastos
anulócitos
Linfócitos
Monócitos
Neutrófilos
Plaquetas
accuracy
macro avg
0.67
0.71
0.69
0.93
0.96
0.94
0.83
0.82
0.83
0.64
0.72
0.68
0.91
0.52
0.66
0.76
0.64
0.7
0.87
0.93
0.9
0.98
0.99
0.99
0.83
0.83
0.83
0.82
0.79
0.8
precision
recall
f1-score
0.9
0.8
0.7
0.6

Código Python do item b)

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from sklearn.metrics import accuracy_score
import numpy as np
from PIL import Image
import medmnist
from medmnist import INFO
import pandas as pd
import matplotlib.pyplot as plt

class MedMNISTDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform
```

```
    def __len__(self):
        return len(self.images)
```

```
    def __getitem__(self, idx):
        img, label = self.images[idx], self.labels[idx]
        img = Image.fromarray(img)
        if self.transform:
            img = self.transform(img)
        return img, label[0]
```

```
class SimpleCNN(nn.Module):
    def __init__(self, num_kernels, kernel_size):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, num_kernels, kernel_size, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(num_kernels * 16 * 16, 8)
        self.softmax = nn.Softmax(dim=1)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.softmax(x)
        return x
```

```
def get_flattened_size(model, input_shape):
```

```
    with torch.no_grad():
        x = torch.zeros(1, *input_shape)
        x = model.conv1(x)
        x = model.relu(x)
        x = model.pool(x)
        x = model.flatten(x)
    return x.shape[1]
```

```
def load_and_prepare_data():
    data_flag = 'bloodmnist'
    info = INFO[data_flag]
    DataClass = getattr(medmnist, info['python_class'])
```

```
    train_data = DataClass(split='train', download=True)
    val_data = DataClass(split='val', download=True)
    test_data = DataClass(split='test', download=True)
```

```
    images = np.concatenate((train_data.imgs, val_data.imgs, test_data.imgs),
axis=0)
    labels = np.concatenate((train_data.labels, val_data.labels,
test_data.labels), axis=0)
```

```
    transform = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor()
    ])
```

```
    dataset = MedMNISTDataset(images, labels, transform=transform)
    train_size = int(0.7 * len(dataset))
    val_size = int(0.1 * len(dataset))
    test_size = len(dataset) - train_size - val_size
    train_dataset, val_dataset, test_dataset =
torch.utils.data.random_split(dataset, [train_size, val_size, test_size])
```

```
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

```
    return train_loader, val_loader, test_loader
```

```
def train_and_evaluate_cnn(num_kernels, kernel_size):
    train_loader, val_loader, test_loader = load_and_prepare_data()
    model = SimpleCNN(num_kernels, kernel_size)
```

```
    # Determinar o tamanho correto da camada linear
    input_shape = (3, 32, 32)
    flattened_size = get_flattened_size(model, input_shape)
    model.fc1 = nn.Linear(flattened_size, 8)
```

```
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
```

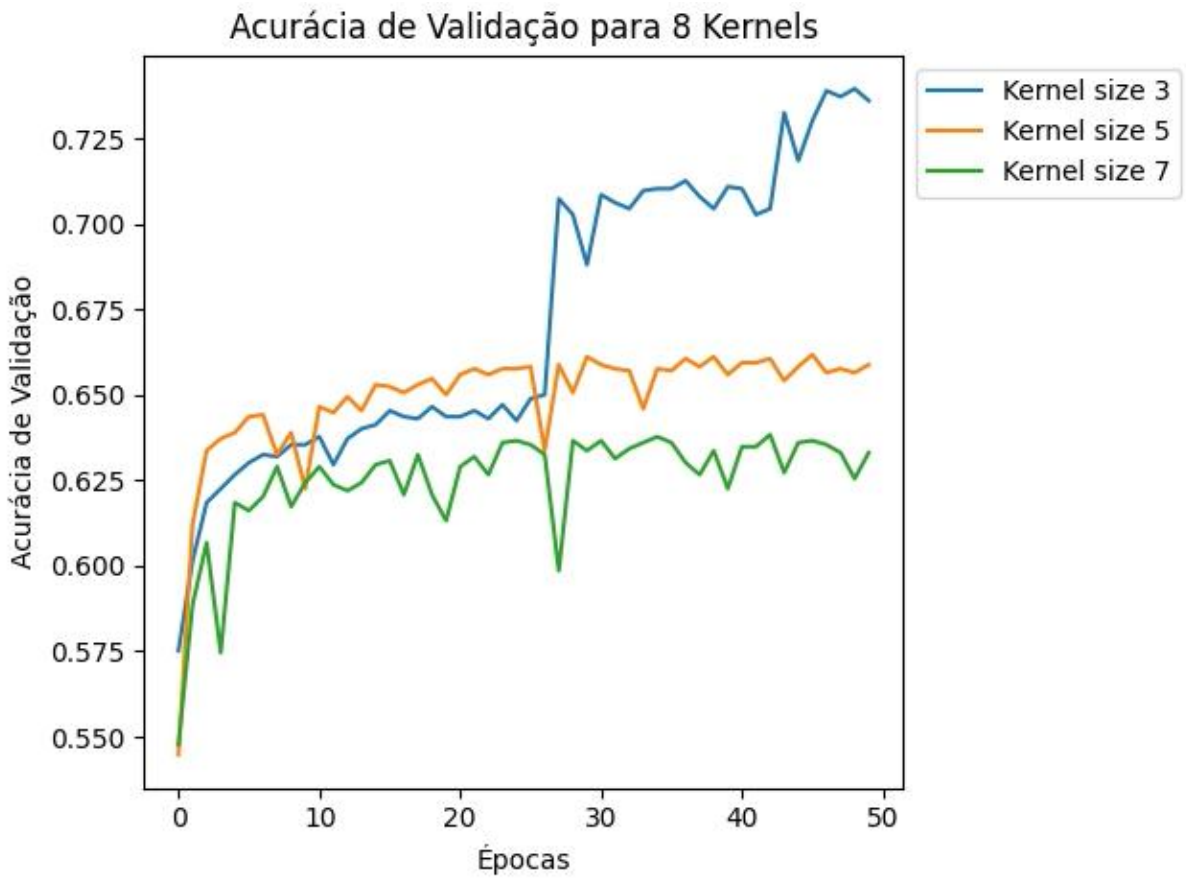
```
    num_epochs = 50
    val_accuracies = []

    for epoch in range(num_epochs):
        model.train()
        for images, labels in train_loader:
            outputs = model(images)
            loss = criterion(outputs, labels.long())

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Avaliação nos dados de validação
        model.eval()
        all_labels = []
        all_preds = []
        with torch.no_grad():
            for images, labels in val_loader:
                outputs = model(images)
                _, preds = torch.max(outputs, 1)
                all_labels.extend(labels.numpy())
                all_preds.extend(preds.numpy())

        val_acc = accuracy_score(all_labels, all_preds)
        val_accuracies.append(val_acc)
        print(f'Epoch {epoch + 1}, Val Accuracy: {val_acc:.4f}')

    return val_accuracies

def save_results_to_excel(results, filename='cnn_results.xlsx'):
    df = pd.DataFrame(results)
    df.to_excel(filename, index=False)

def main():
    num_kernels_list = [8, 16, 32]
    kernel_size_list = [3, 5, 7]
    results = []

    for num_kernels in num_kernels_list:
        fig, ax = plt.subplots()
        for kernel_size in kernel_size_list:
            print(f'\nAvaliação com {num_kernels} kernels e tamanho de kernel
{kernel_size}')
            val_accuracies = train_and_evaluate_cnn(num_kernels, kernel_size)
            ax.plot(val_accuracies, label=f'Kernel size {kernel_size}')
            results.append({
                'num_kernels': num_kernels,
                'kernel_size': kernel_size,
                'val_accuracies': val_accuracies
            })
```

```
        ax.set_xlabel('Épocas')
        ax.set_ylabel('Acurácia de Validação')
        ax.set_title(f'Acurácia de Validação para {num_kernels} Kernels')
        ax.legend(loc='upper left', bbox_to_anchor=(1, 1))
        plt.tight_layout()
        plt.show()
```

```
    save_results_to_excel(results)
```

```
if __name__ == '__main__':
    main()
```

Saída do Terminal do item b)



Avaliação com 8 kernels e tamanho de kernel 3

Epoch 1, Val Accuracy: 0.5752

Epoch 2, Val Accuracy: 0.6015

Epoch 3, Val Accuracy: 0.6185

Epoch 4, Val Accuracy: 0.6226

Epoch 5, Val Accuracy: 0.6267

Epoch 6, Val Accuracy: 0.6302

Epoch 7, Val Accuracy: 0.6325

Epoch 8, Val Accuracy: 0.6319

Epoch 9, Val Accuracy: 0.6355

Epoch 10, Val Accuracy: 0.6355

Epoch 11, Val Accuracy: 0.6378

Epoch 12, Val Accuracy: 0.6296

Epoch 13, Val Accuracy: 0.6372

Epoch 14, Val Accuracy: 0.6401

Epoch 15, Val Accuracy: 0.6413

Epoch 16, Val Accuracy: 0.6454

Epoch 17, Val Accuracy: 0.6437

Epoch 18, Val Accuracy: 0.6431

Epoch 19, Val Accuracy: 0.6466

Epoch 20, Val Accuracy: 0.6437

Epoch 21, Val Accuracy: 0.6437

Epoch 22, Val Accuracy: 0.6454

Epoch 23, Val Accuracy: 0.6431

Epoch 24, Val Accuracy: 0.6472

Epoch 25, Val Accuracy: 0.6425

Epoch 26, Val Accuracy: 0.6489

Epoch 27, Val Accuracy: 0.6501

Epoch 28, Val Accuracy: 0.7074

Epoch 29, Val Accuracy: 0.7028

Epoch 30, Val Accuracy: 0.6881

Epoch 31, Val Accuracy: 0.7086

Epoch 32, Val Accuracy: 0.7063

Epoch 33, Val Accuracy: 0.7045

Epoch 34, Val Accuracy: 0.7098

Epoch 35, Val Accuracy: 0.7104

Epoch 36, Val Accuracy: 0.7104

Epoch 37, Val Accuracy: 0.7127

Epoch 38, Val Accuracy: 0.7080

Epoch 39, Val Accuracy: 0.7045

Epoch 40, Val Accuracy: 0.7109

Epoch 41, Val Accuracy: 0.7104

Epoch 42, Val Accuracy: 0.7028

Epoch 43, Val Accuracy: 0.7045

Epoch 44, Val Accuracy: 0.7326

Epoch 45, Val Accuracy: 0.7185

Epoch 46, Val Accuracy: 0.7303

Epoch 47, Val Accuracy: 0.7390

Epoch 48, Val Accuracy: 0.7373

Epoch 49, Val Accuracy: 0.7396

Epoch 50, Val Accuracy: 0.7361

Avaliação com 8 kernels e tamanho de kernel 5

Epoch 1, Val Accuracy: 0.5448

Epoch 2, Val Accuracy: 0.6121

Epoch 3, Val Accuracy: 0.6337

Epoch 4, Val Accuracy: 0.6372

Epoch 5, Val Accuracy: 0.6390

Epoch 6, Val Accuracy: 0.6437

Epoch 7, Val Accuracy: 0.6442

Epoch 8, Val Accuracy: 0.6325

Epoch 9, Val Accuracy: 0.6390

Epoch 10, Val Accuracy: 0.6226

Epoch 11, Val Accuracy: 0.6466

Epoch 12, Val Accuracy: 0.6448

Epoch 13, Val Accuracy: 0.6495

Epoch 14, Val Accuracy: 0.6454

Epoch 15, Val Accuracy: 0.6530

Epoch 16, Val Accuracy: 0.6524

Epoch 17, Val Accuracy: 0.6507

Epoch 18, Val Accuracy: 0.6530

Epoch 19, Val Accuracy: 0.6548

Epoch 20, Val Accuracy: 0.6501

Epoch 21, Val Accuracy: 0.6559

Epoch 22, Val Accuracy: 0.6577

Epoch 23, Val Accuracy: 0.6559

Epoch 24, Val Accuracy: 0.6577

Epoch 25, Val Accuracy: 0.6577

Epoch 26, Val Accuracy: 0.6583

Epoch 27, Val Accuracy: 0.6337

Epoch 28, Val Accuracy: 0.6589

Epoch 29, Val Accuracy: 0.6507

Epoch 30, Val Accuracy: 0.6612

Epoch 31, Val Accuracy: 0.6589

Epoch 32, Val Accuracy: 0.6577

Epoch 33, Val Accuracy: 0.6571

Epoch 34, Val Accuracy: 0.6460

Epoch 35, Val Accuracy: 0.6577

Epoch 36, Val Accuracy: 0.6571

Epoch 37, Val Accuracy: 0.6606

Epoch 38, Val Accuracy: 0.6583

Epoch 39, Val Accuracy: 0.6612

Epoch 40, Val Accuracy: 0.6559

Epoch 41, Val Accuracy: 0.6594

Epoch 42, Val Accuracy: 0.6594

Epoch 43, Val Accuracy: 0.6606

Epoch 44, Val Accuracy: 0.6542

Epoch 45, Val Accuracy: 0.6583

Epoch 46, Val Accuracy: 0.6618

Epoch 47, Val Accuracy: 0.6565

Epoch 48, Val Accuracy: 0.6577

Epoch 49, Val Accuracy: 0.6565

Epoch 50, Val Accuracy: 0.6589

Avaliação com 8 kernels e tamanho de kernel 7

Epoch 1, Val Accuracy: 0.5477

Epoch 2, Val Accuracy: 0.5886

Epoch 3, Val Accuracy: 0.6068

Epoch 4, Val Accuracy: 0.5746

Epoch 5, Val Accuracy: 0.6185

Epoch 6, Val Accuracy: 0.6161

Epoch 7, Val Accuracy: 0.6202

Epoch 8, Val Accuracy: 0.6290

Epoch 9, Val Accuracy: 0.6173

Epoch 10, Val Accuracy: 0.6243

Epoch 11, Val Accuracy: 0.6290

Epoch 12, Val Accuracy: 0.6238

Epoch 13, Val Accuracy: 0.6220

Epoch 14, Val Accuracy: 0.6243

Epoch 15, Val Accuracy: 0.6296

Epoch 16, Val Accuracy: 0.6308

Epoch 17, Val Accuracy: 0.6208

Epoch 18, Val Accuracy: 0.6325

Epoch 19, Val Accuracy: 0.6208

Epoch 20, Val Accuracy: 0.6132

Epoch 21, Val Accuracy: 0.6290

Epoch 22, Val Accuracy: 0.6319

Epoch 23, Val Accuracy: 0.6267

Epoch 24, Val Accuracy: 0.6360

Epoch 25, Val Accuracy: 0.6366

Epoch 26, Val Accuracy: 0.6355

Epoch 27, Val Accuracy: 0.6325

Epoch 28, Val Accuracy: 0.5986

Epoch 29, Val Accuracy: 0.6366

Epoch 30, Val Accuracy: 0.6337

Epoch 31, Val Accuracy: 0.6366

Epoch 32, Val Accuracy: 0.6314

Epoch 33, Val Accuracy: 0.6343

Epoch 34, Val Accuracy: 0.6360

Epoch 35, Val Accuracy: 0.6378

Epoch 36, Val Accuracy: 0.6360

Epoch 37, Val Accuracy: 0.6302

Epoch 38, Val Accuracy: 0.6267

Epoch 39, Val Accuracy: 0.6337

Epoch 40, Val Accuracy: 0.6226

Epoch 41, Val Accuracy: 0.6349

Epoch 42, Val Accuracy: 0.6349

Epoch 43, Val Accuracy: 0.6384

Epoch 44, Val Accuracy: 0.6273

Epoch 45, Val Accuracy: 0.6360

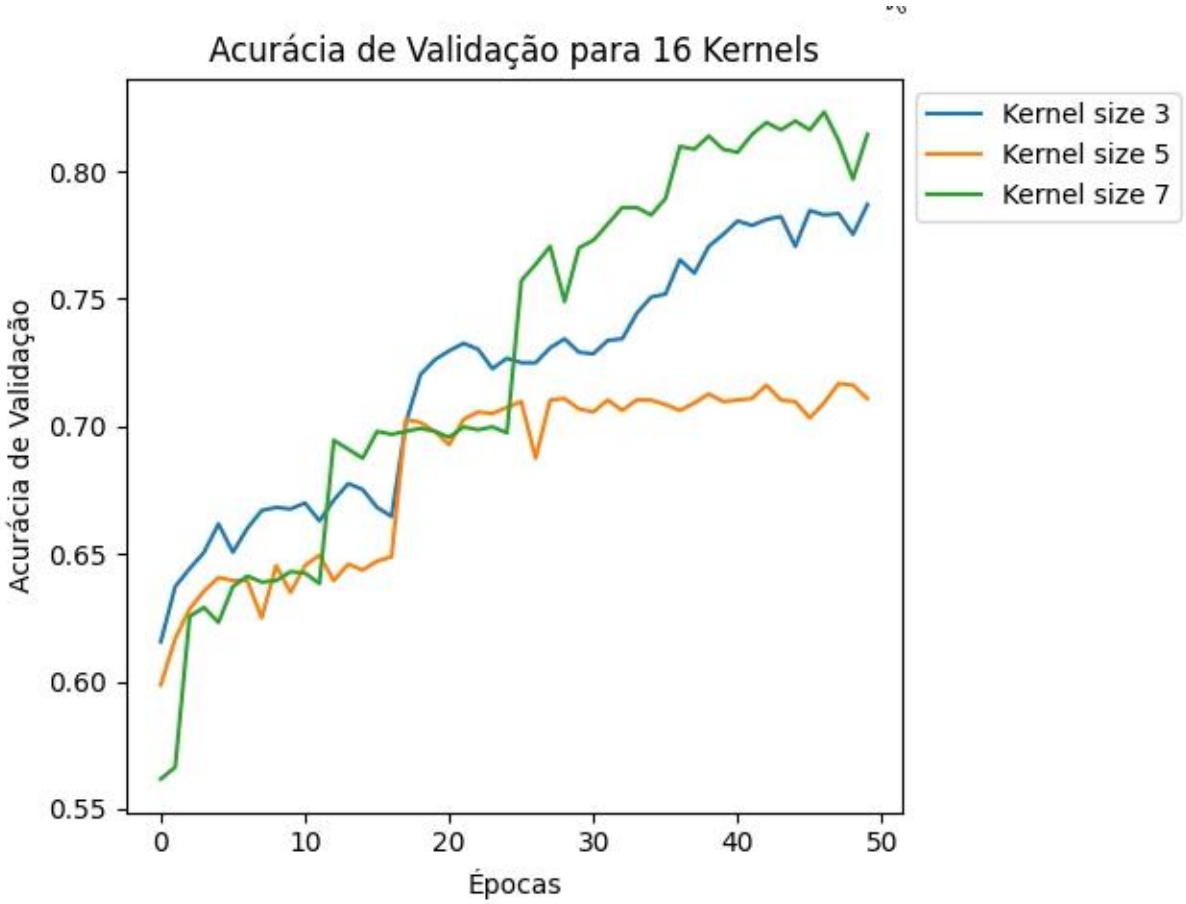
Epoch 46, Val Accuracy: 0.6366

Epoch 47, Val Accuracy: 0.6355

Epoch 48, Val Accuracy: 0.6331

Epoch 49, Val Accuracy: 0.6255

Epoch 50, Val Accuracy: 0.6331



Avaliação com 16 kernels e tamanho de kernel 3

Epoch 1, Val Accuracy: 0.6156

Epoch 2, Val Accuracy: 0.6372

Epoch 3, Val Accuracy: 0.6442

Epoch 4, Val Accuracy: 0.6507

Epoch 5, Val Accuracy: 0.6618

Epoch 6, Val Accuracy: 0.6507

Epoch 7, Val Accuracy: 0.6600

Epoch 8, Val Accuracy: 0.6671

Epoch 9, Val Accuracy: 0.6682

Epoch 10, Val Accuracy: 0.6676

Epoch 11, Val Accuracy: 0.6700

Epoch 12, Val Accuracy: 0.6630

Epoch 13, Val Accuracy: 0.6712

Epoch 14, Val Accuracy: 0.6776

Epoch 15, Val Accuracy: 0.6752

Epoch 16, Val Accuracy: 0.6682

Epoch 17, Val Accuracy: 0.6647

Epoch 18, Val Accuracy: 0.7016

Epoch 19, Val Accuracy: 0.7203

Epoch 20, Val Accuracy: 0.7262

Epoch 21, Val Accuracy: 0.7297

Epoch 22, Val Accuracy: 0.7326

Epoch 23, Val Accuracy: 0.7303

Epoch 24, Val Accuracy: 0.7226

Epoch 25, Val Accuracy: 0.7267

Epoch 26, Val Accuracy: 0.7250

Epoch 27, Val Accuracy: 0.7250

Epoch 28, Val Accuracy: 0.7308

Epoch 29, Val Accuracy: 0.7343

Epoch 30, Val Accuracy: 0.7291

Epoch 31, Val Accuracy: 0.7285

Epoch 32, Val Accuracy: 0.7338

Epoch 33, Val Accuracy: 0.7343

Epoch 34, Val Accuracy: 0.7443

Epoch 35, Val Accuracy: 0.7507

Epoch 36, Val Accuracy: 0.7519

Epoch 37, Val Accuracy: 0.7654

Epoch 38, Val Accuracy: 0.7601

Epoch 39, Val Accuracy: 0.7706

Epoch 40, Val Accuracy: 0.7753

Epoch 41, Val Accuracy: 0.7806

Epoch 42, Val Accuracy: 0.7788

Epoch 43, Val Accuracy: 0.7812

Epoch 44, Val Accuracy: 0.7823

Epoch 45, Val Accuracy: 0.7706

Epoch 46, Val Accuracy: 0.7847

Epoch 47, Val Accuracy: 0.7829

Epoch 48, Val Accuracy: 0.7835

Epoch 49, Val Accuracy: 0.7753

Epoch 50, Val Accuracy: 0.7870

Avaliação com 16 kernels e tamanho de kernel 5

Epoch 1, Val Accuracy: 0.5986

Epoch 2, Val Accuracy: 0.6167

Epoch 3, Val Accuracy: 0.6284

Epoch 4, Val Accuracy: 0.6355

Epoch 5, Val Accuracy: 0.6407

Epoch 6, Val Accuracy: 0.6396

Epoch 7, Val Accuracy: 0.6396

Epoch 8, Val Accuracy: 0.6249

Epoch 9, Val Accuracy: 0.6454

Epoch 10, Val Accuracy: 0.6349

Epoch 11, Val Accuracy: 0.6454

Epoch 12, Val Accuracy: 0.6495

Epoch 13, Val Accuracy: 0.6396

Epoch 14, Val Accuracy: 0.6460

Epoch 15, Val Accuracy: 0.6437

Epoch 16, Val Accuracy: 0.6472

Epoch 17, Val Accuracy: 0.6489

Epoch 18, Val Accuracy: 0.7028

Epoch 19, Val Accuracy: 0.7016

Epoch 20, Val Accuracy: 0.6981

Epoch 21, Val Accuracy: 0.6928

Epoch 22, Val Accuracy: 0.7028

Epoch 23, Val Accuracy: 0.7057

Epoch 24, Val Accuracy: 0.7051

Epoch 25, Val Accuracy: 0.7074

Epoch 26, Val Accuracy: 0.7098

Epoch 27, Val Accuracy: 0.6875

Epoch 28, Val Accuracy: 0.7104

Epoch 29, Val Accuracy: 0.7109

Epoch 30, Val Accuracy: 0.7068

Epoch 31, Val Accuracy: 0.7057

Epoch 32, Val Accuracy: 0.7104

Epoch 33, Val Accuracy: 0.7063

Epoch 34, Val Accuracy: 0.7104

Epoch 35, Val Accuracy: 0.7104

Epoch 36, Val Accuracy: 0.7086

Epoch 37, Val Accuracy: 0.7063

Epoch 38, Val Accuracy: 0.7092

Epoch 39, Val Accuracy: 0.7127

Epoch 40, Val Accuracy: 0.7098

Epoch 41, Val Accuracy: 0.7104

Epoch 42, Val Accuracy: 0.7109

Epoch 43, Val Accuracy: 0.7162

Epoch 44, Val Accuracy: 0.7104

Epoch 45, Val Accuracy: 0.7098

Epoch 46, Val Accuracy: 0.7033

Epoch 47, Val Accuracy: 0.7092

Epoch 48, Val Accuracy: 0.7168

Epoch 49, Val Accuracy: 0.7162

Epoch 50, Val Accuracy: 0.7109

Avaliação com 16 kernels e tamanho de kernel 7

Epoch 1, Val Accuracy: 0.5617

Epoch 2, Val Accuracy: 0.5664

Epoch 3, Val Accuracy: 0.6255

Epoch 4, Val Accuracy: 0.6290

Epoch 5, Val Accuracy: 0.6232

Epoch 6, Val Accuracy: 0.6372

Epoch 7, Val Accuracy: 0.6413

Epoch 8, Val Accuracy: 0.6390

Epoch 9, Val Accuracy: 0.6396

Epoch 10, Val Accuracy: 0.6431

Epoch 11, Val Accuracy: 0.6425

Epoch 12, Val Accuracy: 0.6384

Epoch 13, Val Accuracy: 0.6946

Epoch 14, Val Accuracy: 0.6910

Epoch 15, Val Accuracy: 0.6875

Epoch 16, Val Accuracy: 0.6981

Epoch 17, Val Accuracy: 0.6969

Epoch 18, Val Accuracy: 0.6981

Epoch 19, Val Accuracy: 0.6992

Epoch 20, Val Accuracy: 0.6981

Epoch 21, Val Accuracy: 0.6957

Epoch 22, Val Accuracy: 0.6998

Epoch 23, Val Accuracy: 0.6987

Epoch 24, Val Accuracy: 0.6998

Epoch 25, Val Accuracy: 0.6975

Epoch 26, Val Accuracy: 0.7572

Epoch 27, Val Accuracy: 0.7636

Epoch 28, Val Accuracy: 0.7706

Epoch 29, Val Accuracy: 0.7490

Epoch 30, Val Accuracy: 0.7700

Epoch 31, Val Accuracy: 0.7730

Epoch 32, Val Accuracy: 0.7794

Epoch 33, Val Accuracy: 0.7858

Epoch 34, Val Accuracy: 0.7858

Epoch 35, Val Accuracy: 0.7829

Epoch 36, Val Accuracy: 0.7894

Epoch 37, Val Accuracy: 0.8098

Epoch 38, Val Accuracy: 0.8087

Epoch 39, Val Accuracy: 0.8139

Epoch 40, Val Accuracy: 0.8087

Epoch 41, Val Accuracy: 0.8075

Epoch 42, Val Accuracy: 0.8145

Epoch 43, Val Accuracy: 0.8192

Epoch 44, Val Accuracy: 0.8163

Epoch 45, Val Accuracy: 0.8198

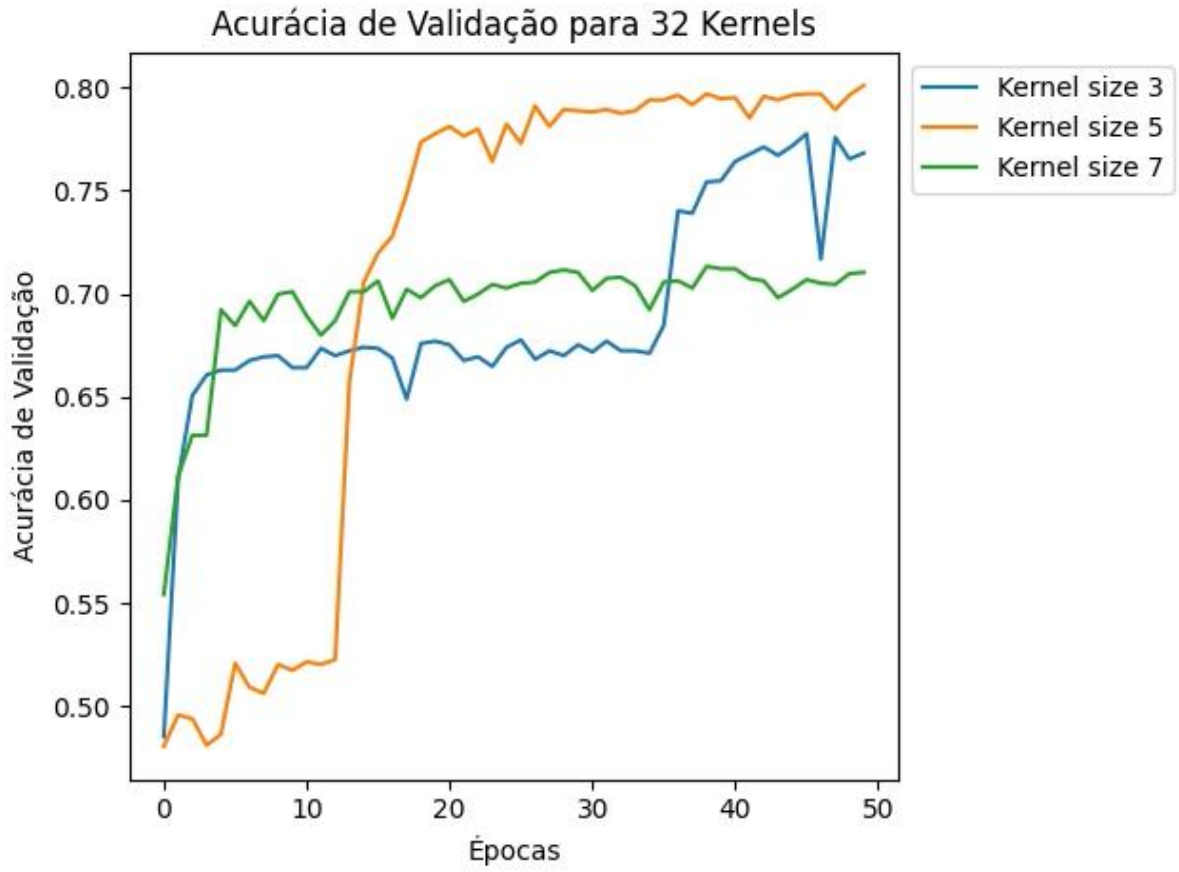
Epoch 46, Val Accuracy: 0.8163

Epoch 47, Val Accuracy: 0.8233

Epoch 48, Val Accuracy: 0.8122

Epoch 49, Val Accuracy: 0.7970

Epoch 50, Val Accuracy: 0.8145



Avaliação com 32 kernels e tamanho de kernel 3

Epoch 1, Val Accuracy: 0.4851

Epoch 2, Val Accuracy: 0.6103

Epoch 3, Val Accuracy: 0.6507

Epoch 4, Val Accuracy: 0.6606

Epoch 5, Val Accuracy: 0.6630

Epoch 6, Val Accuracy: 0.6630

Epoch 7, Val Accuracy: 0.6676

Epoch 8, Val Accuracy: 0.6694

Epoch 9, Val Accuracy: 0.6700

Epoch 10, Val Accuracy: 0.6641

Epoch 11, Val Accuracy: 0.6641

Epoch 12, Val Accuracy: 0.6735

Epoch 13, Val Accuracy: 0.6700

Epoch 14, Val Accuracy: 0.6723

Epoch 15, Val Accuracy: 0.6741

Epoch 16, Val Accuracy: 0.6735

Epoch 17, Val Accuracy: 0.6688

Epoch 18, Val Accuracy: 0.6489

Epoch 19, Val Accuracy: 0.6758

Epoch 20, Val Accuracy: 0.6770

Epoch 21, Val Accuracy: 0.6752

Epoch 22, Val Accuracy: 0.6676

Epoch 23, Val Accuracy: 0.6694

Epoch 24, Val Accuracy: 0.6647

Epoch 25, Val Accuracy: 0.6741

Epoch 26, Val Accuracy: 0.6776

Epoch 27, Val Accuracy: 0.6682

Epoch 28, Val Accuracy: 0.6723

Epoch 29, Val Accuracy: 0.6700

Epoch 30, Val Accuracy: 0.6752

Epoch 31, Val Accuracy: 0.6717

Epoch 32, Val Accuracy: 0.6770

Epoch 33, Val Accuracy: 0.6723

Epoch 34, Val Accuracy: 0.6723

Epoch 35, Val Accuracy: 0.6712

Epoch 36, Val Accuracy: 0.6846

Epoch 37, Val Accuracy: 0.7402

Epoch 38, Val Accuracy: 0.7390

Epoch 39, Val Accuracy: 0.7542

Epoch 40, Val Accuracy: 0.7548

Epoch 41, Val Accuracy: 0.7642

Epoch 42, Val Accuracy: 0.7677

Epoch 43, Val Accuracy: 0.7712

Epoch 44, Val Accuracy: 0.7671

Epoch 45, Val Accuracy: 0.7718

Epoch 46, Val Accuracy: 0.7776

Epoch 47, Val Accuracy: 0.7168

Epoch 48, Val Accuracy: 0.7759

Epoch 49, Val Accuracy: 0.7654

Epoch 50, Val Accuracy: 0.7683

Avaliação com 32 kernels e tamanho de kernel 5

Epoch 1, Val Accuracy: 0.4804

Epoch 2, Val Accuracy: 0.4956

Epoch 3, Val Accuracy: 0.4939

Epoch 4, Val Accuracy: 0.4810

Epoch 5, Val Accuracy: 0.4862

Epoch 6, Val Accuracy: 0.5208

Epoch 7, Val Accuracy: 0.5091

Epoch 8, Val Accuracy: 0.5061

Epoch 9, Val Accuracy: 0.5202

Epoch 10, Val Accuracy: 0.5173

Epoch 11, Val Accuracy: 0.5214

Epoch 12, Val Accuracy: 0.5202

Epoch 13, Val Accuracy: 0.5225

Epoch 14, Val Accuracy: 0.6577

Epoch 15, Val Accuracy: 0.7057

Epoch 16, Val Accuracy: 0.7197

Epoch 17, Val Accuracy: 0.7279

Epoch 18, Val Accuracy: 0.7484

Epoch 19, Val Accuracy: 0.7736

Epoch 20, Val Accuracy: 0.7776

Epoch 21, Val Accuracy: 0.7812

Epoch 22, Val Accuracy: 0.7765

Epoch 23, Val Accuracy: 0.7800

Epoch 24, Val Accuracy: 0.7642

Epoch 25, Val Accuracy: 0.7823

Epoch 26, Val Accuracy: 0.7730

Epoch 27, Val Accuracy: 0.7911

Epoch 28, Val Accuracy: 0.7812

Epoch 29, Val Accuracy: 0.7894

Epoch 30, Val Accuracy: 0.7888

Epoch 31, Val Accuracy: 0.7882

Epoch 32, Val Accuracy: 0.7894

Epoch 33, Val Accuracy: 0.7876

Epoch 34, Val Accuracy: 0.7888

Epoch 35, Val Accuracy: 0.7940

Epoch 36, Val Accuracy: 0.7940

Epoch 37, Val Accuracy: 0.7964

Epoch 38, Val Accuracy: 0.7917

Epoch 39, Val Accuracy: 0.7970

Epoch 40, Val Accuracy: 0.7946

Epoch 41, Val Accuracy: 0.7952

Epoch 42, Val Accuracy: 0.7853

Epoch 43, Val Accuracy: 0.7958

Epoch 44, Val Accuracy: 0.7940

Epoch 45, Val Accuracy: 0.7964

Epoch 46, Val Accuracy: 0.7970

Epoch 47, Val Accuracy: 0.7970

Epoch 48, Val Accuracy: 0.7894

Epoch 49, Val Accuracy: 0.7964

Epoch 50, Val Accuracy: 0.8011

Avaliação com 32 kernels e tamanho de kernel 7

Epoch 1, Val Accuracy: 0.5541

Epoch 2, Val Accuracy: 0.6115

Epoch 3, Val Accuracy: 0.6314

Epoch 4, Val Accuracy: 0.6314

Epoch 5, Val Accuracy: 0.6922

Epoch 6, Val Accuracy: 0.6846

Epoch 7, Val Accuracy: 0.6963

Epoch 8, Val Accuracy: 0.6870

Epoch 9, Val Accuracy: 0.6998

Epoch 10, Val Accuracy: 0.7010

Epoch 11, Val Accuracy: 0.6893

Epoch 12, Val Accuracy: 0.6799

Epoch 13, Val Accuracy: 0.6870

Epoch 14, Val Accuracy: 0.7010

Epoch 15, Val Accuracy: 0.7010

Epoch 16, Val Accuracy: 0.7063

Epoch 17, Val Accuracy: 0.6881

Epoch 18, Val Accuracy: 0.7022

Epoch 19, Val Accuracy: 0.6981

Epoch 20, Val Accuracy: 0.7039

Epoch 21, Val Accuracy: 0.7068

Epoch 22, Val Accuracy: 0.6963

Epoch 23, Val Accuracy: 0.6998

Epoch 24, Val Accuracy: 0.7045

Epoch 25, Val Accuracy: 0.7028

Epoch 26, Val Accuracy: 0.7051

Epoch 27, Val Accuracy: 0.7057

Epoch 28, Val Accuracy: 0.7104

Epoch 29, Val Accuracy: 0.7115

Epoch 30, Val Accuracy: 0.7104

Epoch 31, Val Accuracy: 0.7016

Epoch 32, Val Accuracy: 0.7074

Epoch 33, Val Accuracy: 0.7080

Epoch 34, Val Accuracy: 0.7039

Epoch 35, Val Accuracy: 0.6922

Epoch 36, Val Accuracy: 0.7057

Epoch 37, Val Accuracy: 0.7063

Epoch 38, Val Accuracy: 0.7028

Epoch 39, Val Accuracy: 0.7133

Epoch 40, Val Accuracy: 0.7121

Epoch 41, Val Accuracy: 0.7121

Epoch 42, Val Accuracy: 0.7074

Epoch 43, Val Accuracy: 0.7063

Epoch 44, Val Accuracy: 0.6981

Epoch 45, Val Accuracy: 0.7022

Epoch 46, Val Accuracy: 0.7068

Epoch 47, Val Accuracy: 0.7051

Epoch 48, Val Accuracy: 0.7045

Epoch 49, Val Accuracy: 0.7098

Epoch 50, Val Accuracy: 0.7104

Código Python do item c)

```
import time
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from sklearn.metrics import confusion_matrix, accuracy_score
import numpy as np
from PIL import Image
import medmnist
from medmnist import INFO
import pandas as pd
import openpyxl
import seaborn as sns
import matplotlib.pyplot as plt

class MedMNISTDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform
```

```
    def __len__(self):
        return len(self.images)
```

```
    def __getitem__(self, idx):
        img, label = self.images[idx], self.labels[idx]
        img = Image.fromarray(img)
        if self.transform:
            img = self.transform(img)
        return img, label[0]
```

```
class SimpleCNN(nn.Module):
    def __init__(self, num_kernels, kernel_size):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, num_kernels, kernel_size, padding=3)  #
Ajuste no padding para manter o tamanho
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(num_kernels * 16 * 16, 128)  # Ajuste para
dimensionar corretamente após pooling
        self.fc2 = nn.Linear(128, 8)  # Camada de saída para 8 classes
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.flatten(x)
```

```
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

```
def load_and_prepare_data():
    data_flag = 'bloodmnist'
    info = INFO[data_flag]
    DataClass = getattr(medmnist, info['python_class'])
```

```
    train_data = DataClass(split='train', download=True)
    val_data = DataClass(split='val', download=True)
    test_data = DataClass(split='test', download=True)
```

```
    images = np.concatenate((train_data.imgs, val_data.imgs, test_data.imgs),
axis=0)
    labels = np.concatenate((train_data.labels, val_data.labels,
test_data.labels), axis=0)
```

```
    total_images = images.shape[0]
    print(f"Total de imagens: {total_images}")
```

```
    transform = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor()
    ])
```

```
    dataset = MedMNISTDataset(images, labels, transform=transform)
    train_size = int(0.7 * len(dataset))
    val_size = int(0.1 * len(dataset))
    test_size = len(dataset) - train_size - val_size
    train_dataset, val_dataset, test_dataset =
torch.utils.data.random_split(dataset, [train_size, val_size, test_size])
```

```
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

```
    return train_loader, val_loader, test_loader
```

```
def plot_confusion_matrix(cm, classes):
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=classes,
yticklabels=classes)
    plt.xlabel('Predito')
    plt.ylabel('Verdadeiro')
    plt.title('Matriz de Confusão')
    plt.show()
```

```
def plot_misclassified_images(images, true_labels, predicted_labels,
probabilities, class_names):
    plt.figure(figsize=(20, 10))
```

```
    for i in range(5):
        plt.subplot(1, 5, i + 1)
        plt.imshow(images[i].permute(1, 2, 0))  # Converte de (C, H, W) para
(H, W, C)
        true_class = class_names[true_labels[i]]
        pred_class = class_names[predicted_labels[i]]
        pred_prob = probabilities[i]
        title = f"V: {true_class}\nP: {pred_class}\nProb: {pred_prob:.2f}"
        plt.title(title)
        plt.axis('off')
    plt.show()
```

```
def train_and_evaluate_model(train_loader, val_loader, test_loader):
    model = SimpleCNN(16, 7)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
    num_epochs = 50
    start_time = time.time()
    for epoch in range(num_epochs):
        epoch_start_time = time.time()
        model.train()
        for images, labels in train_loader:
            outputs = model(images)
            loss = criterion(outputs, labels.long())
```

```
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
        epoch_end_time = time.time()
        epoch_duration = epoch_end_time - epoch_start_time
        print(f"Epoch {epoch + 1}/{num_epochs} - Duration: {epoch_duration:.2f}
seconds")
```

```
    end_time = time.time()
    total_duration = end_time - start_time
    print(f"Total training time: {total_duration:.2f} seconds")
```

```
    model.eval()
    all_labels = []
    all_preds = []
    all_images = []
    all_probs = []
    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            probs = nn.Softmax(dim=1)(outputs)
            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.numpy())
            all_preds.extend(preds.numpy())
```

```
            all_images.extend(images)
            all_probs.extend(probs.numpy())
```

```
    acc = accuracy_score(all_labels, all_preds)
    cm = confusion_matrix(all_labels, all_preds)
```

```
    class_names = ["Basófilos", "Eosinófilos", "Eritroblastos", "Granulócitos",
"Linfócitos", "Monócitos", "Neutrófilos", "Plaqueta"]
    plot_confusion_matrix(cm, classes=class_names)
```

```
    # Identificar e plotar as imagens classificadas incorretamente
    misclassified_idxs = [i for i in range(len(all_labels)) if
all_labels[i] != all_preds[i]]
    misclassified_images = [all_images[i] for i in misclassified_idxs[:5]]
    misclassified_true_labels = [all_labels[i] for i in misclassified_idxs[:5]]
    misclassified_pred_labels = [all_preds[i] for i in misclassified_idxs[:5]]
    misclassified_probs = [all_probs[i][all_preds[i]] for i in
misclassified_idxs[:5]]
    plot_misclassified_images(misclassified_images, misclassified_true_labels,
misclassified_pred_labels, misclassified_probs, class_names)
```

```
    return acc, cm
```

```
def main():
    train_loader, val_loader, test_loader = load_and_prepare_data()
    test_accuracy, confusion_matrix = train_and_evaluate_model(train_loader,
val_loader, test_loader)
    print(f"Acurácia nos dados de teste: {test_accuracy * 100:.2f}%")
    print("Matriz de Confusão:")
    print(confusion_matrix)
```

```
if __name__ == '__main__':
    main()
```

Saída do Terminal do item c)

Total de imagens: 17092

Epoch 1/50 - Duration: 7.30 seconds

Epoch 2/50 - Duration: 8.45 seconds

Epoch 3/50 - Duration: 10.36 seconds

Epoch 4/50 - Duration: 10.53 seconds

Epoch 5/50 - Duration: 10.16 seconds

Epoch 6/50 - Duration: 10.28 seconds

Epoch 7/50 - Duration: 10.13 seconds

Epoch 8/50 - Duration: 9.77 seconds

Epoch 9/50 - Duration: 10.63 seconds

Epoch 10/50 - Duration: 9.86 seconds

Epoch 11/50 - Duration: 9.84 seconds

Epoch 12/50 - Duration: 10.27 seconds

Epoch 13/50 - Duration: 9.67 seconds

Epoch 14/50 - Duration: 9.70 seconds

Epoch 15/50 - Duration: 9.89 seconds

Epoch 16/50 - Duration: 9.83 seconds

Epoch 17/50 - Duration: 9.65 seconds

Epoch 18/50 - Duration: 10.34 seconds

Epoch 19/50 - Duration: 7.50 seconds

Epoch 20/50 - Duration: 7.04 seconds

Epoch 21/50 - Duration: 7.13 seconds

Epoch 22/50 - Duration: 6.92 seconds

Epoch 23/50 - Duration: 7.73 seconds

Epoch 24/50 - Duration: 7.86 seconds

Epoch 25/50 - Duration: 7.67 seconds

Epoch 26/50 - Duration: 8.69 seconds

Epoch 27/50 - Duration: 7.33 seconds

Epoch 28/50 - Duration: 6.72 seconds

Epoch 29/50 - Duration: 7.16 seconds

Epoch 30/50 - Duration: 7.15 seconds

Epoch 31/50 - Duration: 6.74 seconds

Epoch 32/50 - Duration: 6.44 seconds

Epoch 33/50 - Duration: 6.42 seconds

Epoch 34/50 - Duration: 6.35 seconds

Epoch 35/50 - Duration: 6.58 seconds

Epoch 36/50 - Duration: 6.43 seconds

Epoch 37/50 - Duration: 6.65 seconds

Epoch 38/50 - Duration: 6.45 seconds

Epoch 39/50 - Duration: 6.45 seconds

Epoch 40/50 - Duration: 6.54 seconds

Epoch 41/50 - Duration: 6.70 seconds

Epoch 42/50 - Duration: 6.63 seconds

Epoch 43/50 - Duration: 6.58 seconds

Epoch 44/50 - Duration: 6.59 seconds

Epoch 45/50 - Duration: 6.58 seconds

Epoch 46/50 - Duration: 6.50 seconds

Epoch 47/50 - Duration: 6.65 seconds

Epoch 48/50 - Duration: 6.56 seconds

Epoch 49/50 - Duration: 6.56 seconds

Epoch 50/50 - Duration: 6.65 seconds

Total training time: 396.64 seconds

Acurácia nos dados de teste: 90.03%

Matriz de Confusão:

[[182 9 0 34 4 7 1 0]

[1 653 1 1 0 1 7 0]

[0 1 287 10 8 2 4 4]

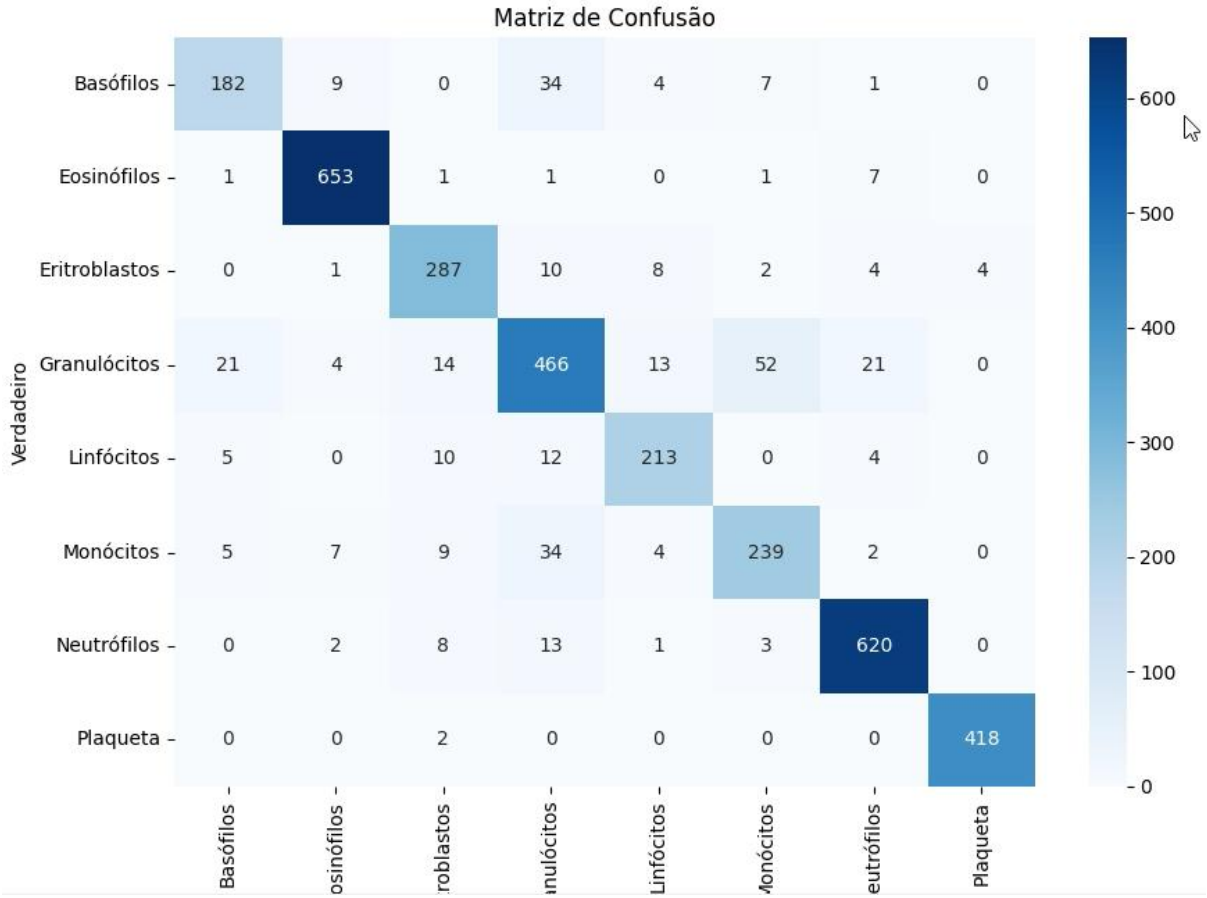
[21 4 14 466 13 52 21 0]

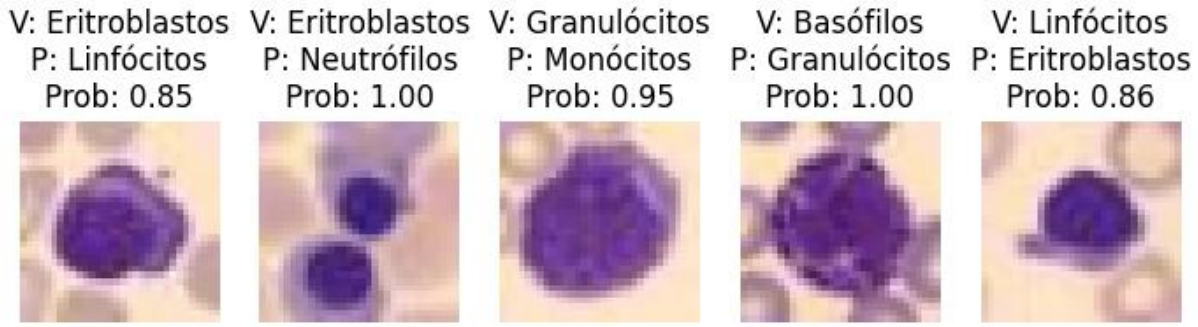
[5 0 10 12 213 0 4 0]

[5 7 9 34 4 239 2 0]

[0 2 8 13 1 3 620 0]

[0 0 2 0 0 0 0 418]]


Matriz de Confusão
Verdadeiro
Basófilos
Eosinófilos
Eritroblastos
Granulócitos
Linfócitos
Monócitos
Neutrófilos
Plaqueta
182 9 0 34 4 7 1 0
1 653 1 1 0 1 7 0
0 1 287 10 8 2 4 4
21 4 14 466 13 52 21 0
5 0 10 12 213 0 4 0
5 7 9 34 4 239 2 0
0 2 8 13 1 3 620 0
0 0 2 0 0 0 0 418
600
500
400
300
200
100
0


V: Eritroblastos
P: Linfócitos
Prob: 0.85
V: Eritroblastos
P: Neutrófilos
Prob: 1.00
V: Granulócitos
P: Monócitos
Prob: 0.95
V: Basófilos
P: Granulócitos
Prob: 1.00
V: Linfócitos
P: Eritroblastos
Prob: 0.86

Código Python do item d)

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report
import numpy as np
import pandas as pd
from PIL import Image
import medmnist
from medmnist import INFO
import matplotlib.pyplot as plt
import seaborn as sns
import time

class_names = ["Basófilos", "Eosinófilos", "Eritroblastos", "Granulócitos",
"Linfócitos", "Monócitos", "Neutrófilos",
               "Plaquetas"]
```

```
class MedMNISTDataset(Dataset):
    def __init__(self, images, labels, transform=None):
        self.images = images
        self.labels = labels
        self.transform = transform
```

```
    def __len__(self):
        return len(self.images)
```

```
    def __getitem__(self, idx):
        img, label = self.images[idx], self.labels[idx]
        img = Image.fromarray(img)
        if self.transform:
            img = self.transform(img)
        return img, label[0]
```

```
class Block(nn.Module):
    def __init__(self, in_channels, out_channels, identity_downsample=None,
stride=1):
        super(Block, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=1, padding=0)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=stride, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
```

```
        self.conv3 = nn.Conv2d(out_channels, out_channels * 4, kernel_size=1,
stride=1, padding=0)
        self.bn3 = nn.BatchNorm2d(out_channels * 4)
        self.relu = nn.ReLU()
        self.identity_downsample = identity_downsample
```

```
    def forward(self, x):
        identity = x
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.conv3(x)
        x = self.bn3(x)
        if self.identity_downsample is not None:
            identity = self.identity_downsample(identity)
        x += identity
        x = self.relu(x)
        return x
```

```
class ResNet(nn.Module):
    def __init__(self, block, layers, image_channels, num_classes):
        super(ResNet, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(image_channels, 64, kernel_size=7, stride=2,
padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0], stride=1)
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * 4, num_classes)
```

```
    def _make_layer(self, block, out_channels, num_blocks, stride):
        identity_downsample = None
        if stride != 1 or self.in_channels != out_channels * 4:
            identity_downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels * 4, kernel_size=1,
stride=stride),
                nn.BatchNorm2d(out_channels * 4))
        layers = []
        layers.append(block(self.in_channels, out_channels,
identity_downsample, stride))
        self.in_channels = out_channels * 4
```

```
        for i in range(1, num_blocks):
            layers.append(block(self.in_channels, out_channels))
        return nn.Sequential(*layers)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = x.reshape(x.shape[0], -1)
        x = self.fc(x)
        return x
```

```
def load_and_prepare_data():
    data_flag = 'bloodmnist'
    info = INFO[data_flag]
    DataClass = getattr(medmnist, info['python_class'])
    train_data = DataClass(split='train', download=True)
    val_data = DataClass(split='val', download=True)
    test_data = DataClass(split='test', download=True)
    images = np.concatenate((train_data.imgs, val_data.imgs, test_data.imgs),
axis=0)
    labels = np.concatenate((train_data.labels, val_data.labels,
test_data.labels), axis=0)
    transform = transforms.Compose([transforms.Resize((32, 32)),
transforms.ToTensor()])
    dataset = MedMNISTDataset(images, labels, transform=transform)
    train_size = int(0.7 * len(dataset))
    val_size = int(0.1 * len(dataset))
    test_size = len(dataset) - train_size - val_size
    train_dataset = DataLoader(torch.utils.data.Subset(dataset,
range(train_size)), batch_size=32, shuffle=True)
    val_dataset = DataLoader(torch.utils.data.Subset(dataset, range(train_size,
train_size + val_size)), batch_size=32,
                             shuffle=False)
    test_dataset = DataLoader(torch.utils.data.Subset(dataset,
range(train_size + val_size, len(dataset))),
                              batch_size=32, shuffle=False)
    return train_dataset, val_dataset, test_dataset
```

```
def display_incorrect_samples(test_loader, model, device, num_samples=5):
    incorrects = []
    model.eval()
    with torch.no_grad():
```

```
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            probs = torch.nn.functional.softmax(outputs, dim=1)
            for i in range(len(images)):
                if preds[i] != labels[i]:
                    incorrects.append((images[i], labels[i], preds[i],
probs[i][preds[i]].item()))
                if len(incorrects) >= num_samples:
                    break
            if len(incorrects) >= num_samples:
                break
    fig, axes = plt.subplots(1, num_samples, figsize=(15, 3))
    for ax, (img, true_label, pred_label, prob) in zip(axes, incorrects):
        img = img.cpu().numpy().transpose((1, 2, 0))
        img = (img - img.min()) / (img.max() - img.min())
        ax.imshow(img)
        ax.set_title(
            f'Verdadeiro: {class_names[true_label.item()]}\nPredito:
{class_names[pred_label.item()]}\nProb: {prob:.2f}')
        ax.axis('off')
    plt.show()
```

```
def plot_classification_report(report, class_names):
    report_data = []
    for label, metrics in report.items():
        if isinstance(metrics, dict):
            report_data.append({
                'class': label,
                'precision': metrics['precision'],
                'recall': metrics['recall'],
                'f1-score': metrics['f1-score'],
                'support': metrics['support']
            })
    report_df = pd.DataFrame.from_records(report_data)
    report_df.set_index('class', inplace=True)
```

```
    plt.figure(figsize=(12, 7))
    sns.heatmap(report_df[['precision', 'recall', 'f1-score']], annot=True,
cmap="YlGnBu", vmin=0, vmax=1)
    plt.title('Relatório de Classificação para Teste')
    plt.show()
```

```
def train_and_evaluate_model(train_loader, val_loader, test_loader, device):
    model = ResNet(Block, [3, 4, 6, 3], 3, len(class_names)).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    num_epochs = 30
```

```
    for epoch in range(num_epochs):
        start_time = time.time()  # Start timer
        print(f"Epoch {epoch + 1}/{num_epochs}")
        model.train()
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
        end_time = time.time()  # End timer
        epoch_duration = end_time - start_time
        print(f"Epoch {epoch + 1} completed in {epoch_duration:.2f} seconds")
```

```
    model.eval()
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())
    acc = accuracy_score(all_labels, all_preds)
    cm = confusion_matrix(all_labels, all_preds)
    print(f"Acurácia Global: {acc * 100:.2f}%")
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names,
yticklabels=class_names)
    plt.xlabel('Predito')
    plt.ylabel('Verdadeiro')
    plt.title('Matriz de Confusão')
    plt.show()
```

```
    # Generate classification report
    report = classification_report(all_labels, all_preds,
target_names=class_names, output_dict=True)
    plot_classification_report(report, class_names)
```

```
    display_incorrect_samples(test_loader, model, device)
```

```
def main():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    train_loader, val_loader, test_loader = load_and_prepare_data()
    train_and_evaluate_model(train_loader, val_loader, test_loader, device)
```

```
if __name__ == "__main__":
    main()
```

Saída do Terminal do item d)

Epoch 1/30

Epoch 1 completed in 194.69 seconds

Epoch 2/30

Epoch 2 completed in 192.20 seconds

Epoch 3/30

Epoch 3 completed in 188.71 seconds

Epoch 4/30

Epoch 4 completed in 187.59 seconds

Epoch 5/30

Epoch 5 completed in 187.81 seconds

Epoch 6/30

Epoch 6 completed in 188.96 seconds

Epoch 7/30

Epoch 7 completed in 187.91 seconds

Epoch 8/30

Epoch 8 completed in 188.34 seconds

Epoch 9/30

Epoch 9 completed in 188.64 seconds

Epoch 10/30

Epoch 10 completed in 184.43 seconds

Epoch 11/30

Epoch 11 completed in 184.63 seconds

Epoch 12/30

Epoch 12 completed in 187.60 seconds

Epoch 13/30

Epoch 13 completed in 187.55 seconds

Epoch 14/30

Epoch 14 completed in 187.42 seconds

Epoch 15/30

Epoch 15 completed in 186.94 seconds

Epoch 16/30

Epoch 16 completed in 187.14 seconds

Epoch 17/30

Epoch 17 completed in 186.72 seconds

Epoch 18/30

Epoch 18 completed in 186.95 seconds

Epoch 19/30

Epoch 19 completed in 186.00 seconds

Epoch 20/30

Epoch 20 completed in 185.81 seconds

Epoch 21/30

Epoch 21 completed in 187.06 seconds

Epoch 22/30

Epoch 22 completed in 186.81 seconds

Epoch 23/30

Epoch 23 completed in 186.60 seconds

Epoch 24/30

Epoch 24 completed in 187.08 seconds

Epoch 25/30

Epoch 25 completed in 188.90 seconds

Epoch 26/30

Epoch 26 completed in 192.03 seconds

Epoch 27/30

Epoch 27 completed in 189.71 seconds

Epoch 28/30

Epoch 28 completed in 187.29 seconds

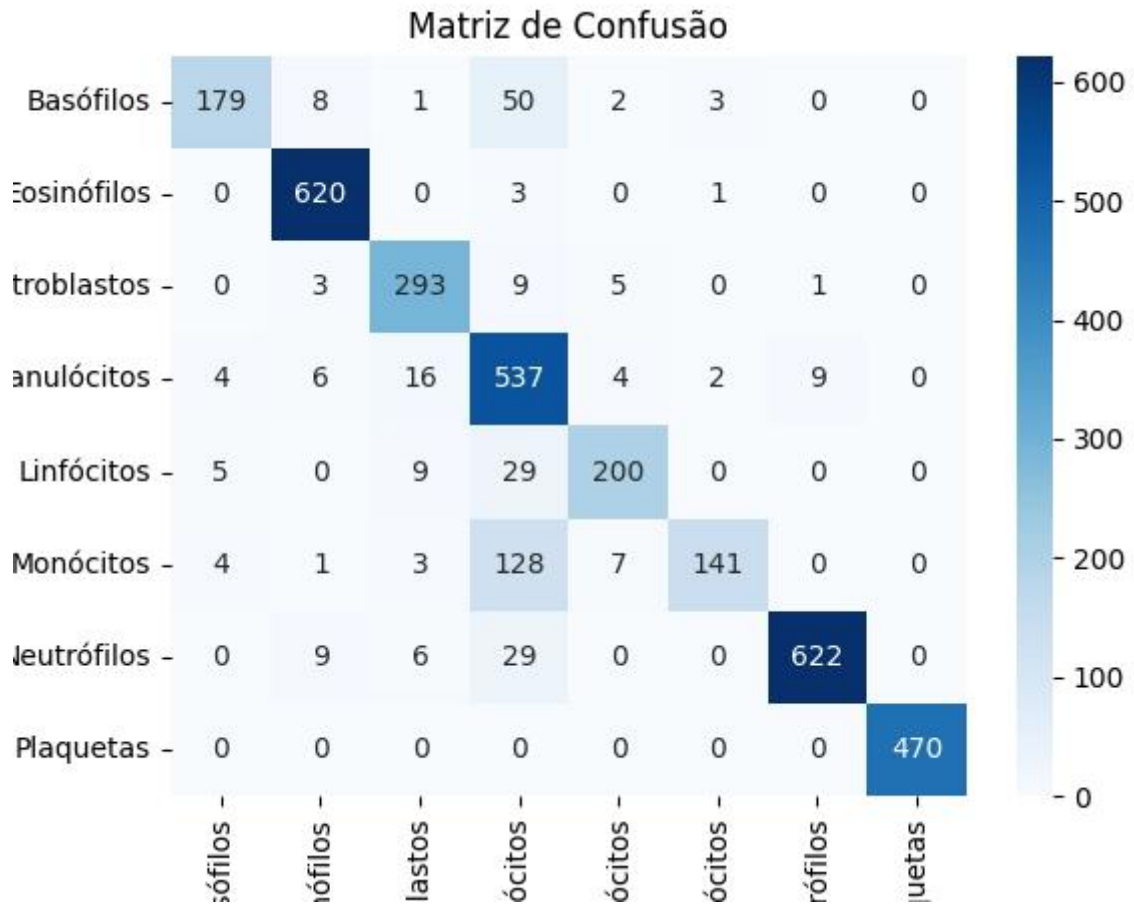
Epoch 29/30

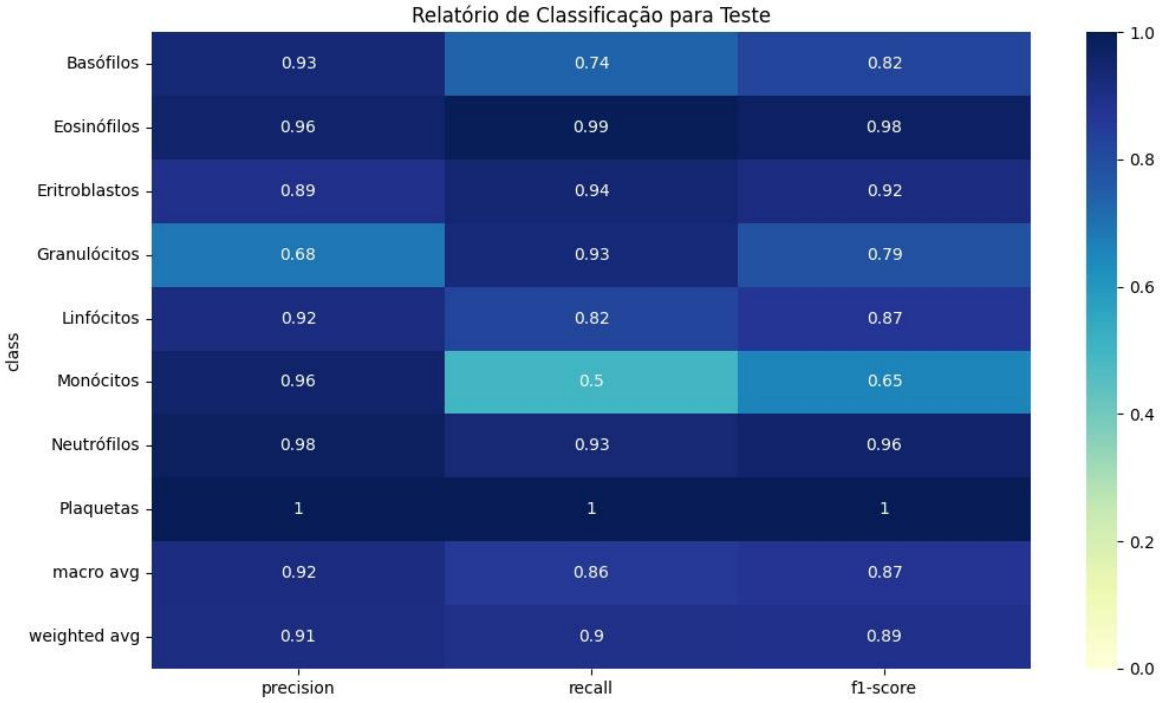
Epoch 29 completed in 187.46 seconds

Epoch 30/30

Epoch 30 completed in 190.70 seconds

Acurácia Global: 89.56%


Matriz de Confusão
Basófilos
Eosinófilos
troblastos
anulócitos
Linfócitos
Monócitos
Neutrófilos
Plaquetas
179 8 1 50 2 3 0 0
0 620 0 3 0 1 0 0
0 3 293 9 5 0 1 0
4 6 16 537 4 2 9 0
5 0 9 29 200 0 0 0
4 1 3 128 7 141 0 0
0 9 6 29 0 0 622 0
0 0 0 0 0 0 0 470
600
500
400
300
200
100
0


Relatório de Classificação para Teste
class
Basófilos 0.93 0.74 0.82
Eosinófilos 0.96 0.99 0.98
Eritroblastos 0.89 0.94 0.92
Granulócitos 0.68 0.93 0.79
Linfócitos 0.92 0.82 0.87
Monócitos 0.96 0.5 0.65
Neutrófilos 0.98 0.93 0.96
Plaquetas 1 1 1
macro avg 0.92 0.86 0.87
weighted avg 0.91 0.9 0.89
precision
recall
f1-score
1.0
0.8
0.6
0.4
0.2
0.0