# Trabalho Final - Inteligência Artificial

Maria Eduarda de Paula Duarte - 2021.1905.061-6

Raquel Freire Cerzosimo - 2020.1905.009-6

Larissa Marques Moraes - 2021.1905.032-2

## 1 Introdução

O problema escolhido foi a classificação de plantas, problema que possui diversas aplicações práticas. Um grande exemplo é a utilização desse mecanismo de identificação em plantações, para, por exemplo, identificar pragas ou espécies invasoras. Nosso algoritmo tem como objetivo processar diversas imagens de plantas recebidas e conseguir corretamente diferenciar as espécies.

## 2 Conjunto de Dados

O conjunto de dados utilizado foi o dataset público de flores provido pela Kaggle. Nela, é provido diversas imagens de 5 tipos de flores, contendo aproximadamente 800 imagens de cada tipo.

Temos 5 classes de flores:

- Tulip - tulipa (1.212 imagens)
- Daisy - margarida (927 imagens)
- Rose - Rosa (975 imagens)
- Dandelion - Dente-de-leão (1.239 imagens)
- Sunflower - Girassol (885 imagens)

Dessa forma separamos nosso dataset em: treino (flower_train), validação (flower_valid) e teste (flower_test). A divisão resultou em:

- Treino:
  - Tulip - 984 imagens
  - Daisy - 764 imagens
  - Rose - 784 imagens
  - Dandelion - 1052 imagens
  - Sunflower - 733 imagens
- Validção:
  - Tulip - 138 imagens
  - Daisy - 107 imagens

  - Rose - 118 imagens
  - Dandelion - 119 imagens
  - Sunflower - 97 imagens

- Teste:
  - Tulip - 90 imagens
  - Daisy - 56 imagens
  - Rose - 73 imagens
  - Dandelion - 68 imagens
  - Sunflower - 55 imagens

# 3 Pré-processamento

Após fazer a divisão do dataset criamos uma seção de código dedicada ao pré-processamendo das nossas imagens. Nele utilizamos as transformações providas pela biblioteca *torch* para normalizar, redimensionar as imagens (utilizamos aqui o padrão de 224x224) e salvá-las em um novo diretório, para serem utilizadas posteriormente no modelo.

A normalização é um processo que ajusta os valores dos pixels para facilitar a posterior leitura pelo modelo, pois reduz a assimetria tornando o processo de aprendizagem mais rápido e mais efetivo. O cálculo da normalização da biblioteca utiliza dois valores, o *mean* e o *std*, fazendo o seguinte cálculo para cada canal da imagem: $image = (image - mean)/std$.

No nosso caso, os valores de *mean* e *std* utilizados são os calculados pela Imagenet, que são valores calculados baseado em milhares de imagens e faz parte das boas práticas a utilização desses valores (quando não se está fazendo o próprio calculo com base em seu dataset).

Código com todas as transformações utilizadas nas imagens:

```
    transforms = v2.Compose([
    v2.ToImage(),
    v2.Resize(size=(224, 224), antialias=True),
    v2.RandomHorizontalFlip(p=0.5),
    v2.ToDtype(torch.float32, scale=True),
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    v2.ToPILImage()
])
```

# 4 Modelo

Para o treinamento do modelo usamos uma (CNN) rede neural convolucional.

Na primeira parte do código temos o gerador de dados. Este código configura geradores de dados de imagens utilizando a classe ImageDataGenerator do Keras, com o objetivo de preprocessar imagens para um modelo de aprendizado de máquina. Ele define geradores de dados para treino, validação e teste, com a finalidade de alimentar um modelo de rede neural durante o treinamento e avaliação. Cada gerador é responsável por uma fase específica (treino, validação e teste), garantindo que as imagens sejam corretamente normalizadas e organizadas para otimizar o desempenho do modelo.

Na parte seguinte temos a construção da CNN. Esa parte do código cria, compila, treina e avalia uma rede neural convolucional (CNN) usando o Keras. O modelo é projetado para classificar imagens de flores em 5 categorias diferentes. Podemos dividir a criação da CNN da seguinte forma:

- Modelo Sequencial
- Camadas Convolucionais e de Pooling
  - Primeira camada convolucional com 32 filtros de tamanho 3x3, ativação ReLU e input shape (150, 150, 3) para imagens de entrada de 150x150 pixels com 3 canais de cor (RGB).
  - Redução da dimensionalidade pela metade usando pooling máximo em regiões 2x2.
  - Segue-se com mais duas camadas, aumentando o número de filtros para 64 e 128, capturando características de alto nível das imagens.
- Camadas Densas e de Regularização
- Compilação
- Treinamento
- Avaliação e Previsões

Em resumo, o código configura e treina uma CNN para classificar imagens de flores em 5 categorias diferentes. Ele inclui a definição das camadas do modelo, compilação com um otimizador e função de perda apropriados, e treinamento com dados de treino e validação. Finalmente, o modelo é avaliado com dados de teste, produzindo previsões que podem ser comparadas com os rótulos verdadeiros para medir a performance.

# 5 Resultados

Após executar o programa diversas vezes, percebemos que quando fazíamos o pré-processamento das imagens tínhamos um resultado muito pior que esperávamos. Pelo fato das imagens serem providas de um dataset público, elas diferem muito em tamanho e qualidade entre si, por esse motivo, ao executar o programa com as imagens pré-processadas perdemos muito a precisão do mesmo.

A figura a seguir mostra os resultados obtidos com as imagens pré-processadas com os valores padrão:

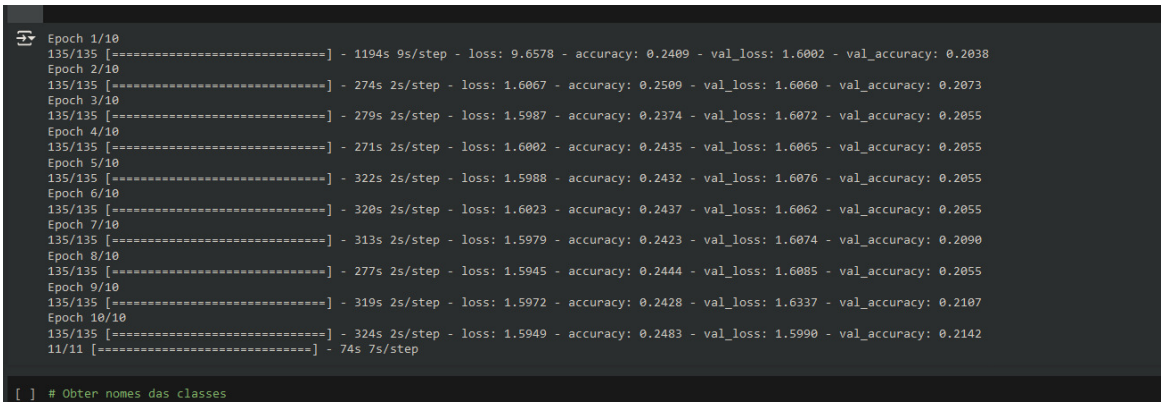


Figura 1: Resultado com imagens pré-processadas

A figura a seguir mostra os resultados obtidos com as imagens pré-processadas com 0.5 de parâmetros no pré-processamento:

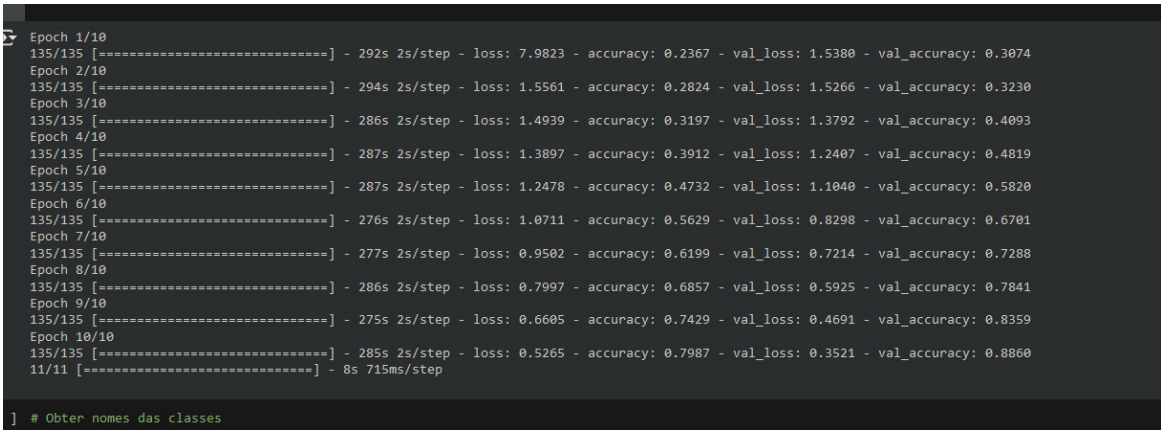


Figura 2: Resultado com 0.5 de parâmetro

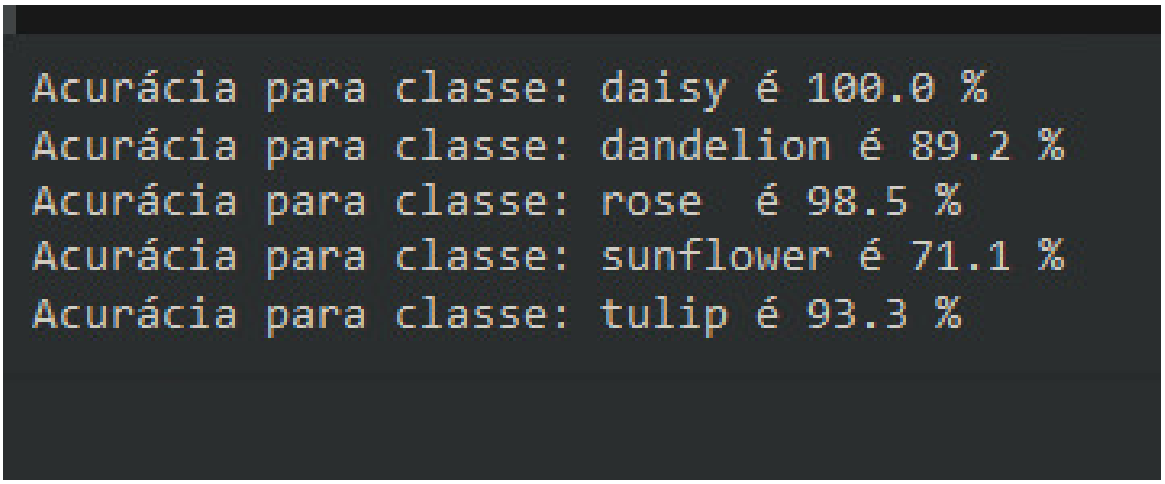


Figura 3: Resultado com 0.5 de parâmetro

A figura a seguir mostra os resultados obtidos com as imagens pré-processadas apenas com resize:

```
Epoch 1/10
135/135 [==============================] - 300s 2s/step - loss: 6.9299 - accuracy: 0.3308 - val_loss: 1.5714 - val_accuracy: 0.2539
Epoch 2/10
135/135 [==============================] - 285s 2s/step - loss: 1.5489 - accuracy: 0.2986 - val_loss: 1.5771 - val_accuracy: 0.3748
Epoch 3/10
135/135 [==============================] - 287s 2s/step - loss: 1.4406 - accuracy: 0.3653 - val_loss: 1.3130 - val_accuracy: 0.4525
Epoch 4/10
135/135 [==============================] - 277s 2s/step - loss: 1.3305 - accuracy: 0.4288 - val_loss: 1.2149 - val_accuracy: 0.4888
Epoch 5/10
135/135 [==============================] - 274s 2s/step - loss: 1.2661 - accuracy: 0.4424 - val_loss: 1.2289 - val_accuracy: 0.4680
Epoch 6/10
135/135 [==============================] - 270s 2s/step - loss: 1.2152 - accuracy: 0.4888 - val_loss: 0.9520 - val_accuracy: 0.6149
Epoch 7/10
135/135 [==============================] - 268s 2s/step - loss: 1.0403 - accuracy: 0.5615 - val_loss: 0.8704 - val_accuracy: 0.6615
Epoch 8/10
135/135 [==============================] - 268s 2s/step - loss: 0.8848 - accuracy: 0.6461 - val_loss: 0.7283 - val_accuracy: 0.7219
Epoch 9/10
135/135 [==============================] - 274s 2s/step - loss: 0.7689 - accuracy: 0.6935 - val_loss: 0.5738 - val_accuracy: 0.7824
Epoch 10/10
135/135 [==============================] - 273s 2s/step - loss: 0.6643 - accuracy: 0.7491 - val_loss: 0.4765 - val_accuracy: 0.8342
11/11 [==============================] - 8s 678ms/step
```

Figura 4: Resultado com as imagens pré processadas apenas com resize

```
Acurácia para classe: daisy é 100.0 %
Acurácia para classe: dandelion é 65.7 %
Acurácia para classe: rose  é 94.1 %
Acurácia para classe: sunflower é 95.5 %
Acurácia para classe: tulip é 91.6 %
```

Figura 5: Resultado com as imagens pré processadas apenas com resize

A figura a seguir mostra os resultados obtidos com as imagens pré-processadas sem resize apenas com normalização com parâmetro de 0.5:

```
Epoch 1/10
135/135 [==============================] - 285s 2s/step - loss: 6.1458 - accuracy: 0.2421 - val_loss: 1.5550 - val_accuracy: 0.2504
Epoch 2/10
135/135 [==============================] - 287s 2s/step - loss: 1.5621 - accuracy: 0.2750 - val_loss: 1.5420 - val_accuracy: 0.2642
Epoch 3/10
135/135 [==============================] - 318s 2s/step - loss: 1.5194 - accuracy: 0.3011 - val_loss: 1.4714 - val_accuracy: 0.3230
Epoch 4/10
135/135 [==============================] - 283s 2s/step - loss: 1.4322 - accuracy: 0.3495 - val_loss: 1.3458 - val_accuracy: 0.4128
Epoch 5/10
135/135 [==============================] - 279s 2s/step - loss: 1.3275 - accuracy: 0.4126 - val_loss: 1.2054 - val_accuracy: 0.4732
Epoch 6/10
135/135 [==============================] - 278s 2s/step - loss: 1.2339 - accuracy: 0.4607 - val_loss: 1.1474 - val_accuracy: 0.5026
Epoch 7/10
135/135 [==============================] - 278s 2s/step - loss: 1.1195 - accuracy: 0.5226 - val_loss: 0.9776 - val_accuracy: 0.6218
Epoch 8/10
135/135 [==============================] - 280s 2s/step - loss: 1.0163 - accuracy: 0.5724 - val_loss: 0.9524 - val_accuracy: 0.5907
Epoch 9/10
135/135 [==============================] - 287s 2s/step - loss: 0.9262 - accuracy: 0.6222 - val_loss: 0.8208 - val_accuracy: 0.6822
Epoch 10/10
135/135 [==============================] - 280s 2s/step - loss: 0.8438 - accuracy: 0.6618 - val_loss: 0.7202 - val_accuracy: 0.7306
7/7 [==============================] - 3s 399ms/step
```

Figura 6: Resultado com as imagens pré-processadas sem resize apenas com normalização com parâmetro de 0.5

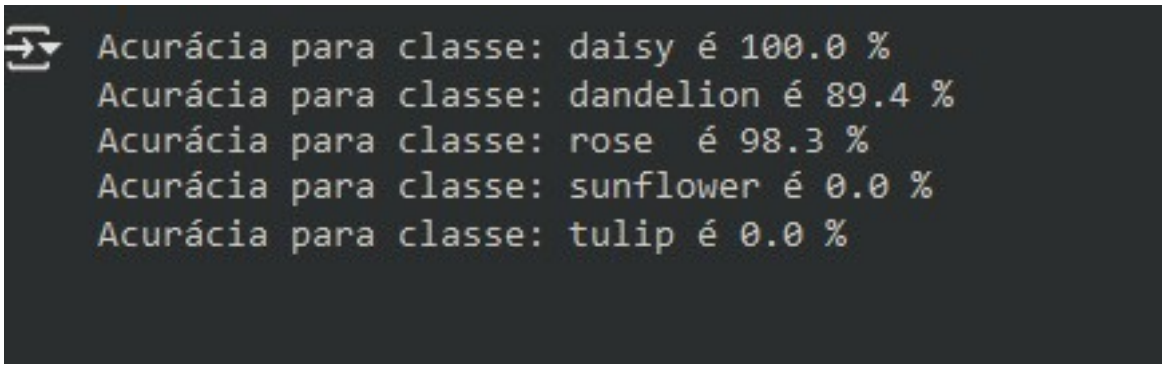


Figura 7: Resultado com as imagens pré-processadas sem resize apenas com normalização com parâmetro de 0.5

A seguir todos os melhores resultados obtidos considerando o código executado sem as imagens pré-processadas, apenas como elas foram baixadas para o dataset.

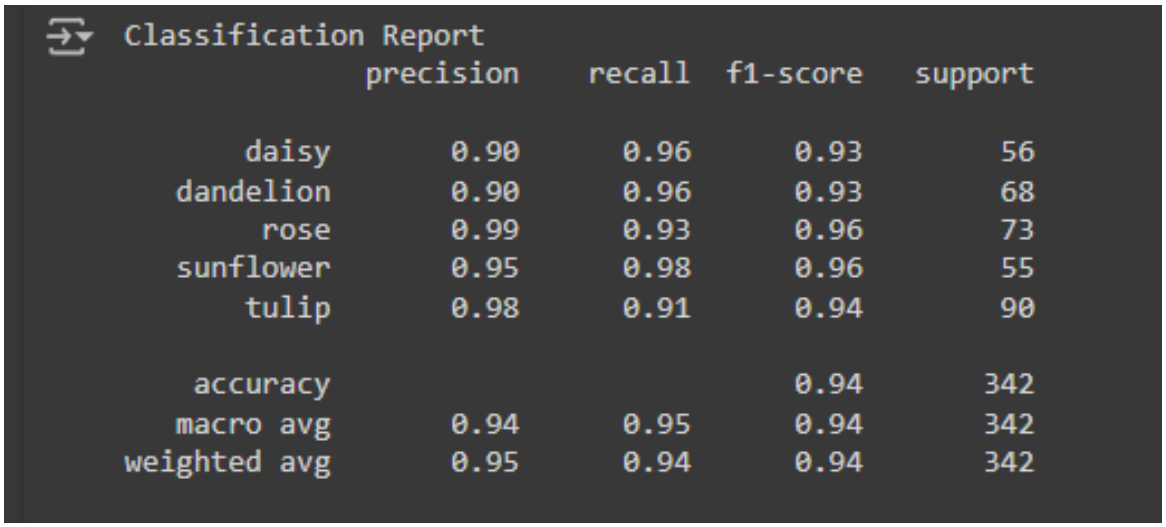


Figura 8: Resultado com as imagens originais do dataset

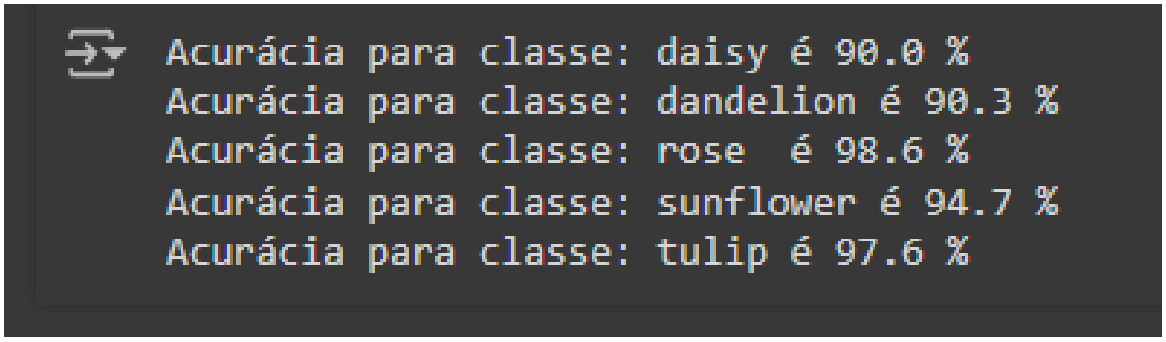


Figura 9: Resultado com as imagens originais do dataset5

```
Epoch 1/10
135/135 [==============================] - 32s 188ms/step - loss: 7.2332 - accuracy: 0.2636 - val_loss: 1.5352 - val_accuracy: 0.3541
Epoch 2/10
135/135 [==============================] - 25s 182ms/step - loss: 1.5168 - accuracy: 0.3526 - val_loss: 1.4900 - val_accuracy: 0.3921
Epoch 3/10
135/135 [==============================] - 25s 186ms/step - loss: 1.3538 - accuracy: 0.4355 - val_loss: 1.1334 - val_accuracy: 0.5544
Epoch 4/10
135/135 [==============================] - 25s 185ms/step - loss: 1.1780 - accuracy: 0.5196 - val_loss: 0.9822 - val_accuracy: 0.6045
Epoch 5/10
135/135 [==============================] - 26s 191ms/step - loss: 0.9762 - accuracy: 0.6169 - val_loss: 0.7158 - val_accuracy: 0.7496
Epoch 6/10
135/135 [==============================] - 25s 182ms/step - loss: 0.7594 - accuracy: 0.7190 - val_loss: 0.6769 - val_accuracy: 0.7547
Epoch 7/10
135/135 [==============================] - 25s 185ms/step - loss: 0.6608 - accuracy: 0.7637 - val_loss: 0.4088 - val_accuracy: 0.8653
Epoch 8/10
135/135 [==============================] - 26s 190ms/step - loss: 0.5224 - accuracy: 0.8263 - val_loss: 0.3038 - val_accuracy: 0.9119
Epoch 9/10
135/135 [==============================] - 25s 182ms/step - loss: 0.4086 - accuracy: 0.8624 - val_loss: 0.2595 - val_accuracy: 0.9309
Epoch 10/10
135/135 [==============================] - 25s 182ms/step - loss: 0.3607 - accuracy: 0.8795 - val_loss: 0.2305 - val_accuracy: 0.9499
11/11 [==============================] - 2s 160ms/step
```

Figura 10: Resultado com as imagens originais do dataset

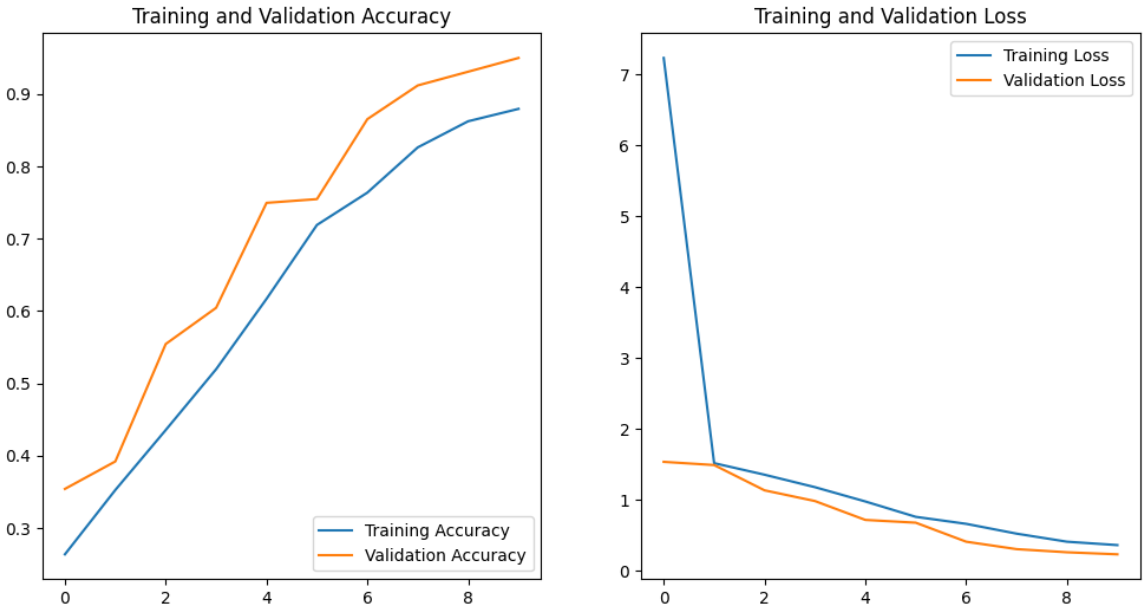


Figura 11: Gráfico do treinamento e validação

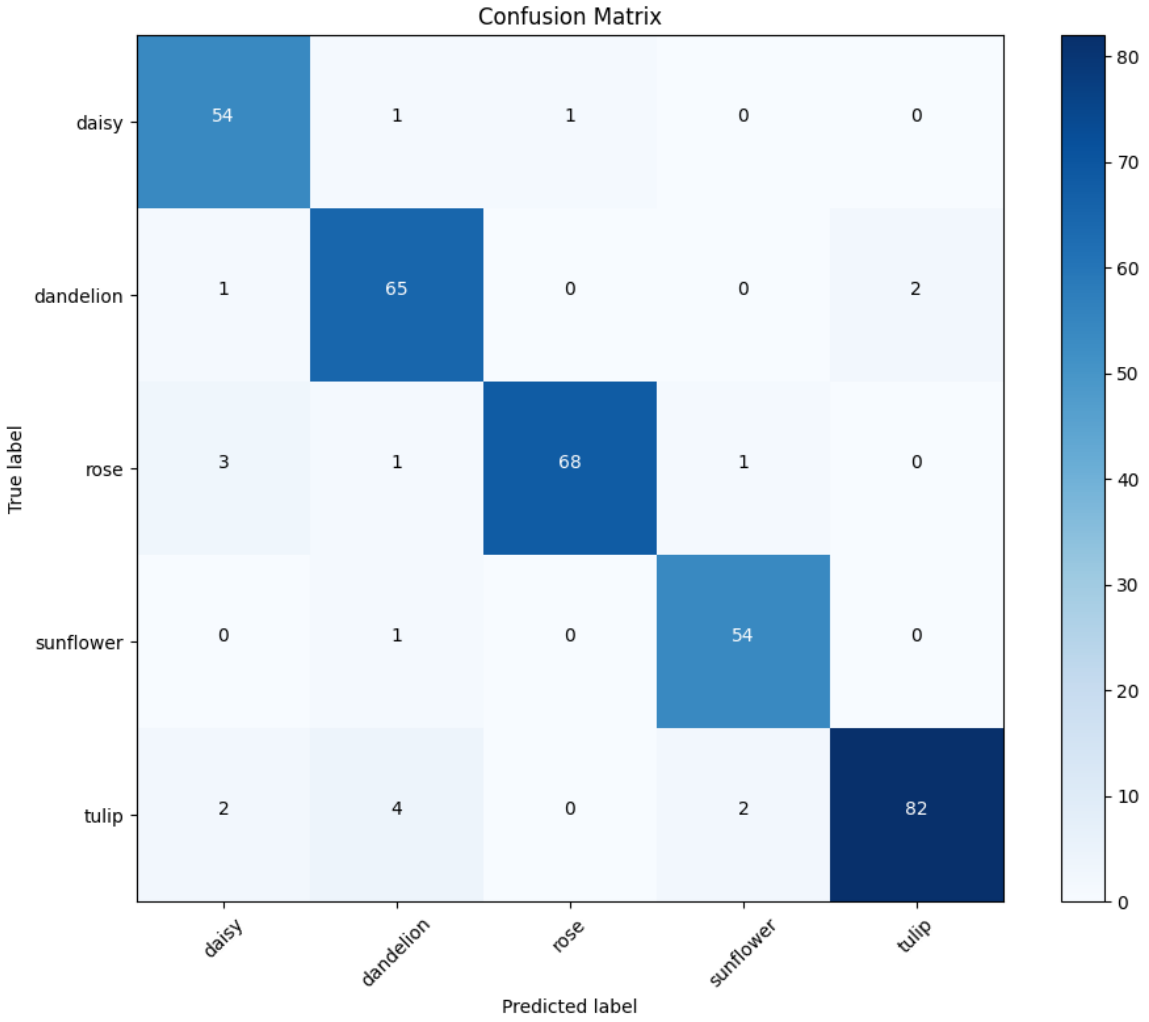


Figura 12: Matriz de confusão

# 6 Conclusão

Como descrevemos anteriormente, nosso melhor resultado foi alcançado executando o nosso código apenas com as imagens baixadas do dataset público. Tivemos um ótimo desempenho com acurácia maior de 90% para todas a classes e loss caíndo de 1.53 para 0.23.

Como melhoria futura poderíamos estudar melhor os parâmetros de normalização e resize bem como aplicar outros métodos de treinamento ou adicionar mais camadas de convolução para que, dessa forma, os números de acurácia e loss sejam melhores do que os números obtidos quando executamos com as imagens pré-processadas.