Relatório de Inteligência Artificial e Visão Computacional

Walter Tiago Bezerra Neto
walter.tiago.neto@usp.br

Treinamento e Inferência para Classificação de Imagens

Processo Seletivo Xmobots

São Carlos
26 de maio de 2023

SUMÁRIO

1 Introdução

Este relatório apresenta o desenvolvimento de um modelo de inteligência artificial (IA) para a identificação de árvores em imagens de satélite com dados provenientes e cedidos para o processo seletivo realizado pela Xmobots. O objetivo principal é fornecer uma solução que, consequentemente, possa ser utilizada durante missões de pulverização realizadas por drones, a fim de evitar colisões com obstáculos e prevenir acidentes.

A Xmobots é uma empresa brasileira especializada no desenvolvimento e fabricação de drones de alta performance para aplicações comerciais e governamentais. Fundada em 2007, a empresa tem se destacado no setor devido à sua tecnologia inovadora e soluções avançadas.

Com sede em São Carlos, estado de São Paulo, a Xmobots tem como missão fornecer soluções aéreas eficientes e seguras para diversas indústrias e setores, incluindo agricultura, topografia, mapeamento, inspeção de infraestruturas e segurança pública. Seus drones são projetados para atender às necessidades específicas de cada setor, oferecendo precisão, confiabilidade e facilidade de uso.

A Xmobots tem sido reconhecida tanto no mercado nacional quanto internacional. Seus drones são utilizados por empresas e instituições renomadas, como o Instituto Brasileiro de Geografia e Estatística (IBGE) e o Instituto Nacional de Colonização e Reforma Agrária (INCRA). Além disso, a empresa já exportou seus produtos para outros países como Estados Unidos, Canadá, México, Colômbia, Argentina, Alemanha, França e Austrália.

Além dos drones de mapeamento, como o Arator, e os drones de vigilância como o Nauru 500 e o Nauru 1000 a Xmobots realiza a revenda de drones de pulverização da DJI, como o DJI T40, que realiza voos a uma altitude referencial de 3 metros.

Durante o desenvolvimento adota-se um *dataset* como base de treinamento, com imagens de 50x50 pixels em três canais – RGB. O objetivo é o treinamento de um modelo com a capacidade de classificação correta para imagens, atribuindo classes específicas de "árvore"ou "solo". Durante o processo de desenvolvimento, explora-se o uso de um modelo pré-treinado – **VGG19** – como base e modificações para implementação de uma rede neural específica para as tarefas e, consequentemente, a missão estabelecida.

Este relatório detalha as etapas do desenvolvimento, as escolhas metodológicas adotadas e os resultados alcançados. Além disso, são fornecidos os *scripts* utilizados para o treinamento do modelo e para a inferência da rede, bem como este link para o projeto no GitHub, onde todo o código-fonte está disponível para referência e reprodução.

2 Metodologia

Durante a metodologia, diversas bibliotecas foram utilizadas, junto ao Google Colab, as bibliotecas requeridas e suas respectivas versões, encontram-se em requirements.txt. Contudo, o comando **!pip** realiza a tarefa de instalação da biblioteca específica, como exemplo demonstra-se com a extensão **tensorflow-addons**.

```
!pip install tensorflow_addons
```

Algumas bibliotecas foram selecionadas para o desenvolvimento deste modelo, estas seguem em código e descritas brevemente.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import pathlib as pl
import tensorflow as tf
import datetime
import shutil
import tensorflow_addons as tfa
import tensorflow.keras as tfk
import sklearn
from sklearn.metrics import confusion_matrix, auc, roc_curve
```

Estes módulos e bibliotecas possuem suas peculiaridades e funções aplicáveis ao modelo. Uma descrição sucinta segue:

- **`numpy`**: fornece suporte para operações matemáticas e manipulação de arrays multidimensionais
- **`matplotlib.pyplot`**: biblioteca para criação de gráficos e visualizações, permitindo a geração de gráficos, histogramas, dispersões e outros tipos de plotagens
- **`datetime`**: biblioteca para trabalhar com datas e horas em Python, permitindo manipulação, formatação e cálculos com facilidade, no modelo é útil para versionar os modelos
- **`pandas`**: biblioteca de análise de dados que oferece estruturas de dados flexíveis e eficientes para manipular e analisar conjuntos de dados
- **`pathlib`**: fornece classes e métodos para lidar com caminhos de arquivos e diretórios de forma mais intuitiva e portável, é uma versão atual e substituta da biblioteca **`os`**
- **`shutil`**: fornece um conjunto de utilitários para operações de manipulação de arquivos e diretórios, incluindo cópia, exclusão, movimentação e renomeação, junto ao **`pathlib`** é útil para criação de diretórios e arquivos

- **tensorflow**: esta plataforma de código aberto para machine learning oferece uma ampla gama de ferramentas e recursos para criar e treinar modelos de aprendizado de máquina
- **tensorflow-addons**: uma biblioteca complementar ao **tensorflow** que oferece implementações de operações e camadas adicionais, como otimizadores avançados e funções de ativação, no caso deste modelo, usa-se para a métrica **f1-score**
- **tensorflow.keras**: camada de alto nível do **tensorflow** para construção e treinamento de redes neurais
- **sklearn**: também conhecida como scikit-learn, é uma biblioteca amplamente utilizada para aprendizado de máquina, com ferramentas para pré-processamento de dados, treinamento de modelos e avaliação de desempenho.

Uma função auxiliar foi implementada para versionar os modelos de forma simples, onde há o retorno de uma **string** com o tempo preciso demarcado.

```
def timestamp():
  return datetime.datetime.now().strftime("%Y%m%d%H%M%S")
```

2.1 Pré-processamento de Dados

O *dataset* cedido pela Xmobots foi armazenado no GitHub em uma pasta de mesmo nome dentro de uma pasta principal denominada **ai-images-dataset**. Para obtenção direta no Colab, aplica-se o comando **clone** direto no repositório pertencente ao autor.

```
!git clone https://github.com/walter-tiago/ai-images-dataset.git
```

Os diretórios que são desenvolvidos pelo próprio código são armazenados em uma pasta **temporary**.

```
dataset_dir = pl.Path("/content/ai-images-dataset/dataset")
temp_dir = pl.Path("/content/ai-images-dataset/temporary")
model_dir = temp_dir / "models"
train_dir = temp_dir / "train"
log_dir = temp_dir / "logs"
dataset_dir.mkdir(parents = True, exist_ok = True)
model_dir.mkdir(parents = True, exist_ok = True)
train_dir.mkdir(parents = True, exist_ok = True)
log_dir.mkdir(parents = True, exist_ok = True)
```

Os valores de `input` são determinados e descritos, respectivamente:

```
image_size = (50, 50)
classes = ["soil", "tree"]
batch_size = 128
epochs = 256
learning_rate = 0.001
regularization_rate = 0.01
num_ref_layer = 3
dropout_rate = 0.5
validation_split = 0.3
```

Nestas entradas temos variáveis comuns para classificadores de imagens. O **batch_size** (tamanho do lote) indica o número de amostras de treinamento que serão usadas em cada iteração do treinamento. Neste caso, a cada iteração, 128 amostras serão usadas para ajustar os pesos do modelo. Este processo ocorre enquanto o conjunto é percorrido 256 (**epochs**) vezes no total. Já o **learning_rate** é uma taxa que controla o tamanho do passo que o algoritmo dá ao ajustar os pesos do modelo durante o treinamento. Uma taxa de aprendizado menor pode levar a ajustes mais lentos, mas mais precisos, enquanto uma taxa maior pode levar a ajustes mais rápidos, mas menos precisos.

A regularização setada por **regularization_rate** é usada para evitar o **overfitting**, que ocorre quando o modelo se ajusta muito bem aos dados de treinamento, mas não generaliza bem para novos dados. Essa taxa especifica a força da regularização, com um valor maior indicando uma regularização mais forte. O **num_ref_layer** indica o número de iterações para implementar camadas de **Conv2D**, **MaxPooling2D** e camadas **Dense** no modelo. Esta variável é específica para o modelo aqui em desenvolvimento. Para este modelo, vamos implementar 3 vezes, na discussão do modelo na subseção 2.2, mostra-se 1 modelo base e 18 camadas.

O **dropout** aqui setado por **dropout_rate** é uma técnica de regularização que ajuda a evitar o **overfitting**, desligando aleatoriamente uma determinada razão de neurônios durante o treinamento. E, por fim, o **validation_split** indica a fração do conjunto de treinamento que será usada como conjunto de validação. Durante o treinamento, o modelo será avaliado nesse conjunto de validação para monitorar seu desempenho. Nesse caso, 30% do conjunto de treinamento será usado como conjunto de validação.

Em sequência temos dados não mutáveis, que definem o modo de classes, a quantidade de canais (3 para RGB), e as dimensões da imagem.

```
class_mode = "categorical"
num_classes = len(classes)
channels = 3
image_shape = image_size + (channels,)
for class_name in classes:
  class_dir = train_dir / class_name
  class_dir.mkdir(parents=True, exist_ok=True)
  for local_file in dataset_dir.iterdir():
```

```
    local_file_name = local_file.name
    if local_file.is_file() and class_name in local_file_name:
      shutil.copy(local_file, class_dir / local_file.name)
  print(f'{class_name.title()}: {len(list(class_dir.iterdir()))} samples
    .')
```

É notável que mesmo com apenas duas classes, seleciona-se o modelo categórico, visto que para futuras implementações pode-se aumentar o número de classes. As iterações, em sequência, separam os dados em classes diferentes e as quantificam, onde descobre-se que temos: **4448 soil samples** e **2224 tree samples**. Logo, têm-se que visar o balanço dos inputs, já que não é interessante o modelo treinar com uma classe com tal desbalanceamento, o que pode ocasionar um **overfitting** para solos, já que identificará uma classe bem mais que outra. Para pré–processamento e normalização das entradas adota-se uma aplicação da VGG19.

```
datagen = tfk.preprocessing.image.ImageDataGenerator(
    preprocessing_function = tfk.applications.vgg19.preprocess_input,
    width_shift_range = 0.2,
    height_shift_range = 0.2,
    zoom_range = 0.1,
    validation_split = validation_split)
train_generator = datagen.flow_from_directory(
    train_dir,
    target_size = image_size,
    batch_size = batch_size,
    class_mode = class_mode,
    subset='training')
validation_generator = datagen.flow_from_directory(
    train_dir,
    target_size = image_size,
    batch_size = batch_size,
    class_mode = class_mode,
    subset='validation',
    shuffle=False)
class_weights = sklearn.utils.class_weight.compute_class_weight(
                'balanced',
                classes = np.unique(train_generator.classes),
                y = train_generator.classes)
print(f"\nClass weights: {classes} = {class_weights}")
```

Após essa etapa de balanceamento, é notável que alguns parâmetros foram selecionados para transformar o banco de dados, visto que aplica-se **shift range** de 20% e **zoom range** de 10%, isto para facilitar a identificação da classe **tree**. Os pesos balanceados e computados pelo **sklearn** são **soil: 0.75, tree: 1.50**.

2.2 Treinamento do Modelo

O modelo possui como base a rede pré-treinada VGG19, uma rede neural convolucional desenvolvida pelo Visual Geometry Group (VGG) da Universidade de Oxford. Com 19 camadas convolucionais e totalmente conectadas, ela se destaca por sua simplicidade e uniformidade. A VGG19 é conhecida por seu desempenho impressionante em tarefas de classificação de imagens. Adota-se o conjunto de dados **`imagenet`**, que consiste em mais de um milhão de imagens rotuladas em 1000 categorias diferentes e é frequentemente usado para o treinamento de redes neurais convolucionais. E seta-se o parâmetro **`trainable`** como **`False`**, para congelar os pesos durante o treinamento. Além disso, adota-se a classe **`Sequential`** para construção de camadas sequenciais.

```
base_model = tfk.applications.vgg19.VGG19(
    weights = "imagenet",
    include_top = False,
    input_shape = image_shape
    )
base_model.trainable = False
model = tfk.models.Sequential()
model.add(base_model)
```

Então inicia-se um loop que adiciona camadas sequenciais. Em cada iteração, três camadas convolucionais são adicionadas ao modelo. Cada camada convolucional possui um número de filtros definidos crescentemente. O tamanho do kernel é fixado e a função de ativação é a **`ReLU`**. Todas as camadas convolucionais têm preenchimento **`padding = same`** para preservar a dimensão espacial das características pré-pooling.

Após cada grupo de três camadas convolucionais, é adicionada uma camada de max pooling com tamanho de pooling (2, 2) e preenchimento **`same`** novamente. Essa camada reduz a dimensionalidade espacial das características extraídas. Então finaliza-se o loop, onde é adicionada uma camada de **`dropout`** para regularização após as camadas convolucionais. A taxa de dropout é metade do **`dropout`** escolhido inicialmente. E a camada **`Flatten`** transforma o tensor de saída das camadas convolucionais em um vetor unidimensional, preparando-o para ser alimentado nas camadas densas.

```
counter_layer = 0
while counter_layer < num_ref_layer:
  model.add(
      tfk.layers.Conv2D(
          filters = batch_size / 2**(num_ref_layer - counter_layer),
          kernel_size = (3, 3),
          activation='relu',
          padding='same',
          input_shape = image_shape,))
  model.add(
      tfk.layers.Conv2D(
```

```
            filters = batch_size / 2**(num_ref_layer - counter_layer),
            kernel_size = (3, 3),
            activation='relu',
            padding='same',
            input_shape = image_shape,))
    model.add(
      tfk.layers.Conv2D(
          filters = batch_size / 2**(num_ref_layer - counter_layer),
          kernel_size = (3, 3),
          activation='relu',
          padding='same',
          input_shape = image_shape,))
    model.add(
        tfk.layers.MaxPooling2D(
            pool_size = (2, 2),
            padding='same',
            ))
    counter_layer += 1
model.add(tfk.layers.Dropout(dropout_rate / 2))
model.add(tfk.layers.Flatten())
```

Ao iniciar as camadas densas o script apresenta outro loop. Em cada iteração, uma camada **Dense** é adicionada seguida de uma camada de **dropout**. Cada camada tem um número de unidades definido de forma decrescente e usa a função de ativação **ReLU**. A regularização **L2** com taxa **regularization_rate** é aplicada em cada camada **Dense**. A regularização **L2** adiciona um termo de penalidade proporcional ao quadrado dos pesos ao cálculo da função de perda durante o treinamento. A regularização **L2** tem a vantagem de incentivar os pesos a serem distribuídos de forma mais uniforme e diminuir o impacto de pesos grandes, evitando assim o **overfitting**.

```
counter_layer = 0
while counter_layer < num_ref_layer - 1:
    model.add(
        tfk.layers.Dense(
            batch_size / 2**counter_layer,
            activation='relu',
            activity_regularizer = tfk.regularizers.l2(regularization_rate
     )
            ))
    model.add(
          tfk.layers.Dropout(dropout_rate)
          )
    counter_layer += 1
```

A camada de saída é uma camada **Dense** que depende do número de classes. Se o número de classes for menor ou igual a 2, a função de ativação é **sigmoid** para problemas

de classificação binária. Caso contrário, a função de ativação é **softmax** para problemas de classificação multiclasse.

```
if num_classes <= 2:
  model.add(
      tfk.layers.Dense(num_classes, activation='sigmoid')
      )
else:
  model.add(
      tfk.layers.Dense(num_classes, activation='softmax')
      )
```

Sobre a compilação do modelo tem-se:

```
# Model compiler:
optimizer = tfk.optimizers.Adam(learning_rate=learning_rate)
metrics = [
    'Accuracy',
    'AUC',
    'Precision',
    'Recall', tfa.metrics.F1Score(num_classes=num_classes, average='
   micro'),
]
model.compile(
    optimizer=optimizer,
    loss=class_mode + '_crossentropy',
    metrics=metrics,
)
tensorboard_callback = tfk.callbacks.TensorBoard(log_dir=log_dir,
   write_graph=True)
```

Onde seleciona-se o otimizador Adam com uma taxa de aprendizado **learning_rate**). O otimizador Adam é comumente usado em redes neurais para atualizar os pesos durante o treinamento, ajustando-os de acordo com o gradiente calculado.

Sobre as métricas que são definidas para avaliar o desempenho do modelo durante o treinamento e validação, inclui-se **Accuracy**, **AUC**, **Precision**, **Recall** e **F1Score**, esta que considera a média **micro** para calcular sua pontuação. A função de perda é **categorical_crossentropy**. E, por fim, é criado um callback do **TensorBoard**, que será usado durante o treinamento para registrar métricas e visualizar o modelo no **TensorBoard**. Contudo, o **Tensorboard** não possui uma boa interação com o Google Colab, então as curvas apresentadas, foram implementadas no script.

Inicia-se o treinamento do modelo em:

```
history = model.fit(
```

```
            train_generator,
            epochs = epochs,
            steps_per_epoch = train_generator.samples // batch_size,
            validation_data = validation_generator,
            validation_steps = validation_generator.samples //
    batch_size,
            class_weight = dict(enumerate(class_weights)),
            callbacks = [tensorboard_callback]
            )
local_time = timestamp()
local_model = model_dir / f"model_{local_time}.h5"
local_weight_model = model_dir / f"model_{local_time}_weights.h5"
model.save(local_model)
model.save_weights(local_weight_model)
```

Estas etapas concluem o treinamento do modelo e salvam os resultados para uso posterior, permitindo a reutilização do modelo treinado ou a análise dos resultados obtidos durante o treinamento.

2.3 Inferência do Modelo

Com a conclusão do treinamento do modelo, este foi salvo e pode ser retreinado, inicia–se então a inferência, onde construímos nossas métricas, estas que serão discutidas na seção 3.

```
model = tfk.models.load_model(local_model)
metrics_base = ['loss', 'Accuracy', 'auc', 'precision', 'recall', '
    f1_score']
metrics = history.history
for metric_name in metrics_base:
    local_train_values = metrics[metric_name]
    local_val_values = metrics['val_' + metric_name]
    plt.figure(figsize=(12, 8))
    plt.plot(local_train_values, label='Train ' + metric_name.title(),
    lw = 0.85)
    plt.plot(local_val_values, label='Validation ' + metric_name.title()
    , lw = 0.85)
    plt.xlabel('Epochs')
    plt.ylabel(metric_name.title() + ' Value')
    plt.grid()
    plt.legend(edgecolor = 'k')
    plt.show()
```

Após a elaboração das curvas de métricas, inicia-se a validação de dados. Onde encontram-se os valores preditos, a implementação da curva `ROC` e da `confusion matrix`. A curva `ROC` mostra a relação entre a taxa de verdadeiros positivos (True Positive Rate - TPR) e a taxa de falsos

positivos (False Positive Rate - FPR) em diferentes pontos de corte do limiar de decisão do modelo. Já a matriz de confusão fornece uma visão geral das previsões corretas e incorretas do modelo para cada classe.

```
y_pred_prob = model.predict(validation_generator)
y_pred = np.argmax(y_pred_prob, axis=1)
y_true = validation_generator.classes
classes = validation_generator.class_indices
cm = confusion_matrix(y_true, y_pred)
fpr, tpr, thresholds = roc_curve(y_true, y_pred)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC curve, area = {np.round(roc_auc, 2)}', lw
    = 0.85)
plt.plot([0, 1], [0, 1], 'k--', label = 'Random Classifier', lw = 0.85)
plt.xlabel(r'False Positive Rate')
plt.ylabel(r'True Positive Rate')
plt.title(r'ROC Curve')
plt.grid()
plt.legend(loc='lower right', edgecolor = 'k')
plt.show()
plt.figure(figsize=(8, 8))
ax = sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
   xticklabels=classes, yticklabels=classes)
ax.set_frame_on(True)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
data = {
  'Filename': validation_generator.filenames,
  'Predicted Class': [list(classes.keys())[list(classes.values()).index(
   pred)] for pred in y_pred]
}
df = pd.DataFrame(data)
df['Filename'] = df['Filename'].str.replace('soil/', '').str.replace('
   tree/', '')
df = df.sort_values(by='Filename')
df.to_csv(str(csv_dir) + f'/validation_{timestamp()}.csv', index=False)
```

É notável que os dados de validação também são salvos em um arquivo `.csv`, caso se queira analisá-lo. Em sequência, faz-se a inferência do **`dataset`** inicial.

```
y_true, y_pred = [], []
for image_path in dataset_dir.glob("*.png"):
    img = tfk.preprocessing.image.load_img(image_path, target_size=
```

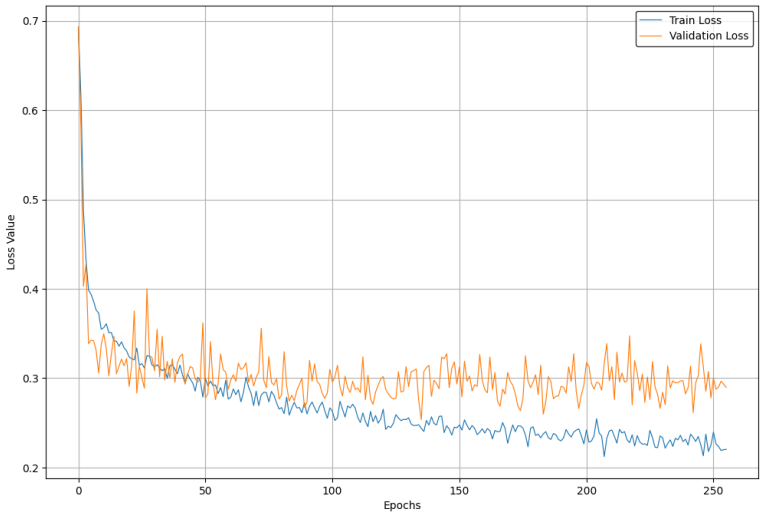
```
    image_size)
     img_array = tfk.preprocessing.image.img_to_array(img)
     img_array = tfk.applications.vgg19.preprocess_input(img_array)
     img_array = np.expand_dims(img_array, axis=0)
     prediction = model.predict(img_array)
     predicted_label = np.argmax(prediction)
     y_true.append(image_path.stem)
     y_pred.append(predicted_label)
```

Esse processo permite fazer a predição do modelo em um conjunto de imagens fora do conjunto de validação usado no treinamento. E, por fim, a inferência é salva como **csv**, e a porcentagem de acertos é calculada.

```
data = {'Filename': y_true, 'Predicted Class': y_pred}
df = pd.DataFrame(data)
df['Predicted Class'] = df['Predicted Class'].replace({0: 'soil', 1: '
    tree'})
df = df.sort_values(by='Filename')
df.to_csv(str(csv_dir) + f'/dataset_{timestamp()}.csv', index=False)
df['Predicted Class Filename'] = df['Filename'].str.split('_').str[-1].
    str.split('.').str[0]
df['compatibility'] = df['Predicted Class Filename'] == df['Predicted
    Class']
for class_name in classes.keys():
  qnt = df['Filename'].str.contains(class_name).sum()
  goalseek = (df['compatibility'] & (df['Predicted Class'] == class_name
    )).sum()
  perc = goalseek / qnt * 100
  print(f'{qnt} {class_name} samples.')
  print(f'{goalseek} {class_name} predicted samples.')
  print(f'{np.round(perc, 2)} % correct samples.')
```

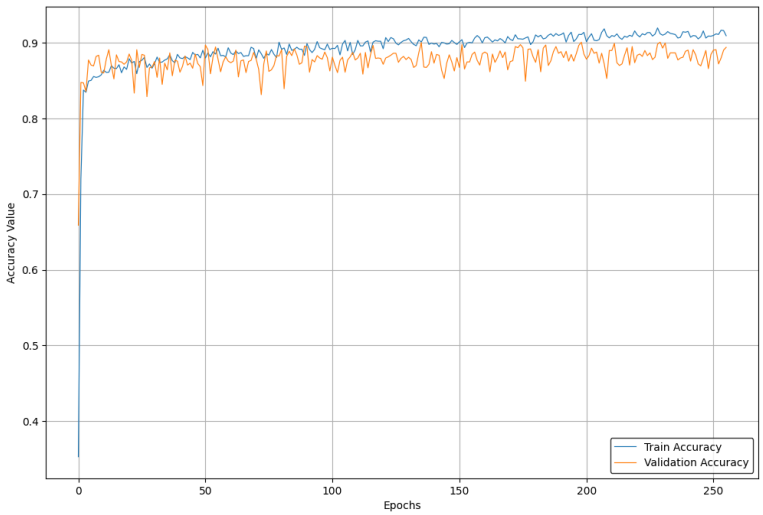
3 Resultados

Figura 1: Curva de perda do modelo



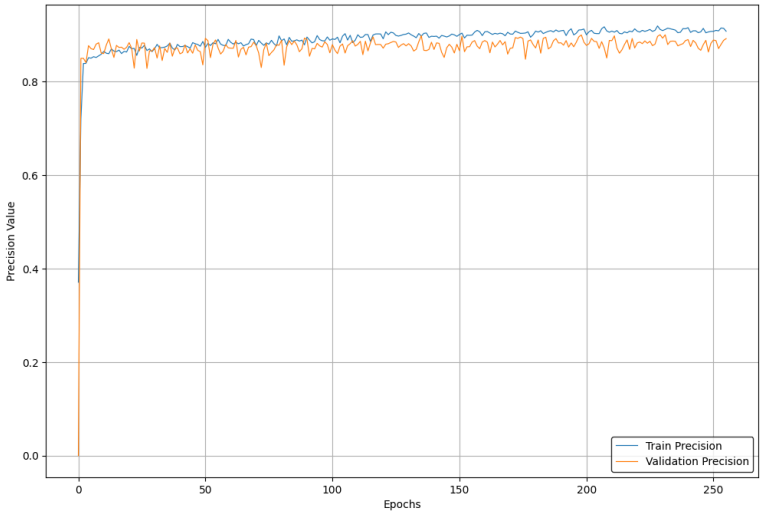
A métrica de loss (perda) mede o quão bem um modelo de aprendizado de máquina está se ajustando aos dados durante o treinamento. Ela representa a diferença entre as previsões feitas pelo modelo e os valores reais dos dados. O objetivo do modelo em minimizar a loss durante o treinamento para reduzir o erro e melhorar sua capacidade de fazer previsões precisas é visto na figura. A loss baixa indica que o modelo está fazendo boas previsões.

Figura 2: Curva de acurácia do modelo



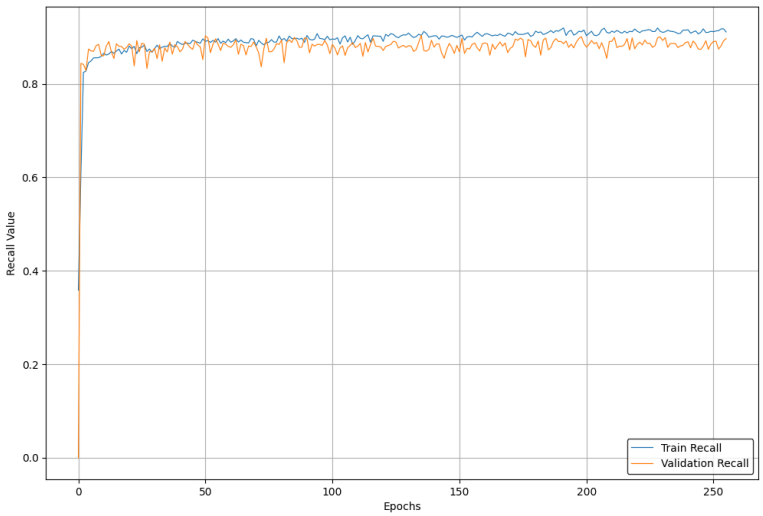
A crescência da curva de acurácia indica que o modelo está melhorando sua capacidade de classificação à medida que é treinado. Além disso, a curva de acurácia tende à estabilidade após 200 épocas, porém ainda não se estabilizou antes de um nível alto de valor, o que indica uma escolha aceitável dos hiperparâmetros.

Figura 3: Curva de precisão do modelo



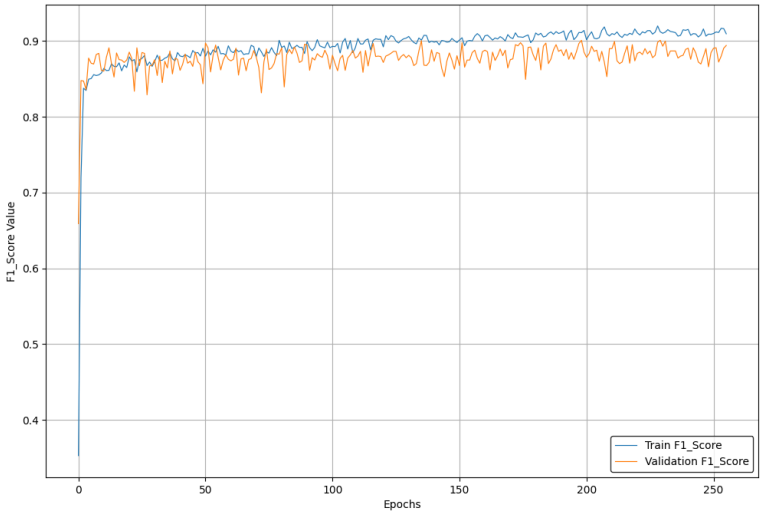
Ao analisar a curva de precisão, observa-se como o modelo lida com a taxa de falsos positivos e a capacidade de evitar classificações incorretas. O aumento na curva de precisão indica que o modelo está melhorando sua capacidade de fazer classificações corretas dos verdadeiros positivos.

Figura 4: Curva de revocação do modelo



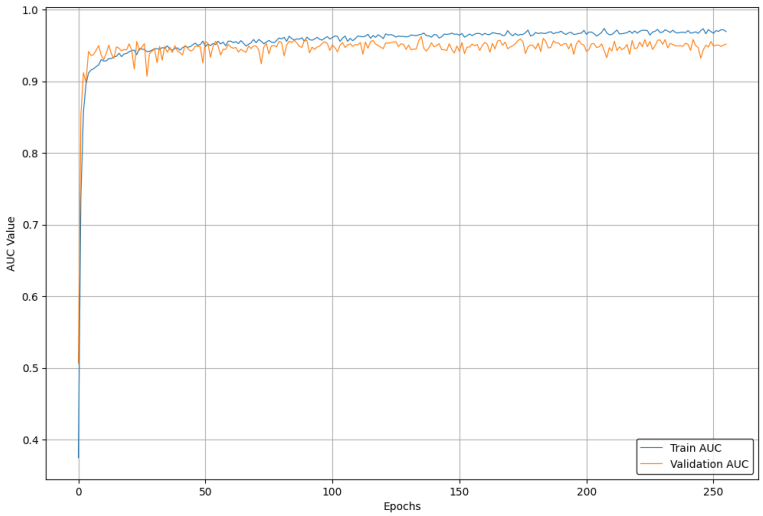
Esta métrica avalia a proporção de casos positivos reais corretamente identificados em relação ao total de casos positivos. Observa-se como o modelo está lidando com a taxa de falsos negativos e sua capacidade de identificar corretamente os casos positivos, já que o aumento na curva de `recall` indica que o modelo está melhorando sua capacidade de detectar corretamente os casos positivos, resultando em uma menor taxa de falsos negativos.

Figura 5: Curva de f1–Score do modelo



Os valores altos de F1 Score indicam um bom equilíbrio entre a precisão e o recall do modelo.

Figura 6: Curva de AUC do modelo

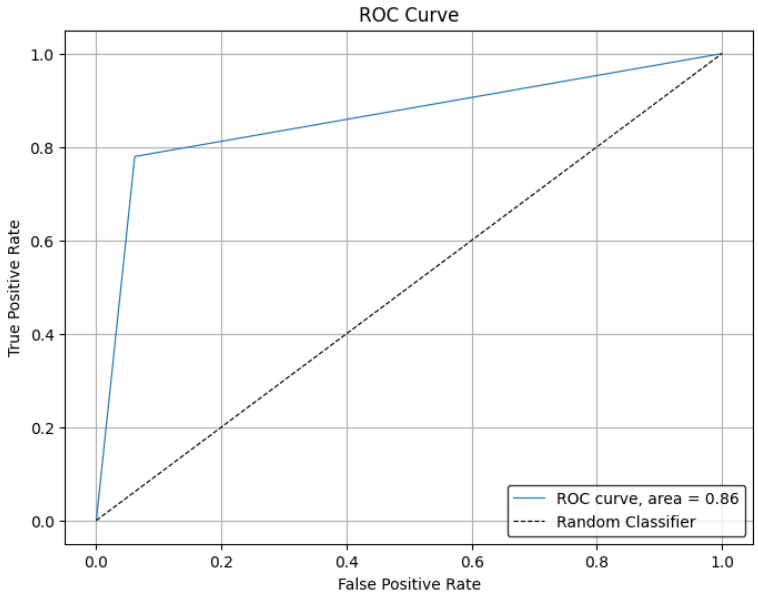


O AUC varia de 0 a 1, onde um valor de 0.5 indica um classificador aleatório e um valor próximo a 1 indica um classificador com alto poder discriminativo, porém o `overfitting` pode acontecer nestes casos. Com o AUC acima de 0.9, tem-se um bom desempenho do modelo na tarefa de classificação binária.

3.1 Validação do Modelo

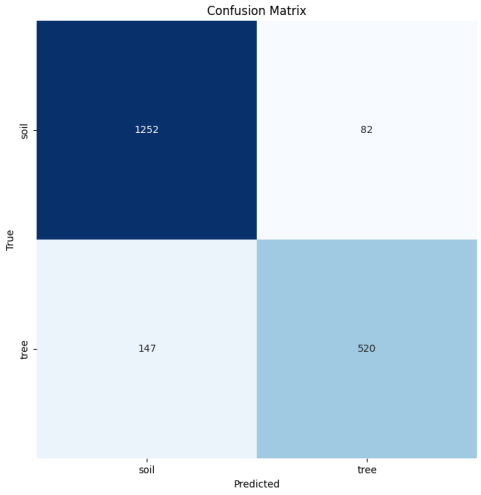
Durante a implementação do modelo, separou-se 30% dos dados para validação, adotou-se a curva ROC e uma matriz de confusão para analisar estes dados.

Figura 7: Curva ROC do modelo



A curva ilustra a relação entre a taxa de verdadeiros positivos (TPR) e a taxa de falsos positivos (FPR) de um modelo de classificação binária. Mesmo com a classificação feita apenas com um threshold, tem-se a tendência aceitável de uma curva ROC.

Figura 8: Matriz de confusão do modelo



A matriz de confusão mostra a contagem de exemplos classificados corretamente e erroneamente para cada classe em um problema de classificação. Neste caso, tem-se valores diagonais aceitáveis, onde para a classe `soil`, temos 93.85% de acertos nos dados de validação, e para a classe `tree`, temos 77.96% de acertos, o que era esperado, visto um número menor de `samples`.

4 Conclusão

Ao combinar todas as métricas da seção 3, é possível ter uma compreensão abrangente do desempenho do modelo de classificação, identificando suas forças e fraquezas. Isso permite aprimorar o modelo, ajustar thresholds de classificação e tomar decisões mais informadas em relação à tarefa de classificação.

Com o modelo preditivo aplicado ao dataset inicial, temos a quantidade de acertos para cada classe, onde para a classe `soil`, o modelo acertou 95.73%, enquanto que para a classe `tree` acertou 81.65%, e novamente, era esperado, visto o treinamento com menos `samples` desta classe. Nota–se que estes valores são aceitáveis, porém a classe `tree` pode ser melhorada, isto pode ser feito na aquisição dos dados, na arquitetura do modelo, no ajuste de hiperparâmetros e nas modificações nos inputs.

Ainda destacam–se algumas possibilidades, como comparações com modelos de pré–processamento diferentes do adotado, outros modelos bases, a diminuição das camadas de `MaxPooling2D` – visto que estas podem ocasionar algum ruído à imagens pequenas – um maior número de épocas, o retreinamento com novos dados para validação, e, uma avaliação contínua e ajustes finos do modelo.