

Inteligência Computacional

2024/2025

Projeto – Meta 3

João Pedro Silveira da Costa, N.º 2022143368

Juliana Silva Teixeira, N.º 2022143379

Índice

1. Descrição do Problema	3
2. Descrição das Metodologias Utilizadas	4
3. Arquitetura de Código	5
4. Aplicação	9
5. Descrição da Implementação dos Algoritmos	10
6. Análise de Resultados	11
7. Conclusões	13
8. Bibliografia	14

Descrição do Problema

O objetivo do projeto é desenvolver um modelo de classificação de imagens de sinais de trânsito para aplicação prática em sistemas de monitoramento e assistência ao condutor. O objetivo é criar uma solução robusta capaz de identificar e classificar corretamente diferentes sinais de trânsito a partir de imagens capturadas em tempo real, utilizando uma rede neural convolucional (CNN).

O desafio principal está na necessidade de criar um modelo que seja capaz de processar imagens de diferentes condições de iluminação, ângulos e qualidade, garantindo alta precisão e eficiência na classificação. As imagens de sinais de trânsito podem variar significativamente em termos de características visuais, o que torna a tarefa de classificação complexa e exige o uso de técnicas avançadas de machine learning.

Outro aspecto importante deste projeto é a otimização do desempenho. Para alcançar um modelo eficaz e eficiente, será necessário ajustar cuidadosamente os hiperparâmetros e testar diferentes estruturas de rede. O objetivo é maximizar a precisão do modelo.

A análise dos resultados de classificação será realizada utilizando métricas padrão como accuracy, precision, recall, f1-score, além da AUC, para avaliar a performance do modelo e garantir que ele seja capaz de classificar corretamente os sinais em diferentes cenários, com um mínimo de erros.

Este modelo será crucial para integrar soluções de reconhecimento de sinais de trânsito em sistemas de assistência ao condutor, como sistemas de condução autônoma e sistemas de segurança avançada, proporcionando maior segurança e eficiência nas estradas.

Descrição das Metodologias Utilizadas

Modelo Simples:

Uma abordagem direta, que não utiliza técnicas de otimização. Este programa foi treinado em um dataset com 5 classes, com 500 amostras cada.

Otimização por Random Search:

O Random Search é uma técnica de otimização de hiperparâmetros em modelos de machine learning. Escolhe combinações aleatórias dentro de um intervalo definido pelo utilizador.

Arquitetura de Código

O código começa com a importação das várias bibliotecas necessárias para a execução do programa.

```
import pickle
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, precision_score, f1_score, roc_auc_score, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import img_to_array, array_to_img
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam
```

Inicialmente, as imagens são carregadas a partir de um arquivo no formato pickle, que contém os dados organizados em arrays de imagens, rótulos, tamanhos originais e coordenadas dos limites de corte. A função `preprocess_data` é utilizada para processar esses dados, realizando diversas etapas essenciais para a preparação das imagens.

As imagens são filtradas com base em 5 classes, garantindo que apenas as categorias de interesse sejam mantidas. Além disso, para evitar um desequilíbrio no conjunto de dados, usamos 500 amostras para cada classe. Esse processo assegura um conjunto de dados balanceado para o treinamento do modelo.

```
def preprocess_data(file, target_size=(224, 224)):
    with open(file, 'rb') as f:
        data = pickle.load(f, encoding='latin1')
        x = data['features'].astype(np.float32)
        y = data['labels']
        sizes = data['sizes']
        coords = data['coords']

    df = pd.DataFrame({
        'features': list(x),
        'labels': y,
        'sizes': list(sizes),
        'coords': list(coords)
    })

    df_filtered = df[df['labels'].isin(classes)]
    df_limited = df_filtered.groupby('labels').apply(lambda x: x.sample(n=500, random_state=42)).reset_index(drop=True)
```

Cada imagem passa por uma etapa de corte utilizando as coordenadas fornecidas no arquivo. Para isso, as dimensões das imagens originais são verificadas, garantindo que as coordenadas de corte estejam dentro dos limites válidos. Após o corte, as imagens resultantes são redimensionadas para o tamanho 224x224 pixels. Posteriormente, as imagens são normalizadas dividindo os valores dos pixels por 255, o que as ajusta para a faixa de valores entre 0 e 1, facilitando o learning rate do modelo.

Os rótulos das imagens são mapeados para valores numéricos, utilizando um dicionário de mapeamento (`class_mapping`) que converte os nomes das classes em índices. Isso é necessário, pois modelos de machine learning operam exclusivamente com dados numéricos durante o treinamento e a classificação.

No final do processo, a função retorna uma lista de imagens pré-processadas e os seus respectivos rótulos numéricos.

```

# Pré-processamento
cropped_images = []
mapped_labels = []

for i in range(len(df_limited)):
    img = df_limited['features'].iloc[i]          # Imagem original
    x1, y1, x2, y2 = df_limited['coords'].iloc[i]
    width, height = df_limited['sizes'].iloc[i]    # Dimensões originais da imagem

    # Verificar que as coordenadas limitadoras estão dentro dos limites da imagem
    if x1 < 0: x1 = 0
    if y1 < 0: y1 = 0
    if x2 > width: x2 = width
    if y2 > height: y2 = height

    # Cortar a imagem pelos limites
    cropped_img = img[y1:y2, x1:x2, :]

    # Redimensionar a imagem para o tamanho esperado pela arquitetura
    img_resized = array_to_img(cropped_img).resize(target_size)
    img_resized = img_to_array(img_resized) / 255.0 # Normalizar

    # Guardar a imagem pré-processada
    cropped_images.append(img_resized)

    original_label = df_limited['Labels'].iloc[i]
    mapped_labels.append(class_mapping[original_label])

# Converter listas para numpy arrays
cropped_images = np.array(cropped_images)
mapped_labels = np.array(mapped_labels)

```

O conjunto de dados é dividido em treino e teste. O treino é usado para ajustar o modelo, enquanto o teste é usado para avaliar o desempenho do modelo durante o treino. A divisão é feita de forma a manter a distribuição de classes uniforme entre os conjuntos de dados.

```

# Dividir o conjunto de dados em treino e teste
X_train, X_test, y_train, y_test = train_test_split(cropped_images, mapped_labels, test_size=0.2, random_state=42, stratify=mapped_labels)

return X_train, X_test, y_train, y_test

```

Um modelo MobileNetV2 pré-treinado é carregado com os pesos do ImageNet e “include_top = False” é utilizado para descartar as camadas finais de classificação originais. Esta abordagem permite personalizar as camadas superiores de acordo com o problema de classificação. As camadas do modelo base são congeladas, garantindo que os pesos otimizados no ImageNet não sejam alterados durante o treino. O modelo foi compilado utilizando o otimizador Adam, com um learning rate de 0,001. A função de perda utilizada foi a sparse_categorical_crossentropy.

O modelo treinado foi guardado no formato .h5, possibilitando a sua reutilização.

```

def create_pretrained_model(input_shape, num_classes, learning_rate=0.001, dense_units=128):
    # Carregar o modelo MobileNetV2 pré-treinado no ImageNet
    base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=input_shape)

    # Congelar os pesos da base
    base_model.trainable = False

    # Adicionar camadas customizadas no topo (top layers)
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(dense_units, activation='relu')(x)
    x = Dropout(0.5)(x) # Regularização com dropout
    x = Dense(dense_units // 2, activation='relu')(x)
    x = Dropout(0.5)(x) # Regularização com dropout
    predictions = Dense(num_classes, activation='softmax')(x)

    # Criar o modelo final
    model = Model(inputs=base_model.input, outputs=predictions)

    # Compilar o modelo
    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    return model

```

```

input_shape = (224, 224, 3) # Imagem 32x32 com 3 canais de cor (RGB)
num_classes = len(classes)

# Definir hiper-parâmetros
learning_rate = 0.001
dense_units = 256

pretrained_model = create_pretrained_model(input_shape, num_classes, learning_rate, dense_units)

pretrained_model.summary()

# Treinar o modelo
history = pretrained_model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test), batch_size=32)

pretrained_model.save("traffic_sign_model.h5")

```

O modelo treinado foi avaliado em um conjunto de testes. As métricas de avaliação foram a accuracy, recall, precision e f-measure e a AUC para cada classe. Além disso, uma matriz de confusão é criada, permitindo uma análise visual dos erros de classificação.

```

# Avaliar o modelo
test_loss, test_acc = pretrained_model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc:.4f}')

y_probs = pretrained_model.predict(X_test)
y_pred = np.argmax(y_probs, axis=1)

# Calcular as Métricas
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred, average='weighted')
precision = precision_score(y_test, y_pred, average='weighted')
fmeasure = f1_score(y_test, y_pred, average='weighted')

# Exibir as métricas finais
print(f"Accuracy: {accuracy:.4f}")
print(f"Recall: {recall:.4f}")
print(f"Precision: {precision:.4f}")
print(f"F-measure: {fmeasure:.4f}")

# Calcular o AUC por classe
auc_scores = []
for i in range(num_classes):
    auc = roc_auc_score((y_test == i).astype(int), y_probs[:, i])
    auc_scores.append(auc)
    print(f"AUC for class {classes[i]}: {auc:.4f}")

```

Os resultados experimentais, incluindo as métricas de desempenho, AUC por classe, matriz de confusão e os detalhes dos conjuntos de dados, foram organizados em DataFrames do Pandas e exportados para um ficheiro Excel.

```

# Criar dataframes para salvar no Excel
df_results = pd.DataFrame({
    'Metric': ['Accuracy', 'Recall', 'Precision', 'F-measure'],
    'Value': [accuracy, recall, precision, fmeasure]
})

df_auc_scores = pd.DataFrame({
    'Class': classes,
    'AUC': auc_scores
})

df_confusion_mtx = pd.DataFrame(
    confusion_matrix(y_test, y_pred),
    index=[f"True_{cls}" for cls in classes],
    columns=[f"Pred_{cls}" for cls in classes]
)

# Criar um relatório de classificação detalhado
class_report = classification_report(y_test, y_pred, target_names=[f"Class_{cls}" for cls in classes], output_dict=True)
df_class_report = pd.DataFrame(class_report).transpose()

# Adicionar o tamanho das classes no conjunto de dados
df_shapes = pd.DataFrame({
    "Dataset": ["Training", "Testing"],
    "Samples": [X_train.shape[0], X_test.shape[0]],
    "Features Shape": [X_train.shape[1:], X_test.shape[1:]]
})

```

```

# Matriz Confusão
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Salvar todos os dados em um único arquivo Excel
with pd.ExcelWriter("model_results.xlsx") as writer:
    df_shapes.to_excel(writer, sheet_name='Formas dos Conjuntos', index=False)
    df_results.to_excel(writer, sheet_name='Resultados Otimizacao', index=False)
    df_auc_scores.to_excel(writer, sheet_name='AUC por Classe', index=False)
    df_confusion_mtx.to_excel(writer, sheet_name='Matriz de Confusão')
    df_class_report.to_excel(writer, sheet_name='Relatório de Classificação')

```

```

# Plotar curvas de treino e validação
plt.figure(figsize=(12, 5))

# Curva de accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Curva de perda
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

```


Aplicação

Para facilitar o uso do modelo de classificação de sinais de trânsito, foi desenvolvida uma interface gráfica com a biblioteca Streamlit. A interface permite o carregamento de imagens e mostra os resultados da previsão de forma intuitiva.

O modelo treinado, guardado como `traffic_sign_model.h5`, é carregado utilizando a função `load_model` com `@st.cache_resource` para melhorar o desempenho ao reutilizar o modelo já carregado.

```
import streamlit as st
from PIL import Image
import numpy as np
import tensorflow as tf

# Carregar o modelo treinado (garanta que o modelo está no mesmo diretório do script)
@st.cache_resource
def load_model():
    model = tf.keras.models.load_model("traffic_sign_model.h5")
    return model
```

A função `predict` realiza o pré-processamento da imagem e previsão, converte para RGB, redimensiona para 224x224 pixels e normaliza os seus valores.

A imagem é passada para o modelo, extrai a classe prevista e mostra a sua confiança.

```
# Função para processar a imagem e fazer a previsão
def predict(image, model):
    try:
        # Pré-processar a imagem
        img = image.convert("RGB") # Converter para RGB
        img = img.resize((224, 224)) # Redimensionar para o tamanho esperado pelo modelo
        img_array = np.array(img) / 255.0 # Normalizar os valores de pixel
        img_array = np.expand_dims(img_array, axis=0) # Adicionar dimensão batch

        # Fazer a previsão
        predictions = model.predict(img_array)
        predicted_class_idx = np.argmax(predictions[0])
        confidence = predictions[0][predicted_class_idx]

        return predicted_class_idx, confidence
    except Exception as e:
        st.error(f"Erro ao processar a imagem: {e}")
        return None, None
```

Um dicionário relaciona os índices numéricos às descrições das classes dos sinais de trânsito.

```
# Dicionário de classes do modelo
classes = [2, 8, 9, 10, 11]
class_names = {
    0: "Sinal 2 - Speed limit (50km/h)",
    1: "Sinal 8 - Speed limit (120km/h)",
    2: "Sinal 9 - No passing",
    3: "Sinal 10 - No passing for vehicles over 3.5 metric tons",
    4: "Sinal 11 - Right-of-way at the next intersection",
}
```

A aplicação apresenta o título e permite o carregamento de imagens. Após o upload, o botão “Classificar” executa o modelo e mostra a classe prevista e a sua respetiva confiança.

```
# Configurar título e descrição da página
st.title("Reconhecimento de Sinais de Trânsito")
st.write("Carregue uma imagem para classificar o sinal de trânsito utilizando o modelo treinado.")

# Carregar o modelo
model = load_model()

# Carregar a imagem
uploaded_image = st.file_uploader("Envie uma imagem no formato JPG ou PNG", type=["jpg", "jpeg", "png"])

if uploaded_image is not None:
    # Exibir a imagem carregada
    image = Image.open(uploaded_image)
    st.image(image, caption="Imagem carregada", use_container_width=True)

    # Botão para classificar a imagem
    if st.button("Classificar"):
        st.write("Classificando a imagem...")

        # Fazer a previsão
        with st.spinner("Processando..."):
            predicted_class_idx, confidence = predict(image, model)

        # Exibir os resultados
        if predicted_class_idx is not None:
            predicted_class = class_names.get(predicted_class_idx, "Classe desconhecida")
            st.success(f"Classe prevista: {predicted_class}")
            st.write(f"Confiança: {confidence:.4f}")
        else:
            st.error("Erro ao classificar a imagem. Verifique o formato ou tente novamente.")
```

Descrição da Implementação dos Algoritmos

Implementação do Random Search:

Pré-processamento para o Random Forest, as imagens são achatadas.

```
X_train_flat = X_train.reshape(X_train.shape[0], -1)
X_test_flat = X_test.reshape(X_test.shape[0], -1)
```

Definição da Random Search, onde os hiperparâmetros do Random Forest são definidos em um grid de procura aleatória.

```
param_grid = {
    'n_estimators': [10, 50, 100, 200],
    'max_depth': [5, 10, 20, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}
```

O modelo base do Random Forest é criado e ajustado usando RandomizedSearchCV. Após a procura, o melhor modelo (best_rf) é avaliado no conjunto de teste, calculando as métricas.

```
# Criar o modelo base para Random Forest
rf_model = RandomForestClassifier(random_state=42)

# Realizar Random Search
random_search = RandomizedSearchCV(rf_model, param_distributions=param_grid, n_iter=20, cv=3, verbose=2, random_state=42, n_jobs=-1)
random_search.fit(X_train_flat, y_train)

# Melhor modelo encontrado
best_rf = random_search.best_estimator_

# Avaliar o modelo Random Forest no conjunto de teste
rf_predictions = best_rf.predict(X_test_flat)
rf_accuracy = accuracy_score(y_test, rf_predictions)
rf_recall = recall_score(y_test, rf_predictions, average='weighted')
rf_precision = precision_score(y_test, rf_predictions, average='weighted')
rf_fmeasure = f1_score(y_test, rf_predictions, average='weighted')

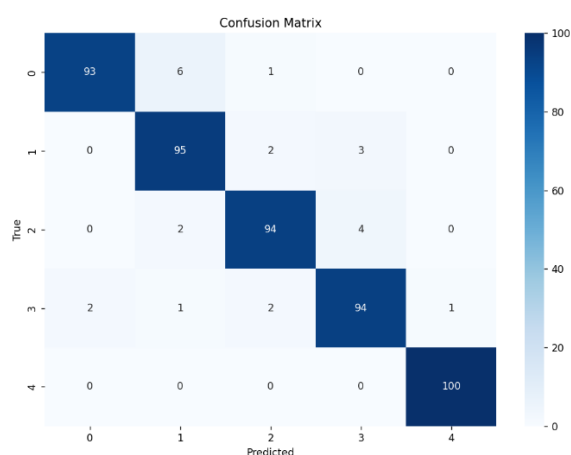
print(f"Random Forest Accuracy: {rf_accuracy:.4f}")
print(f"Random Forest Recall: {rf_recall:.4f}")
print(f"Random Forest Precision: {rf_precision:.4f}")
print(f"Random Forest F-measure: {rf_fmeasure:.4f}")
```

Análise de Resultados

Modelo Simples:

Estas imagens mostram a matriz de confusão, onde podemos observar o número de previsões corretas e incorretas para cada classe. A matriz indica um número elevado de precisões.

	Pred_2	Pred_8	Pred_9	Pred_10	Pred_11
True_2	93	6	1	0	0
True_8	0	95	2	3	0
True_9	0	2	94	4	0
True_10	2	1	2	94	1
True_11	0	0	0	0	100



De seguida, na tabela estão as métricas de precision, recall, f1-score, accuracy, macro avg e weighted avg para cada classe. As métricas indicam um desempenho bom do modelo.

	precision	recall	f1-score	support
Class_2	0,978947	0,93	0,953846	100
Class_8	0,913462	0,95	0,931373	100
Class_9	0,949495	0,94	0,944724	100
Class_10	0,930693	0,94	0,935323	100
Class_11	0,990099	1	0,995025	100
accuracy	0,952	0,952	0,952	0,952
macro avg	0,952539	0,952	0,952058	500
weighted avg	0,952539	0,952	0,952058	500

Os valores da AUC para cada classe, estão todos muito próximos de 1, o que demonstra uma excelente capacidade do modelo diferenciar as diferentes classes.

Class	AUC
2	0,996775
8	0,9963
9	0,996825
10	0,994475
11	0,999975

Random Search:

Esta tabela permite explorar o espaço de hiperparâmetros e identificar os conjuntos que podem ser encontrados através deste método de procura.

Hiperparâmetros	Perda
{'dropout_rate': np.float64(0.46475886306307707), 'learning_rate': np.float64(0.010369442666219138)}	0,148999074
{'dropout_rate': np.float64(0.0907898114535644), 'learning_rate': np.float64(6.07169171990872e-05)}	0,612426519
{'dropout_rate': np.float64(0.3463425699764095), 'learning_rate': np.float64(0.004313719224595805)}	0,158498078
{'dropout_rate': np.float64(0.22857961444137093), 'learning_rate': np.float64(0.044601842475536065)}	0,15648407
{'dropout_rate': np.float64(0.4581471858024578), 'learning_rate': np.float64(0.003630678861289458)}	0,178996838
{'dropout_rate': np.float64(0.311777229678997), 'learning_rate': np.float64(0.004764053218977656)}	0,153996575
{'dropout_rate': np.float64(0.39645511792317695), 'learning_rate': np.float64(0.026417472396234383)}	0,132502818
{'dropout_rate': np.float64(0.4452305322593182), 'learning_rate': np.float64(0.001597516901363555)}	0,231487109
{'dropout_rate': np.float64(0.37807854817412656), 'learning_rate': np.float64(0.003598166934707306)}	0,187000344
{'dropout_rate': np.float64(0.20962145653664688), 'learning_rate': np.float64(0.020059100286424568)}	0,132492312

Comparativamente ao modelo simples, como podemos ver, o Random Search teve resultados piores.

	precision	recall	f1-score	support
2	0,945946	0,7	0,804598	100
8	0,72093	0,93	0,812227	100
9	0,945055	0,86	0,900524	100
10	0,886792	0,94	0,912621	100
11	1	1	1	100
accuracy	0,886	0,886	0,886	0,886
macro avg	0,899745	0,886	0,885994	500
weighted av	0,899745	0,886	0,885994	500

Classe	AUC
2	0,9789
8	0,965175
9	0,982025
10	0,991175
11	1

Conclusões

O projeto apresentado abordou a classificação de sinais de trânsito utilizando modelos de machine learning e técnicas de otimização. Inicialmente, foi implementado um modelo simples utilizando a rede MobileNetV2 pré-treinada, com as camadas superiores adaptadas ao problema específico. O processo incluiu pré-processamento dos dados, garantindo um conjunto balanceado e adequado para o treino.

Adicionalmente, a técnica de Random Search foi utilizada para explorar o espaço de hiperparâmetros, procurando melhorias no desempenho do modelo. Embora tenha demonstrado ser uma abordagem robusta e flexível, os resultados não superaram os obtidos pelo modelo simples, destacando a eficácia da configuração inicial.

Os resultados experimentais, analisados por meio de métricas como accuracy, precision, recall, f1-score e AUC, mostraram um desempenho consistente e elevado, reforçando a capacidade do modelo em diferenciar corretamente as classes de sinais de trânsito. A matriz de confusão evidenciou a baixa incidência de erros de classificação, validando a robustez do modelo.

Por fim, a interface gráfica desenvolvida com Streamlit possibilitou uma aplicação prática e intuitiva do modelo, permitindo ao utilizador carregar imagens e visualizar os resultados de forma eficiente. A conclusão geral destaca a eficácia do uso de redes neurais convolucionais e da otimização de hiperparâmetros na classificação de imagens, bem como a importância de um pré-processamento adequado e de uma avaliação criteriosa dos resultados.

Bibliografia

Fichas Práticas de Inteligência Computacional;
PDFs Teóricos de Inteligência Computacional;