Modelo S sem Data Augmentation

A nossa primeira abordagem foi a mais simples: utilizamos somente os dados do nosso dataset sem qualquer tipo de data augmentation, com o objetivo de analisar o comportamento do modelo assim definido.

O principal desafio desta fase foi, sem dúvida, definir a estrutura base que utilizaríamos nos outros arquivos. Essa estrutura abrange a maneira como lemos os dados de treino, validação e teste dos diversos arquivos.

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import os
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dropout, Flatten,
Dense, Conv2D, MaxPooling2D, BatchNormalization
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping,
CSVLogger, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

"os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'"

Esta linha configura o TensorFlow para limitar a quantidade de mensagens de log que ele gera. O valor '2' faz com que apenas mensagens de erro e warnings sejam exibidas.

Testamos também com multiplos BATCH_SIZES, mas 32 foi o que obteve melhores resultados (por mais incrível que pareça)

Aqui estão os resultados com outros batch_sizes:

BATCH_SIZE = 64, Resultado obtido: 0.8408453464508057 BATCH_SIZE = 128, Resultado obtido: 0.8300280570983887 BATCH_SIZE = 32, Resultado obtido: 0.8544671535491943

(Importante realçar que estes valores estão a ser obtidos do ficheiro csv que geramos em cada execução, esses ficheiros estão a guardar os resultados de validação não de teste!)

```
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
# CONSTANTES
BATCH_SIZE = 32
IMG_SIZE = 32
NUM_CLASSES = 10  # nº classes para identificar
NUM_EPOCHS = 60
LEARNING_RATE = 0.001 # learning rate inicial
```

FOLDERS

```
# Folders do dataset
train_dirs = ['./dataset/train/train1', './dataset/train/train2',
'./dataset/train/train3', './dataset/train/train5']
validation_dir = './dataset/validation'
test_dir = './dataset/test'
```

Normalização das imagens

Aqui nós decidimos utilizar a biblioteca ImageDataGenerator para pré-processamento de imagens

E com ela definimos as instâncias train_datagen, validation_datagen, e test_datagen são configuradas para apenas o rescale dos valores dos pixels das imagens, normalizando-os para o intervalo [0, 1].

Para cada diretório de treino listado em train_dirs, um gerador é criado, a função desses geradores é redimensionar o tamanho das imagens para o tamanho definido na constante acima, mas visto que são criados multiplos geradores para o treino precisamos de os combinar num só e para isso criamos a função *combined_generator* depois são criados de maneira similar os geradores de validation e test

```
# CRIAR OS GERADORES
train_datagen = ImageDataGenerator(rescale=1./255)

validation_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

# training generators
train_generators = [train_datagen.flow_from_directory(
    train_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='categorical') for train_dir in train_dirs]

# Necessário para junstar os trainning generators
def combined_generator(generators):
    while True:
        for generator in generators:
            yield next(generator)

train_generator = combined_generator(train_generators)

# Validation e test generators
validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
```

```
    batch_size=BATCH_SIZE,
    class_mode='categorical')

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode='categorical')
```

Definição das funções para mostrar as metricas de Precision, Recall e F1 Score

```
from tensorflow.keras import backend as K
from tensorflow.keras.metrics import Metric

class Precision(Metric):
    def __init__(self, name='precision', **kwargs):
        super(Precision, self).__init__(name=name, **kwargs)
        self.true_positives = self.add_weight(name='tp',
initializer='zeros')
        self.predicted_positives = self.add_weight(name='pp',
initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_pred = K.round(y_pred)
        y_true = K.cast(y_true, 'float32')
        self.true_positives.assign_add(K.sum(y_true * y_pred))
        self.predicted_positives.assign_add(K.sum(y_pred))

    def result(self):
        return self.true_positives / (self.predicted_positives +
K.epsilon())

    def reset_states(self):
        self.true_positives.assign(0)
        self.predicted_positives.assign(0)

class Recall(Metric):
    def __init__(self, name='recall', **kwargs):
        super(Recall, self).__init__(name=name, **kwargs)
        self.true_positives = self.add_weight(name='tp',
initializer='zeros')
        self.actual_positives = self.add_weight(name='ap',
initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_pred = K.round(y_pred)
```

```
        y_true = K.cast(y_true, 'float32')
        self.true_positives.assign_add(K.sum(y_true * y_pred))
        self.actual_positives.assign_add(K.sum(y_true))

    def result(self):
        return self.true_positives / (self.actual_positives +
K.epsilon())

    def reset_states(self):
        self.true_positives.assign(0)
        self.actual_positives.assign(0)

class F1Score(Metric):
    def __init__(self, name='f1_score', **kwargs):
        super(F1Score, self).__init__(name=name, **kwargs)
        self.precision = Precision()
        self.recall = Recall()

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.precision.update_state(y_true, y_pred)
        self.recall.update_state(y_true, y_pred)

    def result(self):
        precision = self.precision.result()
        recall = self.recall.result()
        return 2 * ((precision * recall) / (precision + recall +
K.epsilon()))

    def reset_states(self):
        self.precision.reset_states()
        self.recall.reset_states()
```

Definição do Modelo

Neste pedaço de código, definimos a arquitetura do nosso modelo com Sequential. O modelo possui diversas camadas:

```
Camadas Conv2D: Utilizadas para extrair características das imagens.

Camadas de Normalização de Batch: Normalizam os outputs das camadas
convolucionais para acelerar o treinamento e melhorar a estabilidade.

Camadas de Ativação com ReLU: Introduzem não-linearidades no modelo,
permitindo a aprendizagem de funções complexas.

Camadas MaxPooling: Reduzem a dimensionalidade das características
extraídas, diminuindo a carga computacional e prevenindo o
overfitting.
```

Camadas de Dropout: Aplicamos dropout progressivamente maior até 50%, conforme recomendado, para prevenir o overfitting.

Camada Dense e Camada de Saída: No final, adicionamos uma camada Dense seguida pela camada de saída com a função de ativação softmax, pois temos 10 classes possíveis.

Escolhemos utilizar o otimizador Adam pois Combina vantagens do AdaGrad e RMSProp e em diversos forums era dito que o mesmo acelerava a convergência e lidava bem com grandes datasets (não que este pareça ser o caso mas também nao achamos problemas com datasets mais reduzidos).

```
model = Sequential([
    Conv2D(128, (3, 3), input_shape=(IMG_SIZE, IMG_SIZE, 3)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.3),

    Conv2D(256, (3, 3)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.5),

    Conv2D(512, (3, 3)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.5),

    Flatten(),
    Dense(512),
    BatchNormalization(),
    Activation('relu'),
    Dropout(0.5),

    Dense(NUM_CLASSES, activation='softmax')
])

# Compilar o modelo
model.compile(optimizer=Adam(learning_rate=LEARNING_RATE),
              loss='categorical_crossentropy',
              metrics=['accuracy', Precision(), Recall(), F1Score()])


model.summary()
```

Criar callbacks

Callbacks que são utilizados durante o treino do modelo.

Decidimos incluir o CSVLogger para registrar o progresso do treino do nosso modelo.

Também optamos por utilizar o EarlyStopping, pois era provável que ocorresse overfitting após um certo número de epochs. Com ele, o treinamento é interrompido automaticamente se o modelo não apresentar melhorias significativas após um número específico de epochs, acelerando assim o processo de treino dos diferentes modelos.

Em busca de mais otimização, encontramos o ReduceLROnPlateau, que permite reduzir a learning rate, conforme necessário (se nao melhorar de x em x epochs neste caso) para evitar oscilações ou estagnações no processo de treinamento, oque se comprovou eficiente!

Resultado dos modelos sem EarlyStopping e ReduceLROnPlateau:

Validation Accuracy: Validation Loss:

```
# Definir os Callbacks

# Para salvar o melhor modelo com base na acurácia de validação
checkpoint = ModelCheckpoint("models/01_sem_data_augmentation_.keras",
monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')

# Parar o treinamento se não houver melhoria na loss após x epochs
early_stopping = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)

# Salvar para csv
csv_logger =
CSVLogger(f'logs/01_sem_data_augmentation_batch_size_{BATCH_SIZE}_imag
e_size_{IMG_SIZE}.csv', append=True)

# Reduzir a learning rate se não houver melhoria na loss após x epochs
(lembrar de deixar este valor sempre menor que a patience no
early_stopping!!)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=3, verbose=1)
```

Avaliação do modelo

Aqui avaliamos o modelo treinado posteriormente e e chamamos os callbacks criados posteriormente.

```
# calcular passos por epoch
steps_per_epoch = sum([gen.samples // BATCH_SIZE for gen in
train_generators])
```

```python
# Treinar o modelo - Nao tirar os callbacks
history = model.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    epochs=NUM_EPOCHS,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // BATCH_SIZE,
    callbacks=[checkpoint, early_stopping, csv_logger, reduce_lr]
)

# Avaliar o modelo no test generator
# Avaliar o modelo no test generator
results = model.evaluate(test_generator)
loss, accuracy, precision, recall, f1_score = results[:5]
print(f"Test Loss: {loss}")
print(f"Test Accuracy: {accuracy}")
print(f"Test Precision: {precision}")
print(f"Test Recall: {recall}")
print(f"Test F1 Score: {f1_score}")

# Plots do treino
plt.figure(figsize=(12, 8))
plt.subplot(2, 1, 1)
plt.plot(history.history['accuracy'], label='train_accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(2, 1, 2)
plt.plot(history.history['val_precision'], label='val_precision')
plt.plot(history.history['val_recall'], label='val_recall')
plt.plot(history.history['val_f1_score'], label='val_f1_score')
plt.xlabel('Epoch')
plt.ylabel('Metrics')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.title('Validation Precision, Recall, F1 Score')

plt.tight_layout()
plt.show()
plt.savefig(f'./plots/01_sem_data_augmentation.png')
```