Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Deep Learning. Avance del Proyecto - Individual

TC3007C.501 Inteligencia artificial avanzada para la ciencia de datos II

Profesores:

Iván Mauricio Amaya Contreras

Blanca Rosa Ruiz Hernández

Félix Ricardo Botello Urrutia

Edgar Covantes Osuna

Felipe Castillo Rendón

Hugo Terashima Marín

Christian Carlos Mendoza Buenrostro

Alumno:

Alberto H Orozco Ramos - A00831719

4 de Noviembre de 2023

En este documento se busca condensar y explicar todas las actividades realizadas con respecto al modelo de Deep Learning implementado individualmente para el modelo de Deep Learning en la concentración de Inteligencia Artificial Avanzada II.

Resumiendo un poco, se nos solicitó elaborar un modelo de Redes Neuronales Convolucionales que permitiera realizar algún tipo de clasificación simple, en mi caso decidí utilizar un dataset de distintos tipos de osos que puede encontrar en este link, con lo cual, haciendo uso de la biblioteca TensorFlow/Keras para la implementación de una CNN que me permita definir, ajustar, entrenar, validar y probar su eficacia, precisión y exactitud.

En primera instancia, importé las librerías de TensorFlow/Keras, Numpy, MatPlotLib, Seaborn y SKLearn, que me sirven para establecer el modelo, manipulación de datos con imágenes, proyección de resultados y la evaluación del mismo modelo con sus respectivas métricas.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix
```

Luego procedí a cargar el dataset dividido en 3 sets para las 3 distintas fases de la elaboración del modelo:

- **Training Set:** El set de entrenamiento este compuesto del 40% total de las imágenes que componen el dataset. Estas son utilizadas dentro de la sección de entrenamiento, que sirve para que el modelo puedo probar algunos valores iniciales, entienda un poco del contexto de la situación y en base a ello pueda empezar a generar algunas cuántas predicciones que, con cada iteración, reduzca el valor del error lo más posible y esto le permita posteriormente proporcionar resultados coherentes y predicciones certeras.
- **Validation Set:** El set de validación este compuesto del 40% total de las imágenes que componen el dataset. Estas son utilizadas dentro de la sección de entrenamiento, que sirve para que el modelo puedo ser evaluado con una sección de los datos originales y medir qué tan bueno es su rendimiento basado en imágenes desconocidas por el modelo.
- **Test Set:** El set de pruebas está compuesto por el 20% restante del total de las imágenes que componen el dataset. Estas dos son utilizadas dentro de la sección de predicciones o pruebas, la cual consiste en que, en base al entrenamiento y la validación realizados previamente, el modelo comience a recibir valores de entrada sin recibir los valores de salida, generando las mismas salidas como predicciones por sí mismo. Evidentemente

se espera que teniendo un buen entrenamiento, validación y muy poca pérdida, el rango de error de las pruebas sea de igual forma pequeño y muy preciso con respecto a los datos reales.

```
train_data_dir = '/content/drive/MyDrive/Colab Notebooks/IA Avanzada
II/Deep Learning/data/train'
validation_data_dir = '/content/drive/MyDrive/Colab Notebooks/IA Avanzada
II/Deep Learning/data/val'
test_data_dir = '/content/drive/MyDrive/Colab Notebooks/IA Avanzada
II/Deep Learning/data/test'
```

Debido a que el docente sugirió aplicar una técnica de Transfer Learning, opté por aplicar Fine Tuning, esta se trata de una técnica del mencionado Transfer Learning para mejorar el rendimiento y resultados de un modelo que, en muchos casos, al no tener acceso a un dataset amplio y variado, se recurre a implementar otro modelo que tenga un propósito similar o parecido al buscado, y con ello mejorar su capacidad de aprendizaje significativamente, claro que solo se toma una porción relevante del otro modelo para este proceso.

El modelo pre-entrenado que se utilizó es el visto en clase llamado VGG16, que fue entrenado con datasets muy robustos (como ImageNet) para que su aprendizaje fuera eficaz y óptimo, reduciendo bastante el margen de error que podría presentar. Este modelo VGG16 lo utiicé una vez que construí el modelo de CNN de forma independiente, y lo implementé para observar la diferencia en los resultados y métricas de evaluación:

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(150,
150, 3))
for layer in base_model.layers:
    layer.trainable = False
```

Continuando con la construcción del modelo, establecí data generators para implementar data augmentation, con el fin de obtener variaciones del mismo dataset y que en base a ello, el modelo pueda interpretar las mismas imágenes interpretado de otras maneras ampliando las imágenes, distorsionándolas, recortando, reescalando, entre otros recursos.

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True)

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

También establecí el valor del batch_size = 32 y el target_size (150, 150), que es la mejor configuración que encontré probando con otros valores y comparando los resultados que

arrojaba el modelo. Estos mismos valores son utilizados en “train_generator” y “validation_generator”, las cuales son variables que definen cómo es que las imágenes deben de ser aumentadas y preprocesadas previo a ser enviadas al modelo, esto se realiza durante el entrenamiento así como en el proceso de validación. El proceso involucra lo antes mencionado, alterar las imágenes usando técnicas como recortes, rotación, zoom o desenfoque:

```
batch_size = 32
target_size = (150, 150)
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=target_size,
    batch_size=batch_size,
    class_mode='categorical')
validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
    target_size=target_size,
    batch_size=batch_size,
    class_mode='categorical')
```

Habiendo establecido lo anterior, ahora es momento de construir el modelo de CNN, con su respectiva cantidad de capas y neuronas por capa. Para la implementación sin utilizar el modelo VGG16 para fine tuning el código utiliza 3 capas iniciales, todas utilizan la función de activación “relu“ con 32, 64 y 128 neuronas por capa respectivamente. Además, después de cada una de estas capas, se aplica un proceso de Pooling para recorrer la imagen con una especie de filtro que se encarga de reducir el tamaño de las imágenes, ya sea ponderando o tomando valores máximos por cada fracción de imagen o feature map analizado:

```
model = models.Sequential()

# Modelo Original CNN's
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150,
3)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(5, activation='softmax')) # Se declaran 5 clases de osos
```

De igual forma, declaré la versión del modelo que incluye la VGG16, debido a que está la utilizé después de obtener los resultados del modelo original:

```
model = models.Sequential()

# Modelo que incluye fine tuning
model.add(base_model)

model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(5, activation='softmax')) # Se declaran 5 clases de osos
```

Ambos modelos coinciden en que poseen la capa para realizar Flattening, es decir, convertir las imágenes de matrices de 2 dimensiones a un arreglo muy grande de una sola dimensión, y también las últimas 2 capas son exactamente iguales, una con función de activación “relu” y 512 neuronas, y la última con función de activación “softmax” y 5 neuronas de salida para 5 clasificaciones de osos.

El siguiente paso fue compilar el modelo:

```
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Y posteriormente lo entrené con 10 iteraciones o epochs, que me parecieron suficientes al supervisar el modelo y los resultados que arrojaba con distintos valores:

```
epochs = 10  # Cantidad de iteraciones para el entrenamiento
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // batch_size)
```

Ahora se evalúa el modelo:

```
model.evaluate(validation_generator)
```

Y se guarda:

```
model.save('/content/drive/MyDrive/Colab Notebooks/IA Avanzada II/Deep
Learning/saved_model')
```

Finalmente, lo que queda por realizar es volver a cargar el modelo definido, declarar el "test_generator" para manipular las imágenes correspondientes al set de testing y desplegar las predicciones realizadas:

```
# Cargarmosel modelo para realizar el proceso de testing
loaded_model = tf.keras.models.load_model('/content/drive/MyDrive/Colab
Notebooks/IA Avanzada II/Deep Learning/saved_model')
# Compilación del modelo
loaded_model.compile(optimizer='adam',
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])
# Declaración del generador test_datagen
test_generator = test_datagen.flow_from_directory(
    test_data_dir,
    target_size=target_size,
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False)
# Generamos predicciones por cada set de pruebas existente
predictions = loaded_model.predict(test_generator)

# Convertimos las predicciones a etiquetas de clase
predicted_labels = np.argmax(predictions, axis=1)

# Convertimos las etiquetas verdaderas a etiquetas de clase
true_labels = test_generator.classes

# Obtenemos los nombres de los archivos para el generador
filenames = test_generator.filenames

names = ["black bear", "grizzly bear", "panda bear", "polar bear", "teddy
bear"]

# Por cada archivo
for i in range(len(filenames)):
  # Obtenemos la ruta del archivo
  img_path = test_data_dir +'/' + filenames[i]
  # Obtenemos la imagen
  img = plt.imread(img_path)
  # Generamos la predicción
  predicted_label = predicted_labels[i]
  true_label = true_labels[i]
  # Desplegamos el resultado
  print(f'Image: {filenames[i]}, Predicted: {names[predicted_label]}, True:
{names[true_label]}')
```

En cuanto a la evaluación del modelo, tanto original como implementando VGG16, utilicé las métricas de evaluación como “accuracy”, “precision”, “recall” y “F1”, además incluí una matriz de confusión para visualizar fácilmente las predicciones del algoritmo con respecto a los valores reales:

```
accuracy = accuracy_score(true_labels, predicted_labels)
precision = precision_score(true_labels, predicted_labels, average='weighted')
recall = recall_score(true_labels, predicted_labels, average='weighted')
f1 = f1_score(true_labels, predicted_labels, average='weighted')

print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
```

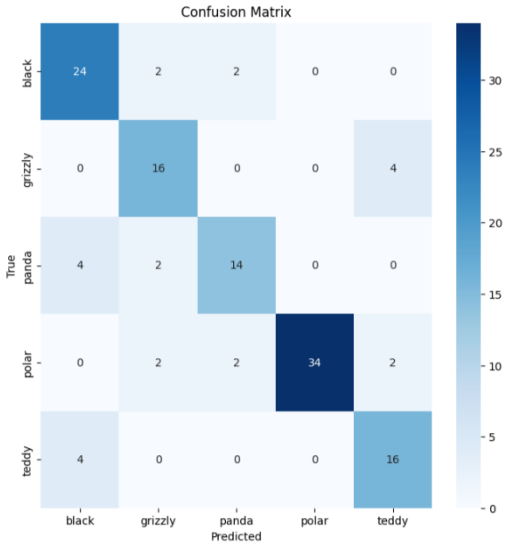
```
cm = confusion_matrix(true_labels, predicted_labels)
class_names = test_generator.class_indices.keys()

plt.figure(figsize=(8, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
yticklabels=class_names)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

Hablando un poco del modelo original, honestamente pienso que no es un mal modelo, claro que podría ser mejor considerando el tiempo que tuve para realizarlo. Este modelo presenta los siguientes resultados:

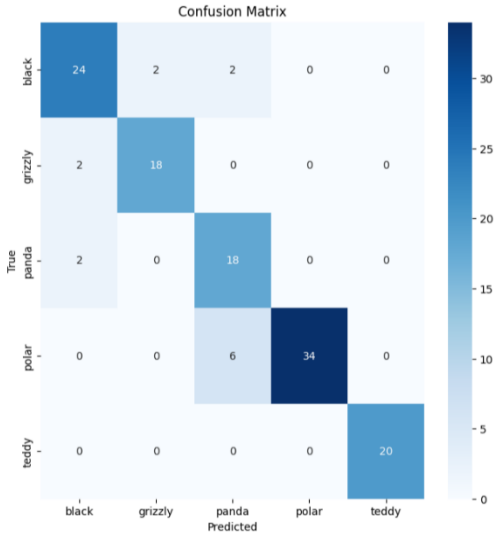
Modelo Original de Clasificación de Osos sin VGG16

```
Accuracy: 0.8125
Precision: 0.8253630050505051
Recall: 0.8125
F1 Score: 0.8153889792047687
```

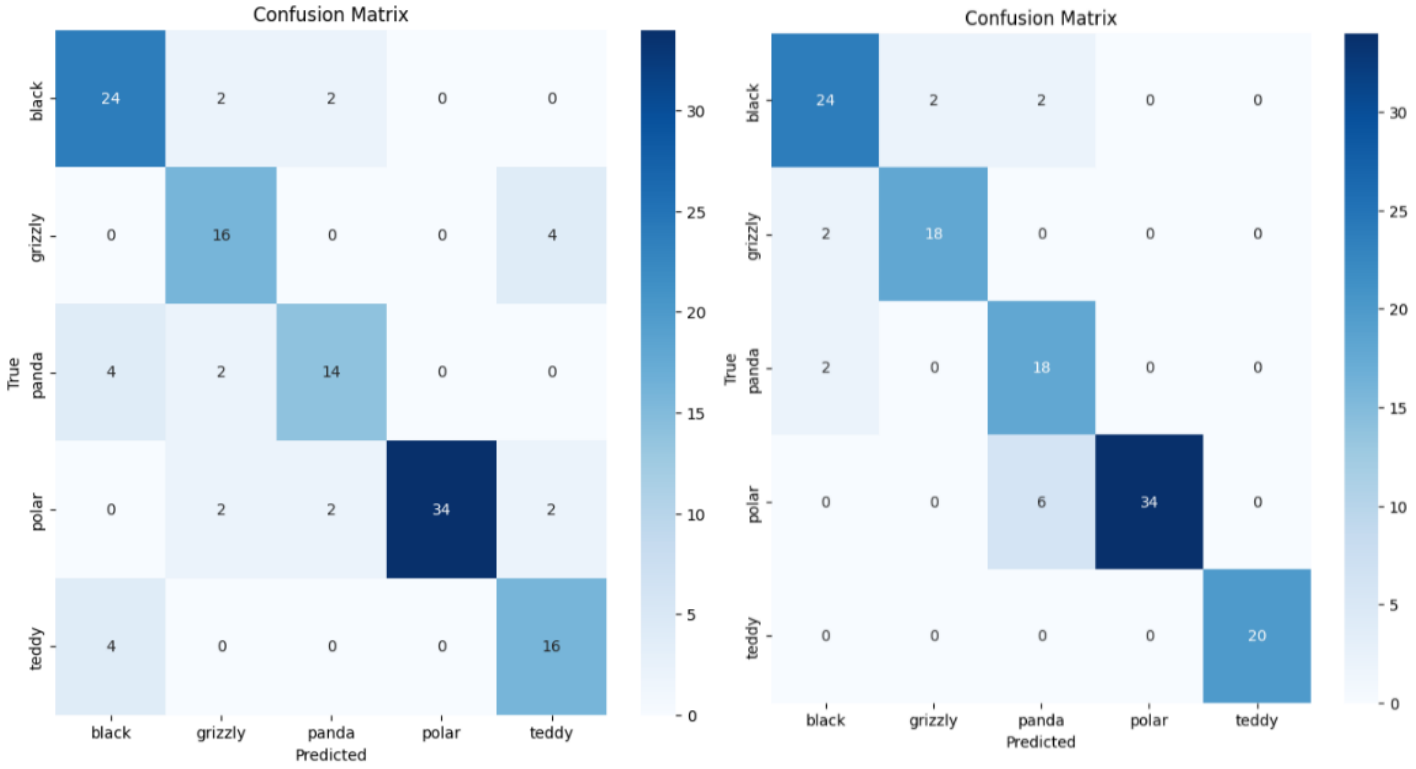


Modelo utilizando VGG16 (Fine Tuning o Transfer Learning)

```
Accuracy: 0.890625
Precision: 0.9050480769230769
Recall: 0.890625
F1 Score: 0.8938197708578144
```



Si comparamos los resultados de ambos modelos, podemos notar una clara diferencia en el valor de “accuracy” que posee cada uno. El primero modelo presenta un 81% aproximado de exactitud, en cambio el modelo que implementa fine tuning tiene un 89%, lo que es una mejora bastante considerable para su rendimiento. Curiosamente, en la métrica de “precision” podemos notar un pequeño decremento de 5% en el modelo que implementa VGG16 con respecto al original (lo mismo aplica para el “recall” y “F1”), lo cual tienes sentido si observamos sus matrices de confusión:



La matriz que se encuentra del lado izquierdo corresponde al modelo original, en ella podemos observar que existe más dispersión en los resultados que presentó el modelo con respecto a los valores reales. Básicamente, muestra un buen rendimiento, sin embargo existen casos en dónde categorizó osos incorrectamente y lo hizo con varias de las categorías.

Si hablamos de la matriz de la derecha, esta presenta un mejor trabajo, el modelo con fine tuning se equivocó en 14 ocasiones, cuando el original se equivocó en 22. Ahora, la razón principal por la cual personalmente creo que existe este decremento en la métrica de “precision” es debido a que el modelo que utiliza VGG16 se equivoca más en una misa categorización de oso que el original. Es decir, el modelo original muestra una confusión más dispersa, los errores se distribuyen por varias de las categorías de osos, el modelo con fine tuning no presenta tanta dispersión de error, sin embargo, concentra la mayor parte de sus errores en una categoría, específicamente se equivoca al intentar categorizar al panda cuando en realidad se trata de un oso polar.

Concluyendo un poco acerca de lo realizado, personalmente creo que ambos modelos son bastante competentes al menos para la tarea que les asigné, clasificar osos. Uno se confunde

más en diversas categorías, el otro falla más específicamente en una sola, sin embargo la mayor parte de los datos destinados a las pruebas se realizan correctamente por parte de ambos, aunque claro que con el debido tiempo, dedicación, análisis y correcta interpretación de los resultados, estos modelos pueden ser mejores, tal vez asegurando métricas arriba del 90%, inclusive quién sabe, tal vez se podría alcanzar un 100% si se refina de forma correcta y se evita caer en overfitting analizando pérdidas de datos con respecto al entrenamiento y validación.

Honestamente, no tuve muchos problemas encontrando los hiper parámetros correctos para el modelo, tampoco considero que los utilizados sean definitivos o que no se pueda llegar a un mejor refinamiento de los mismos, sin embargo, no presentan malos resultados, al contrario, son bastantes buenos y logran que el modelo de CNN cumpla con su tarea de categorización o clasificación de forma correcta, ya sea en un modelo u otro. Si se quisiera escalar este modelo o llevarlo más allá, por supuesto que se deberían implementar técnicas de refinamiento como GridSearch, que es una técnica encargada de realizar búsquedas exhaustivas para encontrar los mejores hiper parámetros posibles. También creo que encontrar un dataset más amplio y robusto, puede ayudar al algoritmo a aprender e interpretar los patrones, figuras y demás características de los distintos tipos de osos, al menos los 5 que utilicé para este trabajo.