Boletín upiita-ipn

HACIENDO FÁCIL EL APRENDIZAJE PROFUNDO

Celia Jade Zúñiga Zarco (Candidato a Ingeniero en Biónica)
czunigaz1400@alumno.ipn.mx
Raúl Pérez Núñez (Candidato a Ingeniero en Biónica)
rperezn1700@alumno.ipn.mx
Dr. en C. Álvaro Anzueto Ríos (Profesor)
aanzuetor@ipn.mx

Academia de Biónica
Unidad Interdisciplinaria en Ingeniería y Tecnologías Avanzadas
Instituto Politécnico Nacional

Resumen

Las redes neuronales con aprendizaje profundo han sido objeto de diversos estudios y se puede encontrar en la literatura diferentes arquitecturas; sin embargo, se tiene la creencia general que son difíciles de configurar y entrenar para que logren identificar a más de un objeto contenido en una imagen. El presente artículo pretende demostrar lo accesible del proceso de configurar y entrenar una arquitectura neuronal con aprendizaje profundo; en este trabajo se ha considerado para esta tarea la red nombrada RESNET50. Se presenta en bloques, las líneas de código en el lenguaje de programación Python, que logran la configuración y entrenamiento para el reconocimiento de dos entes presentes en una imagen; para nuestro caso perros y gatos. Para determinar la eficiencia de la arquitectura neuronal se presentan las gráficas de desempeño de los procesos de entrenamiento y validación; se ha alcanzado un valor de 96.49 % de precisión (accuracy) al identificar los dos entes; por lo tanto, se puede decir que el proceso de "haciendo fácil el aprendizaje profundo" es posible.

Introducción

Si bien las neuronas artificiales desde su aparición, allá por los años 40 [1], demostraron gran capacidad en el proceso de clasificación de datos, por muchos años se les criticó por tan sólo poder resolver sistemas linealmente separables. Con el correr de los años surgieron técnicas matemáticas que dotaron a las redes neuronales con nuevos algoritmos y cambios en sus estructuras de diseño; logrando de esta manera convertirlas en un referente al momento de hablar sobre aprendizaje en máquina y el paradigma de inteligencia artificial. Una clase de este tipo de arquitecturas neuronales fue nombrada como redes neuronales de aprendizaje profundo [2]; este tipo de redes logra el procesamiento e interpretación de escenas visuales (imágenes) extrayendo de ellas patrones que representan objetos o regiones de interés dentro de las mismas. Para la extracción de información de las imágenes, las redes neuronales con aprendizaje profundo generalmente utilizan el proceso matemático de la convolución, es decir, aplican el proceso de convolución entre pequeñas ventanas numéricas (típicamente matrices de dimensiones 3X3) y las imágenes. Una de las arquitecturas neuronales que se ha popularizado es la nombrada RESNET (Residual Neural Network), la cual, dependiendo del número de capas neurales de convolución puede recibir su nombre final, un ejemplo de ello y de uso a lo largo de este trabajo es la red "RESNET 50" [3]. En este trabajo se pretende dotar al lector con las herramientas básicas para la implementación de este tipo de redes. Se ha elegido identificar y clasificar dos tipos de patrones o estructuras gráficas dentro de las imágenes que corresponderán a perros y gatos. Se pretende demostrar que el proceso de aprendizaje y aplicación de las redes neuronales profundas es relativamente

sencillo, sin embargo, es importante informar que son estructuras previamente definidas y estudiadas [4].

Arquitectura de neurona con aprendizaje profundo RESNET

En 2015 la compañía Microsoft saco a la luz una arquitectura neuronal nombrada: Residential Energy Services Network (RESNET), la cual tiene como innovación la aplicación del concepto de redes neuronales residuales. Una arquitectura de tipo residual se basa en dar solución al problema que se presenta cuando se apilan capa tras capa de redes neuronales y que no siempre mejor el desempeño de una arquitectura neural, al contrario, pueden entorpecer el proceso de aprendizaje o síntesis de información por parte de las redes. Propagar la información y su diferencia (el error de ajuste) entre las capas se realiza empleando el proceso de gradiente del error, sin embrago, cuando este

se propaga entre muchas capas de neuronal ocurre que el gradiente pierde fuerza de acción conocida como desvanecimiento del gradiente y la red estanca su aprendizaje, por la profundidad que genera el apilado que las capas de neuronas, la precisión llega a degradarse y no variar, lo que conduce a un fallo en la arquitectura. Un bloque residual (ver Figura 1) tiene como idea principal saltar entre capas neuronales mejorando el proceso de aprendizaje al saltar directamente a la siguiente, esto minimiza el desvanecimiento del gradiente que a su vez ayuda a mejorar la precisión en el reconocimiento de una estructura o región de interés dentro de las imágenes.

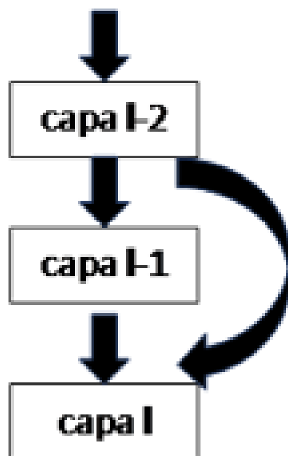

Figura .1 Representación de un bloque residual.

Como se ha mencionado, en este trabajo se ha implementado la arquitectura nombrada RESNET50, la cual podemos apreciar en la Figura 2. La RESNET50 tiene 50 capas profundas y diferentes link’s de recursividad dotándola de un mejor rendimiento y con una cantidad menor de capas neurales, en comparación con otras arquitecturas [5], para el reconocimiento entre perros y gatos que es el objetivo a alcanzar.

Aplicación de red neuronal profunda RESNET50

En este apartado se expone la aplicación del proceso de entrenamiento de la Red Neuronal con aprendizaje profundo RESNET50. La exposición se dará al presentar por bloques las líneas de código que corresponden al proceso total del entrenamiento. Es necesario considerar como una tarea previa la obtención de la base de imágenes a emplear; para nuestro caso escenas visuales que contiene a perros y gatos.

Para realizar el entrenamiento de una CNN, primero se declaran las bibliotecas necesarias:

- Numpy; la cual, permite operar sobre matrices y vectores.
- Tensorflow; se usa para declarar capas neuronales y entrenar redes neuronales.
- Os; para acceder a las direcciones del disco duro.

- Imagedatagenerator; se importa para usar una base de datos (BD) en una localización del disco duro, esta biblioteca ya está optimizada para almacenar datos. De no incluirse, y guardar todas las imágenes en variables, la memoria se llenaría.

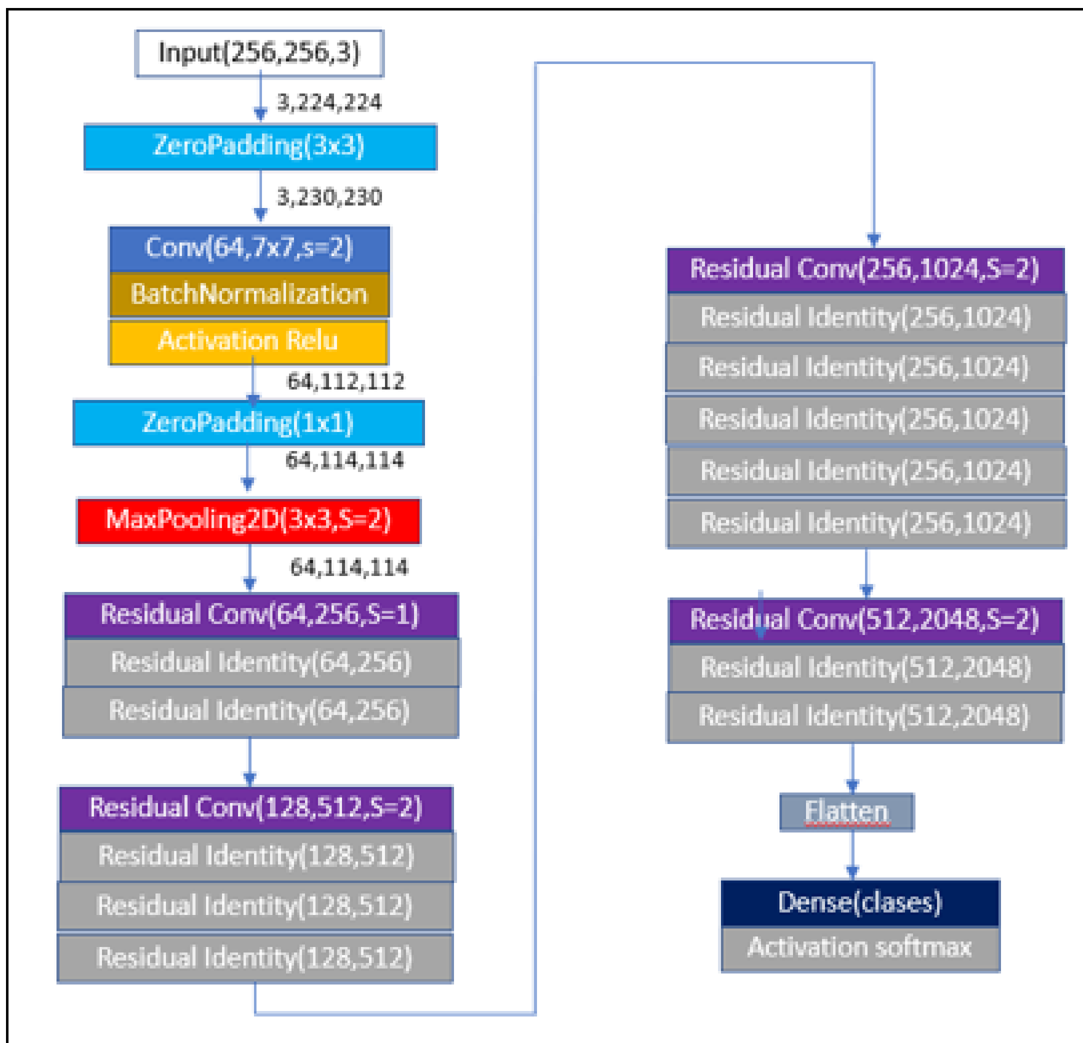

Figura 2. . Modelo RESNET50

Primero se declara el uso de imadata con una escala para que los valores de la imagen vayan de 0 a 1, y el método split para tenerlos como datos de referencia al comprobar que la red esté siendo bien entrenada, es decir, estos datos no son usados en el entrenamiento y representan el 20 % de toda la información.

Después se declara la dirección de la BD y el tamaño a rescalar de las imágenes. Se declara la base para el entrenamiento, se coloca la dirección, el tamaño de las imágenes que será en formato rgb (teniendo 3 capas por imagen).

Se utilizará el método shuffle para que los datos se presenten a la red de manera aleatoria y así aumentar la tasa de aprendizaje. Se utilizará el 80 % de las imágenes para el entrenamiento. Finalmente, se declara el número de clases encontrados en la base de datos. En la figura 3, se presenta el bloque que corresponde al llamado de las bibliotecas anteriormente descritas.

```python
import numpy as np
import os
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Figura 3. Bloque con líneas de código para el llamado de bibliotecas.

Dentro del llamado de las funciones, necesarias para el entrenamiento de la arquitectura neuronal, se tiene Rescale; la tarea que desarrolla es normalizar los valores de los pixeles, es decir, convierte un valor máximo de 255 a un valor máximo igual a 1. Validationsplit, es otra función, la cual, controla el porcentaje de imágenes a ser consideradas como validación, para este ejemplo se considera un 20 %, (referirse a la primer línea de código de la Figura 4.)

```python
entrenamiento_datagen = ImageDataGenerator(rescale=(1.0/255),validation_split=0.2)

datos_entrenamiento="./nuestra"
altura, longitud=224,224

imagen_entrenamiento = entrenamiento_datagen.flow_from_directory(
                                    datos_entrenamiento,
                                    target_size = (altura, longitud),
                                    color_mode = 'rgb',
                                    class_mode = 'categorical',
                                    shuffle = True,
                                    subset='training')
valudation_entrenamiento = entrenamiento_datagen.flow_from_directory(
                                    datos_entrenamiento,
                                    target_size = (altura, longitud),
                                    color_mode = 'rgb',
                                    class_mode = 'categorical',
                                    shuffle = True,
                                    subset='validation')
Num_clase=len(imagen_entrenamiento.class_indices)
```

Figura 4. Bloque de líneas de código para definir el modelo neuronal a emplear.

Continuando en el bloque de la Figura 5, se importa el modelo para declarar que una estructura es el modelo a entrenar, que en este caso será RESNET50. Input, flatten, dense y activation son parámetros que se usarán en la estructura de la red neuronal. Primero se declara el tamaño de la entrada de la red, teniendo 3 capas porque son imágenes en rgb, RESNET50 será la base de la red, donde se usan los pesos de imagenet U. Al parámetro include_top se le asigna el valor de falso debido a que es la capa de las densas ya establecida y no se usará. En la variable x (Figura 5) se almacena la base, al ser la primera parte de la red, se le agrega flatten en la estructura, debido a que los datos se transforman de un arreglo matricial a un arreglo vectorial. cnn será el modelo ya declarado, donde se coloca la entrada y la salida, la cual se representa por capas densas (cnn.summary no es necesario, sólo se incluyó para ver la estructura de la red ya definida).

```
from tensorflow.keras.models import Model
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dropout, Flatten, Dense, Activation
input_tensor = Input(shape=(altura, longitud, 3))
base_model = ResNet50(input_tensor=input_tensor, weights='imagenet', include_top=False)
x = base_model.output
x = Flatten()(x)
predictions = Dense(Num_clase, activation='softmax')(x)
cnn= Model(inputs=input_tensor , outputs=predictions)
cnn.summary()
```

Figura 5. Bloque de líneas de código para el almacenamiento de las imágenes de la BD en arreglos vectoriales.

Como lo muestra la figura 6, se importa la biblioteca modelcheckpoint, quien almacena el modelo dependiendo de una condición a declarada por el programador. En el checkponit de validación, se coloca el nombre del archivo a guardar. Monitor es la variable a monitorear, es decir, se observará para detectar si existe mejoría. Se guarda los mejores pesos, sólo los pesos sinápticos (valores numéricos), priorizando aquellos con mejores resultados, cuando la precisión de la red incremente, se guardará el modelo.

```
from tensorflow.keras.callbacks import ModelCheckpoint
checkpointer_val = ModelCheckpoint(filepath=os.path.join("pesos_cats_dogs.h5"),
                                   monitor='val_accuracy',
                                   verbose=1,
                                   save_best_only=True,
                                   save_weights_only=True,
                                   mode='max')
```

Figura 6. Guardado de los mejores valores para los pesos sinápticos.

La biblioteca optimizers se importa debido a la necesidad de calcular el error de la red y optimizarla mediante dicho error. Para mejorar el rendimiento, primero se declara el factor de aprendizaje, que va de 0 a 1, el compilador se coloca en el error que será calculado mediante una categoría denominada categorical crossentropy. El optimizador utilizado en este caso será Adam y la métrica para la validación será accuracy. Se usarán 100 épocas en el entrenamiento. El historial será guardado en las cuatro variables con el prefijo his de la Figura 7 para observar el error y la mejoría de la red en el tiempo. Se asignan los datos de entrenamiento, validation, épocas, y el callback para el checkpoint, se guardan las métricas de entrenamiento para poder observarlas en el historial de aprendizaje.

```
from tensorflow.keras.callbacks import ModelCheckpoint
checkpointer_val = ModelCheckpoint(filepath=os.path.join("pesos_cats_dogs.h5"),
                              monitor='val_accuracy',
                              verbose=1,
                              save_best_only=True,
                              save_weights_only=True,
                              mode='max')
```

Figura 7. Asignación de los parámetros de entrenamiento.
La variable his_accucary almacena el historial de la precisión de la red durante el entrenamiento, análogamente his_val_accucary almacena la precisión durante la validación. Por su parte, las variables his_loss y his_val_loss almacenan el error de la red durante el entrenamiento y la validación respectivamente.

Con la biblioteca matplotlib se pueden graficar la precisión y el error de la red durante el entrenamiento y la etapa de validación (referirse Figura 8).

```
from tensorflow.keras import optimizers
alpha_lr=0.0001
cnn.compile(loss='categorical_crossentropy',
        optimizer = optimizers.Adam(lr=alpha_lr),
        metrics = ['accuracy'])
epocas=100
history=cnn.fit(imagen_entrenamiento,
        validation_data=valudation_entrenamiento,
        epochs = epocas,
        verbose=1,
        validation_freq=1,

        callbacks=[checkpointer_val])

his_accuracy=history.history['accuracy']
his_val_accuracy=history.history['val_accuracy']
his_loss=history.history['loss']
his_val_loss=history.history['val_loss']
```

Figura 8. Graficación de la precisión y del error de la red a través de las épocas durante el entrenamiento y la validación.

Para hacer predicciones y comprobar la precisión de la red, primero se debe construir el modelo como se llevó a cabo anteriormente, colocando el tamaño de entrada de la imagen y el número de clases.

Resultados

Naturalmente, después de entrenar la red, se debe evaluar el desempeño de la misma, para conocer su precisión y hacer ajustes en caso de requerirse. La Figura 9 describe la precisión y el error de la red durante el entrenamiento (es decir, se evalúa el comportamiento de la red mientras se está entrenando) y la validación de resultados. La sección a) indica que se alcanzó el máximo número de aciertos dentro de las primeras 20 épocas del entrenamiento, con algunos decaimientos intermedios. La siguiente sección representa el error, o que tanto se aleja la respuesta de la red del valor esperado. Se observa un pico cerca de la época 60, lo cual sugiere que la red hizo un ajuste importante en los pesos sinápticos, al término de las épocas, se observa que el error es nulo. La sección c, es análoga de la a); sin embargo,

a diferencia del entrenamiento, la información que representa corresponde al comportamiento de la red ante imágenes que no había visto antes (validación). En esta ocasión, le toma más épocas a la red converger al máximo valor de precisión y hay una caída significativa en la época 60. En la última sección, el error ha disminuido en comparación con el error de entrenamiento y se mantiene cercano a cero después de la época 60.

```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(his_accuracy)
plt.title("accuracy")
plt.xlabel('epoca')
plt.ylabel('accuracy')

plt.figure()
plt.plot(his_val_accuracy)
plt.title("val_accuracy")
plt.xlabel('epoca')
plt.ylabel('accuracy')
plt.figure()
plt.plot(his_loss)
plt.title("loss")
plt.xlabel('epoca')
plt.ylabel('loss')
plt.figure()
plt.plot(his_val_loss)
```

a) b) c) d)

Figura 9. Gráficas de desempeño del entrenamiento y validación de la arquitectura RESNET50.

Después de la etapa de entrenamiento y de las pruebas realizadas durante la validación, se determinó que la red clasifica imágenes de perros y gatos con una precisión (accuracy) de 96.49 %.

Conclusiones

Parráfo

Referencias

Después de implementar y entrenar un modelo de clasificación con capas residuales, capaz de distinguir entre las clases perros y gatos, se puede concluir que el desempeño de la red es bueno, ya que, en la validación, la precisión se mantuvo por arriba del 90 % y el error cercano a 0. También se observó que la época 60 es crítica, ya que tanto en el entrenamiento como en la validación (y para la precisión y el error), en dicha época siempre hay cambios significativos tanto en ganancias como en pérdidas.

NOTA: Los autores se encuentran en la disponibilidad de compartir el archivo fuente del programa y la base de datos empleada; únicamente se requiere enviar un mail para la solicitud.

1. McCulloch, W.S., Pitts, W. (). *A logical calculus of the ideas immanent in nervous activity.Bulletin of Mathematical Biophysics 5,*, 115–133 (1943). https://doi.org/10.1007/BF02478259.

2. LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey. (**2015**). *"Deep Learning". "Deep Learning".* 521 (7553): 436–444. Bibcode:2015Natur.521..436L. doi:10.1038/nature14539. PMID 26017442. S2CID 3074096.

3. K. He, X. Zhang, S. Ren, y J. Sun, (**año**). ≪*Deep residual learning for image recognition*≫,*in Proceedings of the IEEE conference on computer vision and pattern recognition,* 2016, pp. 770–778.

4. E. Stevens, L. Antiga and T. (**2019**). *Viehmann,Viehmann,*2019.

5. Moreno Díaz-Alejo, Lara and Ruiz Iniesta, Almudena, (**año**). *"Análisis comparativo de arquitecturas de redes neuronales para la clasificación de imágenes", Trabajo de fin de master,* Universidad Internacional de La Rioja (UNIR), Enero 2020, URL: https://reunir.unir.net/bitstream/handle/123456789/10008/Moreno %20D %C3 %ADaz-Alejo %2C %20Lara.pdf?sequence=1&isAllowed=y

Cómo citar este artículo en APA

Zúñiga, c., Pérez, R. & Anzueto, A. (**1 de enero de 2022**). Haciendo fácil el aprendizaje profundo. *Boletín UPIITA.* (94).
https://www.poner la liga del articulo

Regresar al índice