SEGUNDA ENTREGA

INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL

CARLOS EDUARDO BAEZ CORONADO.

UNIVERSIDAD DE ANTIOQUIA.

FACULTAD DE INGENIERÍA.

MEDELLÍN.

202

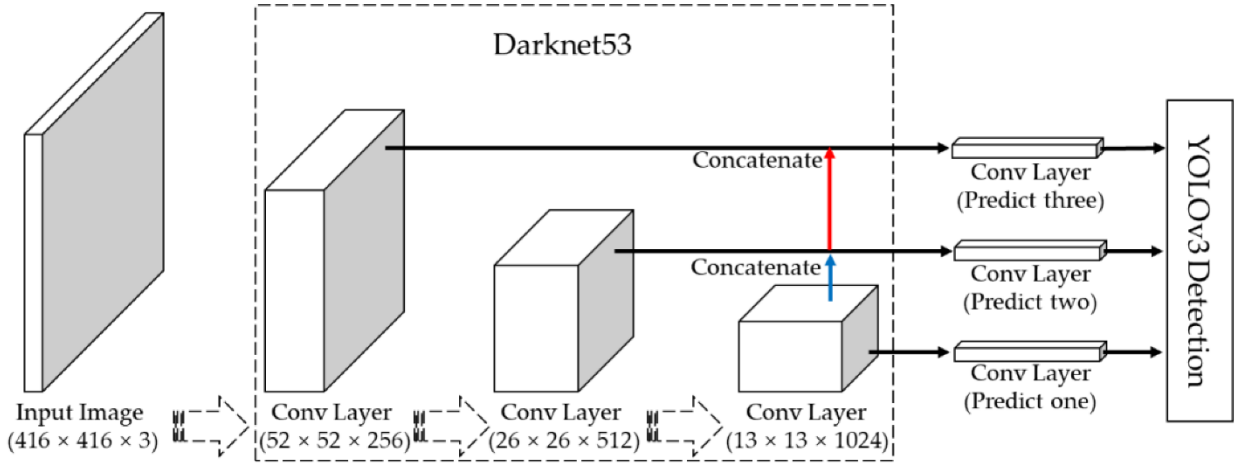
En esta entrega del proyecto, se dará a conocer la estructura principal de la red neuronal que se encargará de realizar la detección de objetos mediante el algoritmo de YOLO, aunque se espera que la implementación completa de esta red se entregue en la entrega final. En este documento solo se dará a conocer una de las partes más importantes de la red, la cual está relacionada con las capas convolucionales que la forman, y se hará una evaluación de su desempeño en realizar las tareas de clasificación de imágenes de un tamaño de entrada a la red de 416*416 pixeles, la cual corresponde a la arquitectura de la red neuronal conocida como DarkNet-53.

Sin embargo, dado el reducido poder computacional con el que se cuenta, el rendimiento de la red neuronal que se va a entrenar en comparación con la arquitectura ya mencionada va a ser más bajo. Estas limitaciones están relacionadas con el menor número de capas convolucionales y filtros (kernels) aplicados a las capas.

En las siguientes páginas se darán a conocer las consideraciones y características de la red propuesta.

El modelo de Darknet que será utilizado como punto de partida es el que se encuentra en la imagen de abajo. Tal como se había mencionado en la parte introductoria, en esta etapa del proyecto solo se diseñará la arquitectura y el entrenamiento de la red neuronal en la parte que se encuentra dentro del recuadro con líneas discontinuas de la imagen. Pero en vez de 53 capas convolucionales como el modelo de Darknet 53 solo se utilizarán 39 de ellas. Adicionalmente, se hará uso de una menor cantidad de filtros aplicados a las capas.

Se empezará con un número de 32 e irá aumentando cada bloque de capas convolucionales hasta llegar a un número de 512; mientras que en el modelo de YOLO V3 el número inicial de 256 filtros llegando a un máximo de 1024 en las últimas capas convolucionales.



A pesar de que la red neuronal tenga poca profundidad comparada con otras, se espera que presente Overfitting porque será entrenada con 5000 imágenes, por tanto, para disminuir este problema, se hará uso de unos objetos que permiten interactuar durante el entrenamiento del modelo, conocidos como callbacks. Los callbacks que se utilizarán son los siguientes:

- **Early Stopping:** Es una Buena herramienta para combatir el Overfitting del modelo, de tal manera que cuando detecte que el error de los datos de testeo comience a aumentar, detiene la operación y guarda el modelo que ha tenido el mejor desempeño.
- **ModelCheckpoint:** Este tipo de callback no reduce el Overfitting, en cambio, se encarga de guardar el modelo que está siendo entrenado cada época, de tal manera que si llega a ocurrir un error y se detiene el entrenamiento, no se pierda el progreso realizado hasta el momento.

Otra técnica ampliamente utilizada para combatir el Overfitting es conocida como DropOut, la cual se encargar de anular algunas neuronas durante el entrenamiento de tal manera que cuando se estén realizando las predicciones no intervengan en el resultando, forzando a las otras neuronas a mejorar la capacidad de predicción.

Para implementar el algoritmo de YOLO-V3 primero se empezará haciendo uso de las funciones que trae Keras.

Dado que la arquitectura de esta red neuronal trae dentro de sí varios bloques repetidos, se va a programar con la API funcional de Keras, ya que permite crear modelos más complejos con múltiples salidas.

Por ejemplo, el bloque de redes convolucionales lo definimos de la siguiente manera:

```
def conv_block (x, kernel_size, filters=32, padding='same', drop_perc=0.5):

  x=keras.layers.Conv2D(filters, kernel_size=kernel_size, padding=padding, activation='relu', kernel_initializer='he_normal')(x)
  x=keras.layers.BatchNormalization()(x)
  x=keras.layers.Dropout(drop_perc)(x)

  x=keras.layers.Conv2D(filters, kernel_size=kernel_size, padding=padding, activation='relu', kernel_initializer='he_normal')(x)
  x=keras.layers.BatchNormalization()(x)
  x=keras.layers.Dropout(drop_perc)(x)

  x=keras.layers.Conv2D(filters, kernel_size=kernel_size, padding=padding, activation='relu', kernel_initializer='he_normal')(x)
  x=keras.layers.BatchNormalization()(x)
  x=keras.layers.Dropout(drop_perc)(x)
  return x
```

Con estos bloques funcionales podemos ahorrar varias líneas de código, y minimizar errores, dado que en cada conjunto de bloques convolucionales diferentes, lo único que será necesario realizar, es cambiar los parámetros de entrada que va a recibir cada función, como por ejemploel tamaño del Kernel y el número de filtros.

La estructura de esta red neuronal se puede visualizar en la siguiente imagen:

```
keras.backend.clear_session()
input=keras.layers.Input(shape=[416,416,3])
x=keras.layers.Resizing(height=208, width=208)(input)
x=conv_block(x, kernel_size=203)
#En este momento tenemos la primera etapa de capas convolucionales, ahora vamos a calcular
#la segunda, para dejarla del tamaño de 104x104 debemos utilizar un kernel de 105
x=conv_block(x, kernel_size=105, filters=64)
x=conv_block(x, kernel_size=105, filters=64)
x=conv_block(x, kernel_size=105, filters=64)

#Las imagenes de la segunda capa tienen un tamaño de 52*52, por tanto el tamaño
#del kernel debe ser de 53
x=conv_block(x, kernel_size=53, filters=128)
x=conv_block(x, kernel_size=53, filters=128)
x=conv_block(x, kernel_size=53, filters=128)
# Ahora vamos a colocar el cuarto bloque de neuronas convolucionales, las cuales
# traen una tamaño de imagen de 26*26, port tanto el kernel debe ser de 27*27
x=conv_block(x, kernel_size=27, filters=256)
x=conv_block(x, kernel_size=27, filters=256)
x=conv_block(x, kernel_size=27, filters=256)
# El utlimo bloque de convolucion es de un tamaño de 13*13, por tanto el kernel es de 14
x=conv_block(x, kernel_size=13, filters=512)
x=conv_block(x, kernel_size=13, filters=512)
x=conv_block(x, kernel_size=13, filters=512)

model=keras.Model(inputs=[input], outputs=[x])
early_stopping=keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
cb=keras.callbacks.ModelCheckpoint('Model_saved', save_freq='epoch')
```