Convolutional Neural Networks in TensorFlow

Víctor Díaz Bustos

9 de febrero de 2023

Índice

1. Introducción

Este curso de Çonvolutional Neural Networks in TensorFlow"va más allá del primer curso en el que aprendiste a usar TensorFlow para implementar una Red Neural Convolucional básica. Durante la primera semana, se aplicarán los conocimientos adquiridos a un conjunto de datos mucho más grande de gatos frente a perros en Kaggle. En el último módulo, se verán caballos y humanos en un conjunto de datos de 1000 imágenes. La segunda semana se enfocará en lidiar con el problema del sobreajuste, a través del aumento de datos en TensorFlow y aprendizaje de transferencia, descargando una red neuronal previamente entrenada por otros en un gran número de imágenes.

¿Por qué es más probable el sobreajuste en datasets pequeños? Porque es menos probable que se encuentren todas las características posibles en el proceso de entrenamiento. El sobreajuste es más probable en datasets pequeños porque un modelo puede aprender características específicas de los datos de entrenamiento que no generalizan bien a datos nuevos, causando una mala performance en la tarea de predicción. Con un dataset pequeño, hay menos datos disponibles para entrenar el modelo y limitar su capacidad para aprender patrones relevantes y genéricos en los datos, aumentando la probabilidad de sobreajuste.

1.1. Visualizar el efecto de la convolución

```
import numpy as np
import random
from tensorflow.keras.utils import img_to_array, load_img

# Define a new Model that will take an image as input,
# and will output intermediate representations for all
# layers in the previous model
successive_outputs = [layer.output for layer in model.layers]
visualization_model = tf.keras.models.Model(inputs = model.input,
    outputs = successive_outputs)

# Prepare a random input image from the training set.
cat_img_files = [os.path.join(train_cats_dir, f) for f in
    train_cat_fnames]
dog_img_files = [os.path.join(train_dogs_dir, f) for f in
    train_dog_fnames]
img_path = random.choice(cat_img_files + dog_img_files)

# this is a PIL image
img = load_img(img_path, target_size=(150, 150))

# Numpy array with shape (150, 150, 3)
x = img_to_array(img)

# Numpy array with shape (1, 150, 150, 3)
x = x.reshape((1,) + x.shape)

# Scale by 1/255
```

```
x /= 255.0

# Run the image through the network, thus obtaining all
# intermediate representations for this image.
successive_feature_maps = visualization_model.predict(x)

# These are the names of the layers, so you can
# have them as part of our plot
layer_names = [layer.name for layer in model.layers]

# Display the representations
for layer_name, feature_map in zip(layer_names,
    successive_feature_maps):

  if len(feature_map.shape) == 4:

    #----------------------------------------
    # Just do this for the conv / maxpool layers, not the fully-
        connected layers
    #----------------------------------------
    # number of features in the feature map
    n_features = feature_map.shape[-1]

    # feature map shape (1, size, size, n_features)
    size = feature_map.shape[ 1]

    # Tile the images in this matrix
    display_grid = np.zeros((size, size * n_features))

    #-------------------------------------------------
    # Postprocess the feature to be visually palatable
    #-------------------------------------------------
    for i in range(n_features):
      x  = feature_map[0, :, :, i]
      x -= x.mean()
      x /= x.std ()
      x *=  64
      x += 128
      x  = np.clip(x, 0, 255).astype('uint8')

      # Tile each filter into a horizontal grid
      display_grid[:, i * size : (i + 1) * size] = x

    #-----------------
    # Display the grid
    #-----------------
    scale = 20. / n_features
    plt.figure( figsize=(scale * n_features, scale) )
    plt.title ( layer_name )
    plt.grid  ( False )
    plt.imshow( display_grid, aspect='auto', cmap='viridis' )
```

La clave para esto es entender la API **model.layers**, que le permite encontrar las salidas e iterar a través de ellas, creando un modelo de visualización para cada una. La variable a seguir es **display_grid** que se puede construir a partir de x que se lee como un mapa de características y se procesa un poco para la

visibilidad en el bucle central. A continuación, renderizaremos cada una de las convoluciones de la imagen , además de su agrupación, luego otra convolución, etc.

1.2. Evaluando accuracy y loss

```
#-----------------------------------------------
# Retrieve a list of list results on training
# and test data sets for each training epoch
#-----------------------------------------------
acc      = history.history[     'accuracy' ]
val_acc  = history.history[ 'val_accuracy' ]
loss     = history.history[    'loss' ]
val_loss = history.history['val_loss' ]

# Get number of epochs
epochs   = range(len(acc))

#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot  ( epochs,     acc )
plt.plot  ( epochs, val_acc )
plt.title ('Training and validation accuracy')
plt.figure()

#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot  ( epochs,     loss )
plt.plot  ( epochs, val_loss )
plt.title ('Training and validation loss'   )
```

2. Aumento de la imagen

El aumento de imágenes y el aumento de datos es una de las herramientas más utilizadas en el aprendizaje profundo para aumentar el tamaño de su conjunto de datos y hacer que sus redes neuronales funcionen mejor.

Así, por ejemplo, si tenemos un gato y nuestros gatos en nuestro conjunto de datos de entrenamiento siempre están en posición vertical y sus orejas son así, es posible que no detectemos un gato que está acostado. Pero con el aumento, ser capaz de rotar la imagen, o ser capaz de sesgar la imagen, o tal vez algunas otras transformaciones podrían generar efectivamente esos datos para entrenar. Así que distorsiona la imagen y solo la arrojas en el set de entrenamiento. Pero hay un truco importante para cómo hacer esto en TensorFlow también para no tomar una imagen , deformar, sesgar y luego volar los requisitos de memoria.

El Aumento de la Imagen es una herramienta muy sencilla, pero muy potente, que nos ayudará a evitar el sobreajuste de los datos. El concepto es muy sencillo: si los datos son limitados, las posibilidades de tener datos que coincidan con posibles predicciones futuras también son limitadas y, lógicamente, cuantos menos datos tenga, menos posibilidades tendrá de obtener predicciones precisas para datos que su modelo aún no ha visto. En pocas palabras, si se está entrenando un modelo para detectar gatos y el modelo nunca ha visto el aspecto de un gato tumbado, es posible que no lo reconozca en el futuro.

El aumento simplemente modifica las imágenes sobre la marcha durante el entrenamiento mediante transformaciones como la rotación. Así, podría "simular"la imagen de un gato tumbado girando 90 grados la de un gato "de pie". Es una forma barata de ampliar el conjunto de datos más allá de lo que ya se tiene.

```
train_datagen = ImageDataGenerator (rescale=1/255.0, rotation_range
    =40, width_shift_range=0.2, height_shift_range=0.2, shear_range
    =0.2, zoom_range=0.2, horizontal_flip=True, fill_mode='nearest'
    )
```

- ***rescale***: Es un escalar o un arreglo que se multiplica por los valores de las imágenes en el rango [0, 255]. En este caso, se está dividiendo por 255.0, lo que significa que los valores de las imágenes se escalarán al rango [0, 1].
- ***rotation_range***: Es un número que especifica el rango de rotación de las imágenes. Cualquier número dentro de este rango será elegido aleatoriamente y las imágenes se rotarán en ese ángulo. En este caso, la imagen se rotará en un ángulo aleatorio entre 0^{o} y 40^{o}.
- ***width_shift_range*** y ***height_shift_range***: Son números que especifican la proporción de desplazamiento en ancho y alto de las imágenes, respectivamente. Cualquier número dentro de estos rangos será elegido aleatoriamente y las imágenes se desplazarán en esa cantidad. En este caso se desplazarán entre un 0 % y un 20 % tanto horizontal como verticalmente.

- ***shear_range***: Es un número que especifica el rango de deformación de las imágenes. Cualquier número dentro de este rango será elegido aleatoriamente y las imágenes se deformarán en esa cantidad. En este caso se deformarán entre un 0 % y un 20 %.
- ***zoom_range***: Es un número que especifica el rango de zoom de las imágenes. Cualquier número dentro de este rango será elegido aleatoriamente y las imágenes se ampliarán o reducirán en esa cantidad.
- ***horizontal_flip***: Es un booleano que especifica si las imágenes se deben voltear horizontalmente o no. Si es True, las imágenes se voltearán aleatoriamente.
- ***fill_mode***: Es una cadena que especifica cómo rellenar los pixels que quedan después de la rotación, desplazamiento, deformación y zoom. 'nearest' significa que se utilizará el valor del pixel más cercano.

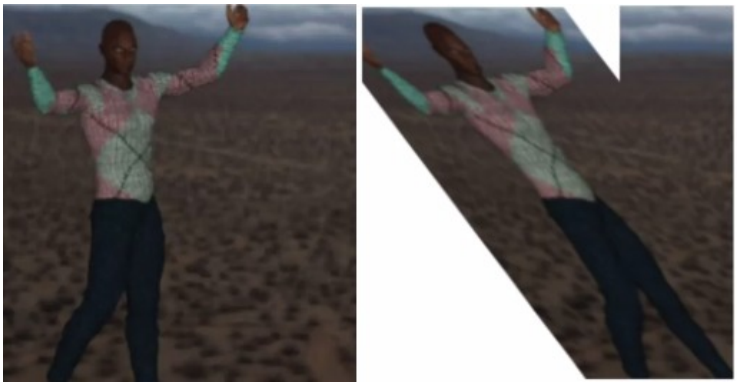


Figura 1: Ejemplo de deformación

Por supuesto, el aumento de imágenes no es la bala mágica para curar el sobreajuste. Realmente ayuda tener una gran diversidad de imágenes.

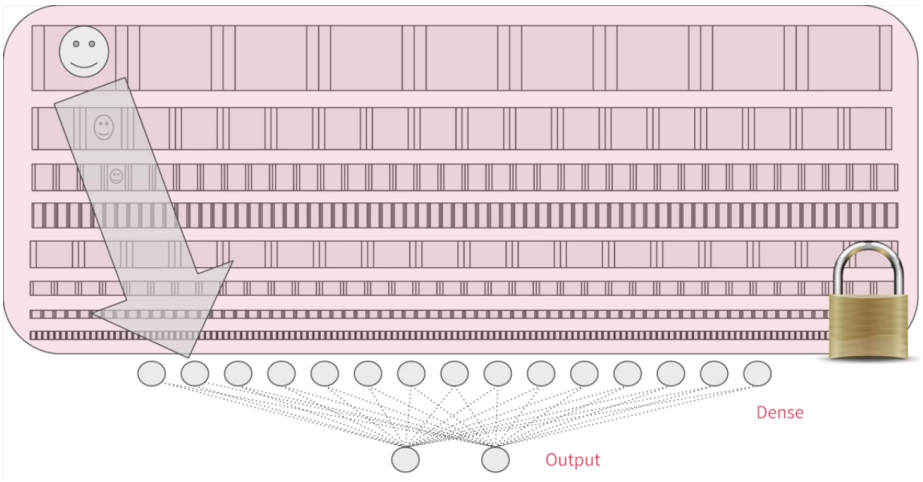
En algunas ocasiones, al principio, la precisión de validación parece estar en paso, pero luego la verá variando salvajemente. Lo que sucede aquí es que, a pesar del aumento de la imagen, la diversidad de imágenes sigue siendo demasiado escasa y el conjunto de validación también puede estar mal diseñado, es decir, que el tipo de imagen que contiene está demasiado cerca de las imágenes del conjunto de entrenamiento.

Entonces, lo que podemos aprender de esto es que el aumento de la imagen introduce un elemento aleatorio a las imágenes de entrenamiento, pero si el conjunto de validación no tiene la misma aleatoriedad, entonces sus resultados pueden fluctuar. Así que ten en cuenta que no solo necesitas un amplio conjunto de imágenes para entrenar, sino que también las necesitas para probar o el aumento de imágenes no te ayudará mucho.

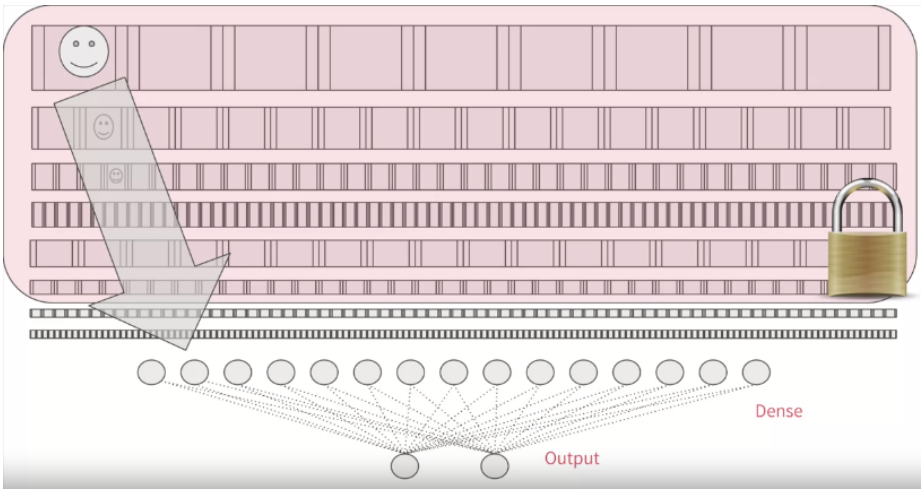
3. Aprendizaje de transferencia

El aprendizaje de transferencia es una de las técnicas más importantes del aprendizaje profundo y TensorFlow te permite hacerlo con solo un puñado de líneas de códigos. En lugar de tener que entrenar una red neuronal desde cero necesitando una gran cantidad de datos y tomar mucho tiempo para entrenar, podemos descargar un modelo de código abierto que alguien más ya haya entrenado en un conjunto de datos enorme tal vez durante semanas y usar esos parámetros como para luego entrenar su modelo un poco más en tal vez un conjunto de datos más pequeño que tiene para una tarea determinada.

Considere el modelo de otra persona, quizás uno que sea mucho más sofisticado que el suyo, entrenado en muchos más datos. Tienen capas convolucionales y están intactas con entidades que ya se han aprendido. Por lo tanto , puede bloquearlos en lugar de volver a entrenarlos en sus datos y hacer que solo extraigan las entidades de sus datos utilizando las convoluciones que ya han aprendido. Así, puede tomar un modelo que ha sido entrenado en unos conjuntos de datos muy grandes y utilizar las convoluciones que aprendió al clasificar sus datos.



Es típico que puedas bloquear todas las convoluciones, pero no tienes por qué hacerlo. También puede elegir volver a entrenar algunas de las inferiores porque pueden ser demasiado especializadas para las imágenes a mano. Se necesita un poco de prueba y error para descubrir la combinación correcta.



Así que echemos un vistazo a cómo lograríamos esto en el código. Empezaremos con las entradas. En particular, usaremos la API de capas keras, para seleccionar las capas, y para entender cuáles queremos usar y cuáles queremos volver a entrenar.

```
import os
from tensorflow.keras import layers
from tensorflow.keras import Model
from tensorflow.keras.applications.inception_v3 import InceptionV3

local_weights_file = '/tmp/
    inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'

pre_trained_model = InceptionV3(input_shape=(150,150,3),
    include_top=False, weights=None)

pre_trained_model.load_weights(local_weights_file)
```

InceptionV3() tiene una capa totalmente conectada en la parte superior. Por lo tanto, al establecer ***include_top*** en **False**, está especificando que desea ignorar esto y llegar directamente a las convoluciones.

Ahora que tengo mi modelo preentrenado creado en una instancia, puedo iterar a través de sus capas y bloquearlas, diciendo que no serán entrenables con este código.

```
for layer in pre_trained_model.layers:
    layer.trainable = False
```

A continuación, puede imprimir un resumen de su modelo preentrenado con este código, pero estar preparado, es enorme.

```
pre_trained_model.summary()
```

3.1. Añadir nuestra propia NN

Todas las capas tienen nombres, por lo que puede buscar el nombre de la última capa que desea utilizar. Si inspecciona el resumen, verá que las capas inferiores se han convertido en 3 por 3. Pero quiero usar algo con un poco más de información. Así que moví la descripción del modelo para encontrar mixed7, que es la salida de una gran cantidad de convolución que son 7 por 7. No tienes que usar esta capa y es divertido experimentar con otros. Pero con este código, voy a agarrar esa capa desde el inicio y llevarla a la salida.

```
last_layer = pre_trained_model.get_layer('mixed7')
last_output = last_layer.output
```

Así que ahora vamos a definir nuestro nuevo modelo, tomando la salida de la capa mixed7 del modelo inicial, que habíamos llamado *last_ouput*.

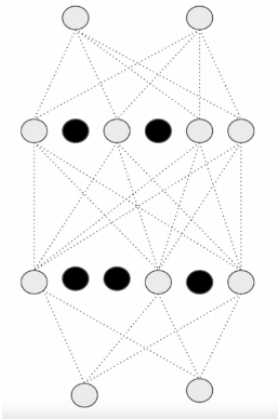
```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense(1, activation='sigmoid')(x)

model = Model(pre_trained_model.input, x)
model.compile(optimizer=RMSprop(learning_rate=0.0001), loss='
    binary_crossentropy', metrics=['acc'])
```

El código es un poco diferente, pero esta es solo una forma diferente de usar la API de capas. Comienza aplanando la entrada, que simplemente resulta ser la salida desde el inicio. Y, a continuación, agregue una capa oculta densa. Y luego su capa de salida que tiene sólo una neurona activada por un sigmoide para clasificar entre dos elementos. A continuación, puede crear un modelo utilizando la clase abstracta Model. Y pasando en la entrada y la definición de capas que acaba de crear. Y luego lo compila como antes con un optimizador y una función de pérdida y las métricas que desea recopilar.

Aquí está el gráfico de la precisión de la formación frente a la validación. Como puede ver, aunque comenzó bien, la validación está desviando de la formación de una manera muy mala. Entonces, ¿cómo arreglamos esto? Echaremos un vistazo a eso en la próxima lección.

3.2. Dropouts

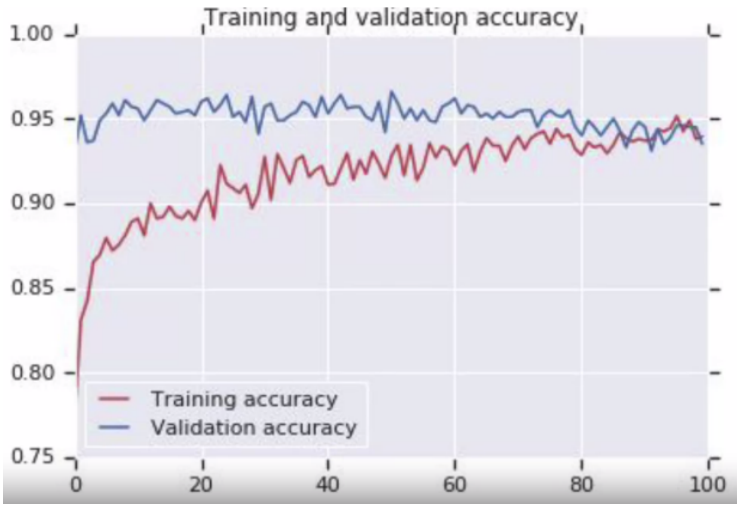
La idea detrás de esto es **eliminar un número aleatorio de neuronas** en su red neuronal. Esto funciona muy bien por dos razones: la primera es que **las neuronas vecinas a menudo terminan con pesos similares**, lo que puede conducir a un **sobreajuste**, por lo que eliminar algunas al azar puede eliminar esto. La segunda es que a menudo **una neurona puede sobresalir el input de una neurona en la capa anterior y puede especializarse en exceso como resultado**. Por lo tanto, el dropout puede romper este potencial mal hábito de la red neuronal.



Dejando algunos, hacemos que se vea así. Y eso tiene el efecto de que los vecinos no se afecten demasiado y potencialmente eliminen el exceso de ajuste.

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
# Add dropout
x = layers.Dropout(0.2)(x)
x = layers.Dense(1, activation='sigmoid')(x)

model = Model(pre_trained_model.input, x)
model.compile(optimizer=RMSprop(learning_rate=0.0001), loss='
    binary_crossentropy', metrics=['acc'])
```

El parámetro de *Dropout()* está entre 0 y 1 y es la fracción de unidades a soltar. En este caso, estamos abandonando el 20 % de nuestras neuronas. Cuando ves que la validación se aleja de la formación como esta con el tiempo, en realidad es un gran candidato intentar usar un dropout.



Y aquí está el impacto de la deserción. Puedes ver que es muy significativo.

4. Clasificación multiclase

En las últimas lecciones, has estado construyendo un clasificador binario. Uno que detecta dos tipos diferentes de objetos, caballo o humano, gato o perro, ese tipo de cosas. En esta lección, vamos a echar un vistazo a cómo podemos extender eso para varias clases.

Una vez configurado el directorio, debe configurar el generador de imágenes. Para varias clases, tendrá que cambiar esto a categórico como este.

```
train_datagen = ImageDataGenerator (rescale=1/255.0)

train_generator = train_datagen.flow_from_directory(train_dir,
    target_size=(300,300), batch_size=128, class_mode='categorical'
    )
```

El siguiente cambio viene en la definición del modelo, donde tendrá que cambiar la capa de salida. Para un clasificador binario, era más eficiente para usted tener una sola neurona y usar una función sigmoide para activarla. Esto significaba que saldría cerca de cero para una clase y cerca de una para la otra. Ahora, eso no encaja para multi-clase, así que tenemos que cambiarlo, pero es bastante simple. Ahora, tenemos una capa de salida que tiene N neuronas, y es activada por softmax que convierte todos los valores en probabilidades que suman hasta uno.

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(16,(3,3), activation='relu', input_shape
        =(300,300,3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32,(3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64,(3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.Flatten(),
    tf.keras.Dense(512, activation='relu'),
    tf.keras.Dense(3, activation='softmax')
])
```

El cambio final se produce cuando se compila la red. Si recuerda con los ejemplos anteriores, su función de pérdida era entropía cruzada binaria (*binary crossentropy*). Ahora, cambiarás es una entropía cruzada categórica como esta.

```
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.losses import CategoricalCrossentropy

model.compile(loss=CategoricalCrossentropy(), optimizer=RMSprop(
    learning_rate=0.001), metrics=['accuracy'])
```

Existen otras funciones de pérdida categórica, incluyendo escasas entropía categórica cruzada(*sparse categorical cross entropy*), que usaste en el ejemplo de la moda, y por supuesto también puedes usarlas.

La principal diferencia entre *'Sparse Categorical Crossentropy'* y *'Categorical Crossentropy'* es cómo se esperan las etiquetas de clase.

SparseCategoricalCrossentropy espera que las etiquetas de clase se encuentren en formato de enteros, es decir, cada etiqueta está representada como un entero único en lugar de un vector one-hot.

CategoricalCrossentropy espera que las etiquetas de clase se encuentren en formato one-hot, es decir, cada etiqueta está representada como un vector donde todos los elementos son ceros excepto el que corresponde a la clase.

En términos de cálculo de pérdida, ambas funciones realizan la misma operación, que es calcular la entropía cruzada entre las etiquetas y las salidas de la red neuronal. La diferencia radica en cómo se esperan las etiquetas. Por lo tanto, debes elegir la función adecuada en función de la forma en que tengas representadas tus etiquetas de clase.

```
1 from tensorflow.keras.optimizers import RMSprop
2 from tensorflow.keras.losses import SparseCategoricalCrossentropy
3
4 model.compile(loss=SparseCategoricalCrossentropy(from_logits=False)
    , optimizer=RMSprop(learning_rate=0.001), metrics=['accuracy'])
```