Introduction to TensorFlow for Artificial Intelligence, Machine Learning, and Deep Learning

DeepLearning.AI

Víctor Díaz Bustos

31 de enero de 2023

Índice

1. Introducción

La programación ha sido esencial en el desarrollo de aplicaciones, descomponiendo los requisitos en problemas parciales que luego son programados. El **aprendizaje automático reorganiza** este proceso, donde en lugar de descubrir las reglas nosotros mismos, el computador lo hace a través de un gran conjunto de ejemplos y etiquetas, especialmente valioso para problemas que no pueden ser resueltos de otra manera. El aprendizaje automático es similar al proceso de programación, pero con la inversión de los ejes, donde los datos y respuestas entran y las reglas son inferidas por la máquina.

Una **red neuronal** es sólo una implementación ligeramente más avanzada de aprendizaje automático que llamamos aprendizaje profundo. Pero afortunadamente es realmente muy fácil de programar. Vamos a saltar directamente al **aprendizaje profundo**.

1.1. TensorFlow y Keras

En este documento se presenta el uso de **TensorFlow**, una librería de código abierto desarrollada por Google, para el aprendizaje automático. TensorFlow es una plataforma altamente escalable y flexible para la construcción y el entrenamiento de modelos de aprendizaje automático, ya sea para problemas de inteligencia artificial, aprendizaje automático o aprendizaje profundo. En este documento se explicará cómo utilizar TensorFlow para construir y entrenar modelos de aprendizaje automático, proporcionando una introducción práctica para aquellos interesados en utilizar esta herramienta para resolver problemas reales en el mundo real.

También veremos el uso de **Keras**, una biblioteca de ***python*** que hace muy fácil definir redes neuronales. La red neuronal más simple posible es aquella que tiene una solo neurona en ella, eso es lo que hace el siguiente código.

```
1 import tensorflow
2 from tensorflow import keras
3
4 model = keras.Sequential ([keras.layers.Dense(units=1, input_shape
      =[1])])
```

En Keras usas la palabra *dense* para definir una capa de neuronas conectadas. Aquí solo hay un dense. Así que solo hay una capa y solo hay una unidad en ella, es una sola neurona. Las sucesivas capas están definidas en secuencia, de aquí la palabra sequential.

1.2. Funciones de pérdida y optimizadores

Para el aprendizaje automático, necesitas saber y usar un montón de matemáticas, cálculo, probabilidades y cosas similares. Es realmente muy bueno entenderlo al querer optimizar los modelos, pero lo bueno por ahora sobre TensorFlow y Keras es que muchas matemáticas están implementadas para ti en funciones. Existen dos roles de funciones de los que debes ser consciente estas son las funciones de **pérdida** y los **optimizadores**.

```
1    model.compile (optimizer='sgd',
2         loss='mean_squared_error')
```

La función de pérdida mide lo buena o mala que es una conjetura (conclusión o proposición que se ofrece de manera tentativa sin demostración, a través de indicios u observaciones) y luego pasa los datos al optimizador que averigua la siguiente conjetura. El optimizador piensa sobre lo bien o mal que se hizo la conjetura usando los datos con la función de pérdida y es responsable de ajustar los pesos del modelo para reducir el valor de la función de pérdida, minimizando el error del modelo. La lógica es que cada conjetura debe ser mejor que la anterior. A medida que las conjeturas se mejoran la precisión se acerca al 100 por ciento, se usa el término *convergencia*.

```
1    model.fit(xs, ys, epochs=500)
```

La función fit() es una función de Keras que se utiliza para entrenar un modelo de aprendizaje automático a partir de un conjunto de datos de entrada y objetivos. Toma como argumentos los datos de entrada (xs), los objetivos (ys) y un número de épocas (epochs) para el entrenamiento.

```
1    prediction = model.predict([new_x])
```

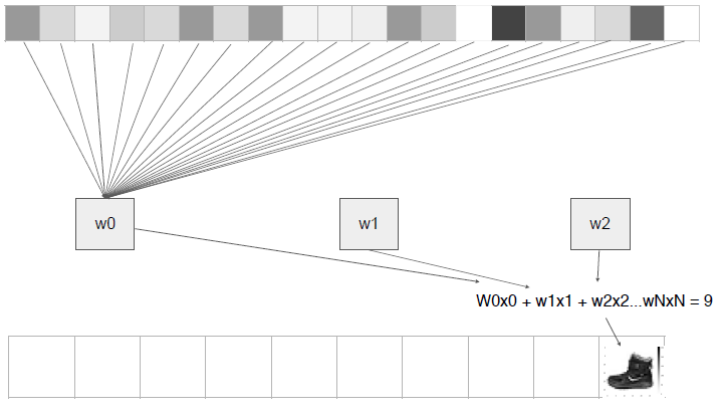
La función predict() es una función de Keras que se utiliza para hacer predicciones con un modelo entrenado. Toma como argumento un conjunto de datos de entrada y devuelve las predicciones del modelo para esos datos.

2. Visión por computador

Visión por computador es el campo en el que una máquina entiende y etiqueta lo que está presente en una imagen.

```
import tensorflow
from tensorflow import keras

model = keras.Sequential ([
    keras.layers.Dense(input_shape=(28,28)),
    keras.layers.Dense(units=128,
        activation=tf.nn.relu)
    keras.layers.Dense(units=10,
        activation=tf.nn.softmax)
])
```

Ahora vemos el código para la definición de una red neuronal. Ahora tenemos 3 capas. Las cosas importantes a mirar son la primera y última capas. La última capa tiene 10 neuronas porque tenemos 10 clases de ropa en el conjunto de datos. Esto debe siempre coincidir. La primera capa es una capa **flatten** con la forma de entrada de 28 por 28. Si recuerdas, nuestras imágenes son de 28x28, estamos especificando que esta es la forma en que debemos esperar los datos de entrada. Flatten toma este cuadrado de 28 por 28 y lo transforma en un vector lineal simple. Lo interesante pasa en la capa intermedia, muchas veces llamada también capa oculta. Esta tiene 128 neuronas, y podrían verse como las variables en una función (x1, x2, ..., x128).



Existe una regla que incorpora todas estas que convierte los 784 valores de un botín en el valor nueve, y similarmente para todos los otros 70.000. Es una función muy compleja de ver mapeando las imágenes tú mismo, pero eso es lo que hace una red neuronal. Por ejemplo si dices la función y es igual a w1 por x1, más w2 por x2, más w3 por x3, hasta w128 por x128. Hallando los valores de w, entonces y dará nueve, cuando tienes el valor de entrada del zapato.

2.1. ReLu

¿Qué es ReLu?

ReLu es una función de activación no lineal que se utiliza en redes neuronales multicapa o redes neuronales profundas. Esta función se puede representar como: $f(x) = max(0, x)$, donde x = un valor de entrada

Según la ecuación 1, la salida de ReLu es el valor máximo entre cero y el valor de entrada. Una salida es igual a cero cuando el valor de entrada es negativo y al valor de entrada cuando la entrada es positiva.

El propósito de ReLu

Tradicionalmente, algunas **funciones de activación no lineales prevalentes**, como las funciones sigmoidea (o logística) y tangente hiperbólica, se utilizan en redes neuronales para obtener los valores de activación correspondientes a cada neurona. Recientemente, se ha utilizado en su lugar la función ReLu para **calcular los valores de activación** en los paradigmas de redes neuronales tradicionales o redes neuronales profundas. Las razones de sustituir sigmoide y tangente hiperbólica por ReLu consisten en:

- ***Ahorro computacional:*** la función ReLu es capaz de acelerar la velocidad de entrenamiento de las redes neuronales profundas en comparación con las funciones de activación tradicionales, ya que la **derivada** de ReLu es 1 para una entrada positiva. Al ser una constante, las redes neuronales profundas no necesitan dedicar tiempo adicional al cálculo de los términos de error durante la fase de entrenamiento.

Al resolver el problema del gradiente de fuga

El problema del gradiente evanescente es un problema que surge a veces al entrenar algoritmos de aprendizaje automático mediante el descenso de gradiente. Esto ocurre con mayor frecuencia en redes neuronales que tienen varias capas neuronales, como en un sistema de aprendizaje profundo, pero también ocurre en redes neuronales recurrentes. El punto clave es que las derivadas parciales calculadas se utilizan para calcular el gradiente a medida que se profundiza en la red. Dado que los gradientes controlan cuánto aprende la red durante el entrenamiento, si los gradientes son muy pequeños o nulos, el entrenamiento será escaso o nulo, lo que dará lugar a un rendimiento predictivo deficiente.

La función ReLu no provoca el problema del gradiente de fuga cuando crece el número de capas. Esto se debe a que esta función **no tiene un límite superior e inferior asintótico**. Así, la capa más temprana (la primera capa oculta) es capaz de recibir los errores procedentes de las últimas capas para ajustar todos los pesos entre capas. Por el contrario, una función de activación tradicional como la sigmoidea está restringida entre 0 y 1, por lo que los errores se vuelven pequeños para la primera capa oculta. Este escenario dará lugar a una red neuronal mal entrenada.

2.2. Softmax

¿Qué es la función softmax?

La función softmax es una función que convierte un vector de K valores reales en un vector de K valores reales que **suman 1**. Los valores de entrada pueden ser positivos, negativos, cero o mayores que uno, pero la función softmax los transforma en **valores entre 0 y 1**, para que puedan interpretarse como **probabilidades**. Si una de las entradas es pequeña o negativa, la función softmax la convierte en una probabilidad pequeña, y si una entrada es grande, entonces la convierte en una probabilidad grande, pero siempre se mantendrá entre 0 y 1.

La función softmax se denomina a veces función **softargmax** o **regresión logística multiclase**. Esto se debe a que la softmax es una generalización de la regresión logística que se puede utilizar para la clasificación multiclase, y su fórmula es muy similar a la función sigmoidal que se utiliza para la regresión logística. La función softmax puede utilizarse en un clasificador sólo cuando las clases son **mutuamente excluyentes**.

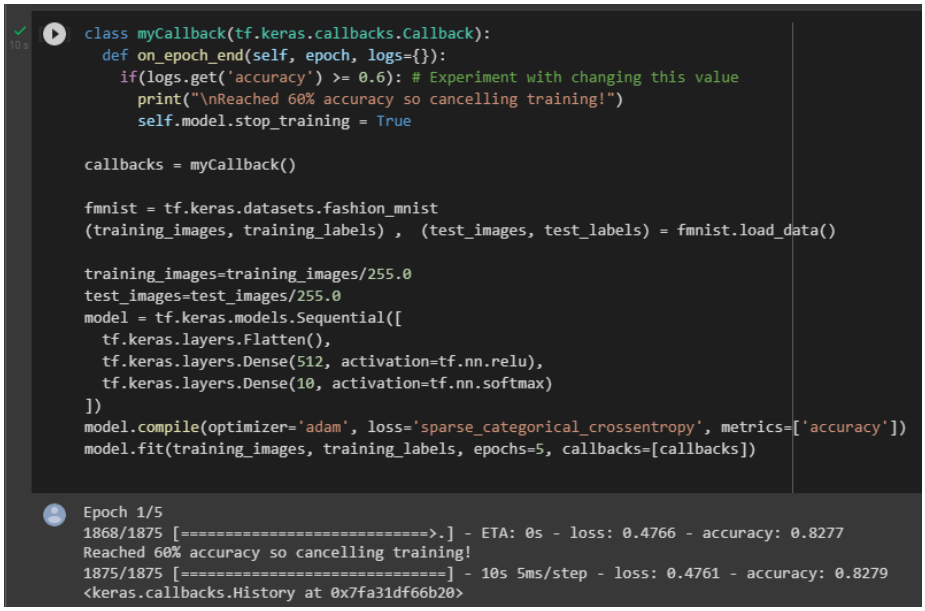
Muchas redes neuronales multicapa terminan en una penúltima capa que emite puntuaciones de valor real que no se escalan convenientemente y con las que puede ser difícil trabajar. En este caso, el softmax resulta muy útil porque convierte las **puntuaciones en una distribución de probabilidad normalizada**, que puede mostrarse a un usuario o utilizarse como entrada para otros sistemas. Por este motivo, es habitual añadir una función softmax como **capa final** de la red neuronal.

Fórmula softmax

La fórmula softmax es la siguiente: $\sigma(\overrightarrow{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_i}}$, donde todos los valores z_i son los elementos del vector de entrada y pueden tomar cualquier valor real. El término de la parte inferior de la fórmula es el término de normalización que garantiza que todos los valores de salida de la función sumen 1, constituyendo así una distribución de probabilidad válida.

3. Callbacks

Antes, cuando entrenabas por épocas extra, tenías el problema de que tu pérdida podía cambiar. Puede que te haya llevado un poco de tiempo esperar a que el entrenamiento haga eso, y puede que hayas pensado '¿no estaría bien si pudiera parar el entrenamiento cuando alcance un valor deseado?' – por ejemplo, 60 % de precisión puede ser suficiente para ti, y si lo alcanzas después de 3 épocas, ¿por qué sentarse a esperar a que termine muchas más épocas....Entonces, ¿cómo arreglarías eso? Como cualquier otro programa... ¡tienes callbacks! Veámoslas en acción...



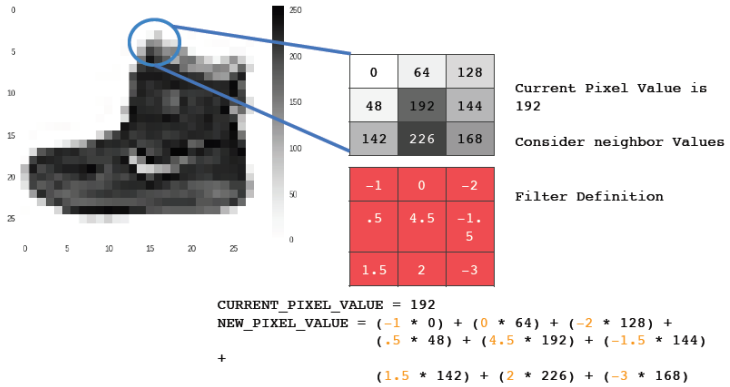
```
class myCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get('accuracy') >= 0.6): # Experiment with changing this value
      print("\nReached 60% accuracy so cancelling training!")
      self.model.stop_training = True

callbacks = myCallback()

fmnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels) ,  (test_images, test_labels) = fmnist.load_data()

training_images=training_images/255.0
test_images=test_images/255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(training_images, training_labels, epochs=5, callbacks=[callbacks])
```

```
Epoch 1/5
1868/1875 [============================>.] - ETA: 0s - loss: 0.4766 - accuracy: 0.8277
Reached 60% accuracy so cancelling training!
1875/1875 [==============================] - 10s 5ms/step - loss: 0.4761 - accuracy: 0.8279
<keras.callbacks.History at 0x7fa31df66b20>
```

4. Redes Neuronales Convolucionales

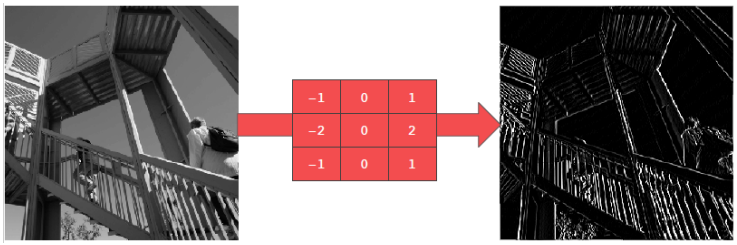
4.1. Convolución y pooling

La **convolución** es un filtro que pasas sobre una imagen de la misma forma que si estuvieras realzando los bordes. El proceso funciona un poco como esto:

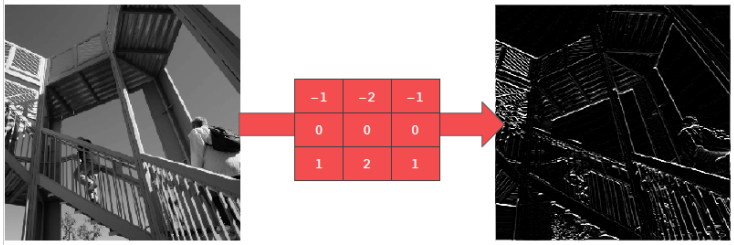


Para cada píxel, tomas su valor, y miras los valores de sus vecinos. Si nuestro filtro es tres por tres, podemos echar un vistazo al vecino inmediato, tienes la cuadrícula correspondiente de tres por tres. Para obtener el nuevo valor para el píxel, simplemente multiplicamos cada vecino por el correspondiente valor en el filtro.

La idea es que algunas convoluciones cambiarán la imagen de tal forma que ciertas características en la imagen se enfaticen.

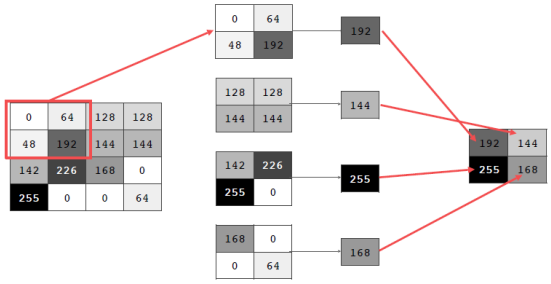


Por ejemplo si ves este filtro las líneas verticales en la imagen sobresaldrán.



Con este filtro, sobresaldrán las líneas horizontales. Esto es una introducción muy básica a lo que hacen las convoluciones, y cuando se combinan con algo llamado ***pooling***, pueden llegar a ser muy potentes. Simplemente, el pooling es una forma de comprimir una imagen.

Una forma rápida y simple de hacerlo, es ir sobre la imagen cuatro píxeles a la vez, por ejemplo, el píxel actual y sus vecinos de abajo y de su derecha. De estos cuatro, selecciona el mayor valor y guarde precisamente ese.



Lo puedes ver esta imagen. Los 16 píxeles de la izquierda se convierten en 4 píxeles de la derecha, viéndolas en cuadrículas de dos por dos y seleccionando el mayor valor. Esto preservará las características que fueron resaltadas por la convolución, a la vez que divide simultáneamente por cuatro el tamaño de la imagen. Tenemos los ejes horizontales y verticales.

```
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding='valid',
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Más info

```
tf.keras.layers.MaxPool2D(
    pool_size=(2, 2),
    strides=None,
    padding='valid',
    data_format=None,
    **kwargs
)
```

Más info

4.2. Implementación

No tenemos que hacer todas las matemáticas para filtrar y comprimir, solo definimos las capas convolucionales y de pooling para que hagan el trabajo.

Partiendo del código 2, para añadir convoluciones a esto, se usa código como este:

```
model = tk.keras.Sequential([
    tf.keras.layers.Conv2D(64,(3,3),activation='relu', input_shape
        =(28,28,1)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64,(3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

Verás que las últimas líneas son las mismas, la capa Flatten, la capa Dense oculta con 128 neuronas, y la capa de salida Dense con 10 neuronas. La diferencia es lo que se ha añadido encima de esto. Echemos un vistazo a esto, línea por línea:

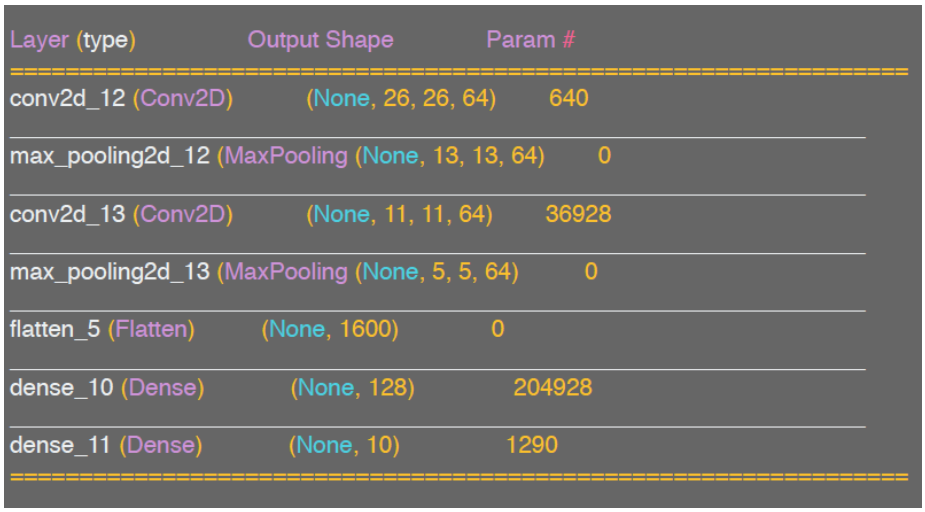
[1] Estamos especificando la primera convolución generando 64 filtros. Estos filtros son 3 por 3, su activación es relu, es decir los valores negativos se descartarán, finalmente la forma de la entrada es como antes, 28 por 28. Ese 1 extra solo significa que estamos usando **un solo byte para el color**. Como vimos antes nuestra imagen es en escala de grises, y solo usamos un byte.

[2] Estamos agregando una capa de max pooling al modelo. La capa de max pooling es utilizada para reducir la dimensionalidad de los datos, tomando el valor máximo de cada submatriz de tamaño (2,2) en la capa anterior. En este caso, se está tomando el valor máximo de cada submatriz de 2x2 y se está reemplazando por ese valor, reduciendo así el número de elementos en la capa anterior en un factor de 4. Es comúnmente utilizado para **reducir el sobreajuste** en redes neuronales y mejorar el rendimiento del modelo.

[3] Es similar a la línea **[1]**, pero en esta ocasión no se especifica el parámetro input_shape, lo que significa que esta capa se asume como una capa adicional en la arquitectura, y se espera que reciba una **entrada con las mismas dimensiones que la salida de la capa anterior**.

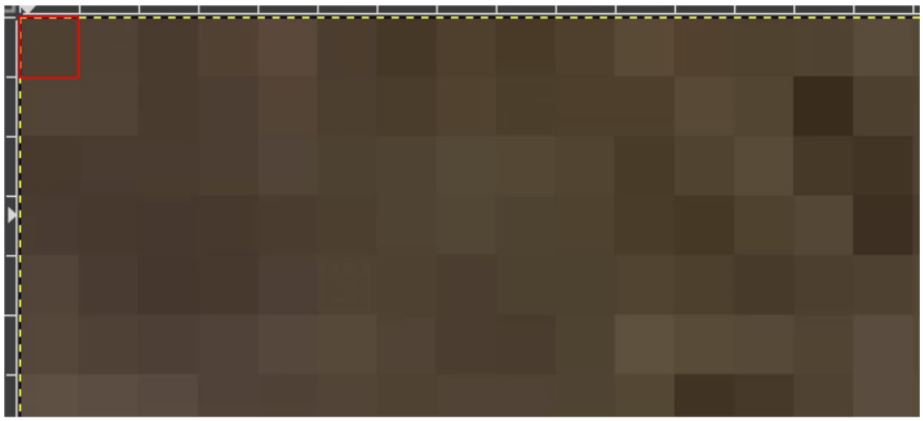
En el momento que la imagen llega a *Flatten()* para entrar a las capas densas, ya es mucho más pequeña. Su contenido ha sido enormemente simplificado, el objetivo es que las convoluciones van a filtrar las características que determinan la salida.

Un método realmente muy útil en este modelo es el método ***model.summary()***. Esto te permite inspeccionar las capas del modelo, y ver el trayecto de la imagen a través de las convoluciones.



Esta es la salida, una buena tabla mostrándonos las capas, y algunos de sus detalles incluyendo la forma de la salida. Es importante prestar atención a la columna de la forma de la salida (*Output Shape*).

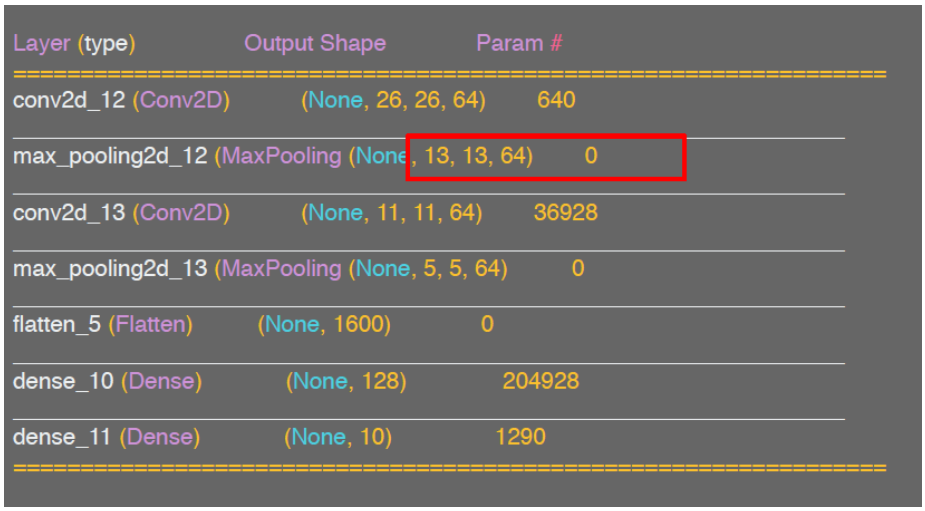
Cuando lo ves por primera vez, puede ser un poco confuso y sentirlo como un error. Después de todo, no son los datos de 28 por 28, entonces por qué la salida es de **26 por 26**. La clave para esto es recordar que el filtro es un filtro **3 por 3**. Considera lo que sucede cuando empiezas a barrer una imagen empezando por la parte superior izquierda.



No puedes calcular el filtro para el píxel de la parte superior izquierda, porque no tiene ningún vecino ni encima, ni a su izquierda. De forma similar, el próximo pixel a su derecha no funcionará tampoco porque no tiene vecinos encima de él. Así, el primer píxel con el que puedes hacer cálculos es el de la siguiente imagen, que tiene los **8 vecinos** que un filtro 3 por 3 necesita.

Esto, cuando lo piensas, significa que **no puedes usar un margen de un píxel** alrededor de toda la imagen, para que la salida de la convolución sea dos píxeles más pequeña en x, y dos píxeles más pequeña en y. Si tu filtro fuera de **5 por 5** por razones similares tu salida será **4 píxeles más pequeña** en x y cuatro más pequeña en y.

Este es el motivo por el que nuestra salida, partiendo de la imagen de 28 por 28, es ahora de 26 por 26, hemos quitado un píxel en x e y, y en cada uno de los bordes.



Lo siguiente es la primera de las capas de max-pooling. Recuerda que especificamos que fuera de 2 por 2, transformando así 4 píxeles en 1. Nuestra salida queda reducida de 26 por 26, a **13 por 13**.

Las convoluciones van a trabajar sobre eso, y por supuesto perdemos el píxel de margen como antes, llegamos a **11 por 11**, añades otra max-pooling de 2 por 2 para tenerlo redondeado hacia abajo, y llegar a bajar a imágenes de **5 por 5**.

La red neuronal es la misma que antes, pero está alimentada con imágenes de 5 por 5 en vez de las de 28 por 28. Pero recuerda, no es solo una imagen comprimida de cinco por cinco en vez de la original de 28 por 28, hay un número de convoluciones por imagen que hemos especificado, en este caso 64. Hay 64 nuevas imágenes de cinco por cinco que han sido suministradas. Hacemos Flatten() en eso y tienes 25 píxeles por 64, que son **1600**, contra los 784 que tenía antes.

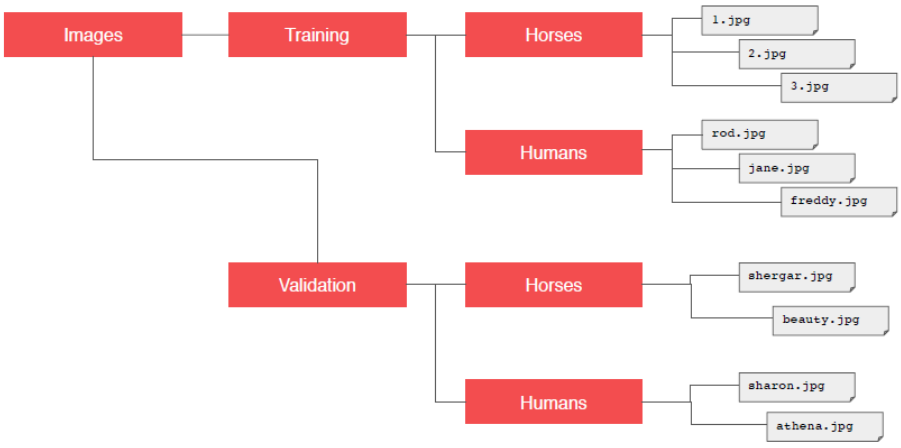
De forma no visible, TensorFlow prueba diferentes filtros en tu imagen y aprende cuáles funcionan mientras mira los datos de entrenamiento. Como resultado, cuando funciona, obtendrás una gran reducción de la información que pasa a través de la red, porque aísla e identifica las características, puedes también obtener un incremento de la precisión.

5. Usando imágenes del mundo real

¿Qué sucede si se usan imágenes más grandes donde las características pueden estar en diferentes posiciones?

El sujeto puede tener diferentes tamaños y diferentes proporciones de aspecto y estar en diferentes posiciones. En algunos casos, puede haber hasta múltiples sujetos. Además, los ejemplos anteriores con datos de moda usaban un conjunto de datos incorporado. Todos los datos estaban divididos fácilmente en conjuntos de entrenamiento y conjuntos de prueba y había etiquetas disponibles. En muchos escenarios ese no será el caso y tendrás que hacerlo tú mismo. En esta lección, echaremos un vistazo a algunos las API disponibles para hacértelo más fácil. En particular, el **generador de imágenes de TensorFlow**.

Una característica del generador de imágenes es que puedes dirigirla a un directorio, luego sus subdirectorios generarán automáticamente las etiquetas para ti.



Por ejemplo, considera esta estructura de directorios. Tienes un directorio de imágenes y en él, tienes subdirectorios para entrenamiento y validación. Cuando pones subdirectorios en ellos para caballos y humanos y almacenas las imágenes requeridas ahí, el generador de imágenes puede crear un alimentador para esas imágenes y las etiqueta automáticamente para ti. Por ejemplo, si dirijo un generador de imágenes al directorio de entrenamiento, las etiquetas serán caballos y humanos y todas las imágenes en cada directorio serán cargadas y etiquetadas como corresponde. Similarmente, si dirijo uno al directorio de validación pasará lo mismo.

Echemos un vistazo a esto en el código:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGeneration(rescale=1./255)

train_generator = train_datagen.flow_from_directory(train_dir,
    target_size=(300,300), bath_size=128, class_mode='binary')
```

Pasamos ***rescale*** para normalizar los datos.

Podemos llamar al método ***flow_from_directory()*** para que cargue las imágenes de ese directorio y sus subdirectorios. Es un error común que la gente dirija el generador al subdirectorio. Fallará si se hace así. Debes siempre **apuntar al directorio que contiene los subdirectorios** que contienen las imágenes.

El **nombre de los subdirectorios** serán las **etiquetas** de las imágenes que están contenidas en ellos. Asegúrate de que el directorio al que te diriges es el correcto.

Las imágenes podrán venir en todo tipo de formas y tamaños y desafortunadamente para entrenar una red neuronal, los datos de entrada tienen que ser todos del mismo tamaño, las imágenes deberán ser redimensionada para hacerlas consistentes. Lo bueno de este código es que las imágenes son **redimensionadas** (***target_size***) por ti cuando son cargadas. De esta forma, puedes experimentar con tamaños diferentes sin afectar los datos originales.

Las imágenes serán cargadas para el entrenamiento y validación en **lotes** (***batch_size***), que es más eficiente que hacerlo una por una. Las imágenes serán cargadas para el entrenamiento y validación en lotes lo que es más eficiente que hacerlo una por una.

Finalmente, está ***class_mode***, en este caso es un clasificador binario, elige entre dos cosas distintas; caballos y humanos. Otras opciones en particular para más de dos cosas se explorarán más adelante en el curso.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGeneration(rescale=1./255)

validation_generator = train_datagen.flow_from_directory(
    validation_dir, target_size=(300,300), bath_size=32, class_mode
    ='binary')
```

El generador de **validación** debe ser **exactamente lo mismo** excepto, por supuesto, que se dirige a un directorio diferente, el que contiene los subdirectorios que contienen las imágenes de prueba.

5.1. Diseñar NN para manejar imágenes más complejas

```
model = tk.keras.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.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Diferencias respecto al caso anterior:

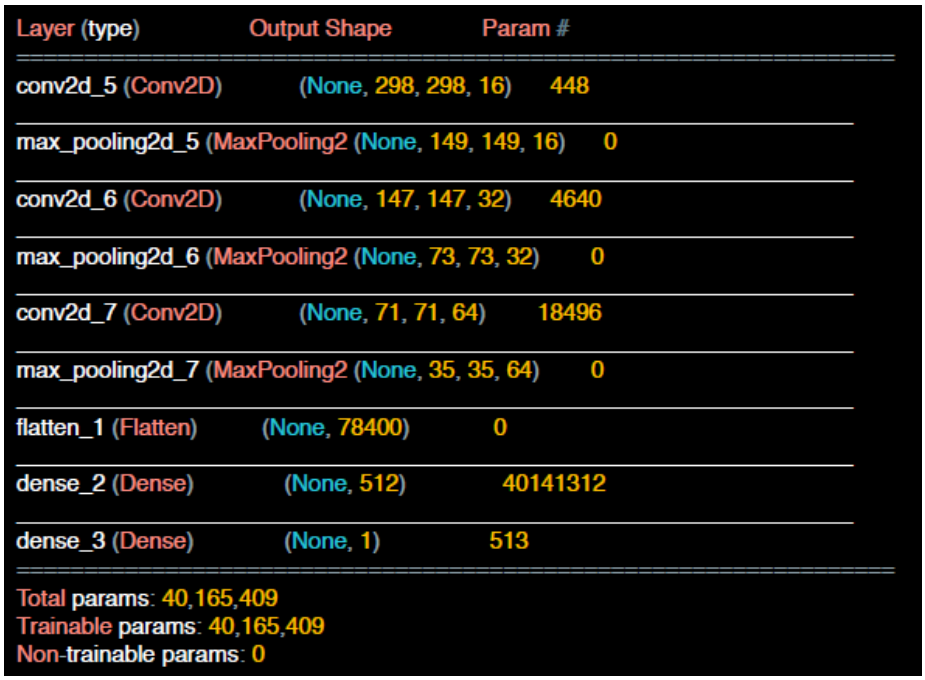
1. Hay **3** conjuntos de convolución y pooling arriba. Esto refleja la mayor complejidad y tamaño de las imágenes. Recuerda que antes teníamos imágenes de 28x28, ahora de 300x300. Podemos incluso añadir otro par de capas si deseamos llegar al mismo tamaño de antes, pero lo mantendremos en 3 por ahora.

2. La **forma de la entrada**. Redimensionamos la imagen para que sea de **300x300** cuando sean cargadas, pero también son imágenes de color. Son **3** bytes por píxel. Un byte para el rojo, un byte para el verde, y un byte para el canal azul, este es un patrón común de color de 24-bit.

3. La **capa de salida**. Recuerda que antes cuando creaste la capa de salida, tenías una neurona por clase, pero ahora solo hay **una neurona para dos clases**. Eso es porque estamos usando una función de activación diferente que es **sigmoid** que es genial para la clasificación **binaria**, cuando una clase tiende a cero, la otra clase tiende a uno. Podrías usar dos neuronas si lo deseas, y la misma función softmax como antes, pero para binary esto es un poco más eficiente.

Si echamos un vistazo al resumen de nuestro modelo, podemos ver el trayecto de los datos de la imagen a través de las convoluciones.

Los 300 por 300 se convierten en 298 por 298 después del filtro de tres por tres, se lleva a 149 por 149 que a su vez se reduce a 73 por 73 después del filtro que luego es llevada a 35 por 35, esto luego pasa por Flatten(), 64 convoluciones que son 35 al cuadrado y esa forma será alimentada a la DNN. Si multiplicas 35 por 35 por 64, obtienes 78400 y esa es la forma de los datos una vez que sale de las convoluciones. Si solo hubiéramos suministrado imágenes sin tratar de 300 por 300 sin las convoluciones, eso sería más de 900000 valores. Ya lo hemos reducido bastante.



5.2. Entrenar NN

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

Ahora compilaremos el modelo y como siempre tenemos una función de pérdida y un optimizador. Como estamos haciendo una elección binaria, escogemos una pérdidA **binary_cross_entropy**. Con optimizador **RMSprop**, podemos ajustar la tasa de aprendizaje para experimentar con el rendimiento.

```
1 history = model.fit(train_generator, steps_per_epoch=8, epochs=15,
      validation_data=validation_generator, validation_steps=8,
      verbose=2)
```

Lo siguiente es el entrenamiento, esto parece un poco diferente al de antes cuando llamábamos a model.fit. Porque ahora llamas a **model.fit_generator,** eso es porque estamos usando un generador en vez de un conjunto de datos. Veamos cada parámetro en detalle.

[1] El primer parámetro es **train_generator** que configuraste anteriormente. Esto hace fluir las imágenes desde el directorio de entrenamiento.

[2] Recuerda el tamaño del lote que usaste cuando lo creaste, era 128, eso es importante en el siguiente paso. Hay 1.024 imágenes en el directorio de entrenamiento, estamos cargándolas de 128 en 128 cada vez. Para cargarlas todas tenemos que hacerlo en 8 lotes. Configuramos **steps_per_epoch** para lograrlo.

[3] Con **epochs** configuramos el número de épocas a entrenar. Esto es un poco más complejo, digamos, 15 epochs en este caso.

[4] Ahora especificamos el conjunto de validación que viene del **validation_generator** que creamos anteriormente.

[5] Tenía 256 imágenes y queríamos tratarlas en lotes de 32, haremos 8 pasos con el parámetro **validation_steps**.

[6] El parámetro verbose especifica cuánto mostrar mientras se entrena. Con verbose puesto a 2, obtenemos menos animación ocultando el progreso del epoch.

Para realizar predicciones podemos usar el siguiente código:

```
import numpy as np
from google.colab import files
from keras.preprocessing import image

uploaded = files.upload()

for fn in uploaded.keys():

    #predict images
    path = '/content/' + fn
    img = image.load_img(path, target_size=(300,300))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)

    images = np.vstack([x])
    classes = model.predict(images, batch_size=10)

    if classes[0] > 0.5:
        print(fn + " is a human")
    else:
        print(fn + " is a horse")
```

El bucle itera a través de todas las imágenes de esta colección. Asegura que las **dimensiones coinciden** con las dimensiones de la entrada que especificaste cuando diseñaste el modelo.

Puedes llamar a **model.predict**, devolverá un vector de clases. En el caso de clasificación binaria, este solo contendrá un artículo con un valor cercano a 0 para una clase y cercano a 1 para la otra.

5.3. Binary Crossentropy & RMSprop

La función de pérdida **Binary Crossentropy** es una función de pérdida utilizada en modelos de clasificación binaria, es decir, en los que el objetivo es clasificar una entrada en una de dos categorías. Se utiliza para medir la **distancia** entre la **distribución de probabilidad predicha** y la **verdadera distribución** en la clasificación binaria. La forma en que se calcula es comparando la probabilidad predicha para cada clase y la verdadera etiqueta: $CE = -\sum_i^C t_i * log(s_i)$, donde t_i y s_i son la **verdad básica** y el **puntaje de CNN** para cada clase i en C. En un problema de clasificación binaria $C = 2$. Se define como la **media negativa de la entropía cruzada binaria** para cada dato en el conjunto de entrenamiento.

Esta función se utiliza para optimizar el modelo durante el entrenamiento y su objetivo es minimizarse para mejorar la precisión de las predicciones. Cuanto más pequeño sea el valor de la función de pérdida, mejor será la capacidad del modelo para predecir la clase correcta.

RMSprop (*Root Mean Squared Propagation*) es un optimizador de descenso de gradiente que se utiliza en problemas de aprendizaje profundo. La idea es **ajustar la tasa de aprendizaje** en cada iteración para asegurarse de que los pesos de la red neuronal se actualizan de manera eficiente. El optimizador mantiene una media móvil de los valores recientes de la tasa de aprendizaje y los gradientes, y ajusta la tasa de aprendizaje para cada peso en función de la media móvil.

Esto ayuda a resolver dos problemas que pueden surgir durante el entrenamiento de una red neuronal: la oscilación y la convergencia lenta. La **oscilación** ocurre cuando la tasa de aprendizaje es demasiado alta, lo que hace que los pesos se actualicen de manera errática y no converge a una solución óptima. La **convergencia** lenta ocurre cuando la tasa de aprendizaje es demasiado baja, lo que hace que el proceso de aprendizaje sea lento.