Clasificación multiclase

Castro Granados Celeste, celsgazu@ciencias.unam.mx
Valdez Astudillo Almendra María, almendramva@ciencias.unam.mx
Seminario de Machine Learning. Proyecto Final

1. INTRODUCCIÓN

El arreglo de transformaciones lineales seguidas de una traslación, conocidas como transformaciones afínes T, se denota $\Phi = (T_1, T_2, .., T_L)$ y se le conoce como red neuronal. Su distribución es la siguiente,

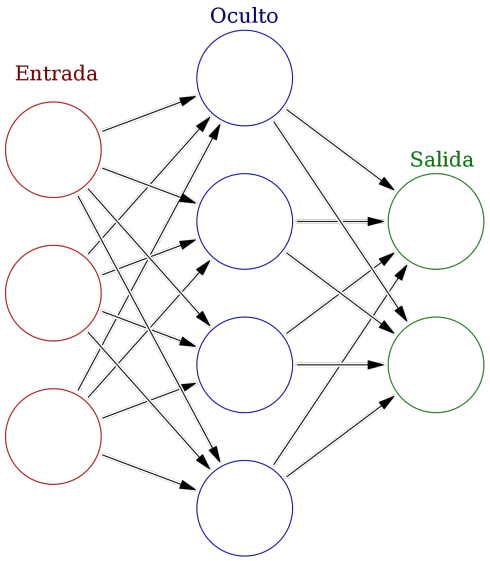

Fig. 1: Red Neuronal

en donde, las flechas negras que pasan de `Entrada` a `Oculto` y finalizan en `Salida` denotan a las transformaciones afines T_i, con $i \in \{1, ..., L\}$, que hacen que el arreglo funcione; la cantidad de T's son los niveles o capas de una red neuronal.

En general, una red neuronal es un modelo que trata de imitar el modo en que el cerebro procesa la información, en este caso para que procese información específica, y para poder usar el modelo es necesario que sigan algún algoritmo de aprendizaje, es decir, entrenarlo (al igual que el cerebro).

La primera capa de un modelo tiene datos (también llamados pesos) de entrada que al pasar por las transformaciones afines de esa capa, llega a la siguiente capa y así sucesivamente por las capas ocultas, hasta terminar en la última capa y ser los resultados del modelo. Los pesos avanzan entre capas por medio de la composición con una función de activación que facilita su intercambio y los ajusta para minimizar el error de la salida, para esto es necesario un algoritmo de optimización que ajuste los pesos. Ejemplos de funciones de activación son función logística, gaussiana, lineal, etc. Ejemplos de algoritmos de optimización es el descenso del gradiente estocástico, backpropagation, etc.

Hay varias formas de clasificar a las redes neuronales, se pueden etiquetar por el número de capas: monocapa o multicapa; por los tipos de conexiones: recurrentes o no; por el número de conexiones: totalmente conectada o parcialmente conectado.

Dentro de los tipos de redes neuronales multicapa existe la que se aplica en pesos que son matrices bidimensionales, como en la clasificación y segmentación de imágenes, el modelo que responde a estos pesos es la red neuronal convolucional.

Las redes neuronales pueden implementarse utilizando lenguajes de programación, el más utilizado actualmente es `Python`. Python es un lenguaje orientado a objetos y tiene programación funcional, también contiene librerías que son un conjunto de implementaciones funcionales bien definidas; para fines de este proyecto usaremos la librería de `TensorFlow` que tiene código abierto que permite construir y entrenar redes neuronales.

`TensorFlow` es una librería de `Python` creada por Google en el 2015 para poder construir y entrenar redes neuronales, que a su vez tiene librerías de código abierto, por ejemplo `Keras`, que se ejecutan cuando se llama a `TensorFlow`. *Keras* es una librería escrita en `Python` por Francois Chollet en 2015 que tiene definidas funciones de activación, algoritmos de optimización, transformaciones afínes, etc. El código de `Keras` está disponible en GitHub [1] y en otras plataformas para su soporte y mantenimiento. Dentro de `Python`, `TensorFlow` y `Keras` se importan de la siguiente manera,

```
import tensorflow as tf
from tensorflow import keras
```

Fig. 2: Importación de librerías

2. PLANTEAMIENTO DEL PROBLEMA

En este proyecto se busca construir y entrenar una red neuronal (utilizando la librería de Tensorflow, específicamente Keras), que sea capaz de clasificar imágenes variadas en clases definidas previamente.

Específicamente, se trabajará con el dataset conocido como CIFAR10 [4], el cual consta de 50,000 (32x32) imágenes de entrenamiento (`train_images`) a color y 10,000 imágenes de prueba (`test_images`) también a color, etiquetadas en 10 clases (`train_labels` y `test_labels`) respectivamente.

Como se trata de imágenes a color, las imágenes tendrán una dimensión igual a 3 puesto que siguen el formato RGB (cada dimensión corresponderá a uno de estos canales), y sus valores de pixel irán de 0 a 255. Por otro lado, las etiquetas serán enteros del 0 al 9, ya que se tienen 10 clases distintas.

A continuación se muestran los nombres de las clases:

Label	Class
0	avión
1	carro
2	pájaro
3	gato
4	ciervo
5	perro
6	rana
7	caballo
8	barco
9	camión

Fig. 3: Nombres de las clases

3. METODOLOGÍA

La metodología seguida para abordar este problema estuvo basada en el notebook desarrollado por Keras titulado ”Clasificación Básica: Predecir una imagen de moda” [5], la cual fue dividida en las siguientes subsecciones:

1. Exploración del dataset

Previo a construir un modelo es importante conocer un poco los datos que se tienen, por lo cual, se imprimieron la forma de las imágenes y de sus etiquetas, paso en donde se observó que el conjunto de etiquetas estaba estructurado como un arreglo de listas con un solo elemento. Debido a lo anterior fueron redefinidos los conjuntos de etiquetas de tal manera que contuvieran solo los números para facilitar su acceso más adelante.

Asimismo, se graficó una de las imágenes para poder apreciarla de mejor manera, misma que se muestra a continuación:

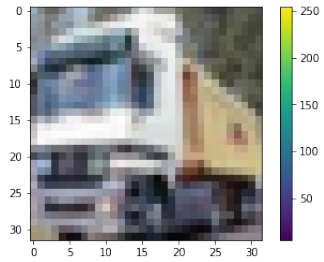

Fig. 4: Imagen 1 del conjunto de entrenamiento

2. Pre-procesamiento

El *Pre-procesamiento del dataset* es un paso preliminar que se hace buscando el acondicionamiento de las imágenes con el fin de garantizar la correcta extracción de características por parte de la red neuronal. Específicamente, lo que se realizó fue una normalización de los valores de los pixeles de todas las imágenes con el fin de evitar la aparición de un sesgo en los datos. Adicionalmente se mostraron algunas imágenes de entrenamiento con el nombre de su clase debajo para asegurarnos de que el dataset estaba listo y en el formato adecuado. A continuación se muestran 4 de las imágenes que se mostraron en el código:

Fig. 5: Imágenes con sus respectivas etiquetas

3. Modelo

1. Construcción

Como se trata de un problema de clasificación de imágenes se utilizó una red convolucional para abordarlo. Para definir el modelo se tomó como referencia la red neuronal convolucional encontrada en el repositorio de Github de Francois Chollet en el capítulo 8 "Introduction to deep learning for computer vision" [6], en la cual se ajustaron solo algunos parámetros para que pudiera trabajar con el tamaño de nuestras imágenes y nos devolviera un total de 10 salidas. La siguiente figura muestra un resumen del modelo utilizado.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 3)]       0

conv2d (Conv2D)              (None, 30, 30, 32)        896

max_pooling2d (MaxPooling2D  (None, 15, 15, 32)        0
)

conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496

max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)          0
2D)

conv2d_2 (Conv2D)            (None, 4, 4, 128)         73856

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 10)                20490

=================================================================
Total params: 113,738
Trainable params: 113,738
Non-trainable params: 0
_________________________________________________________________
```

Fig. 6: Resumen del modelo

La primera capa corresponde a una capa de convolución 2d (espacial), este tipo de capas consisten en tomar «grupos de pixeles cercanos» de la imagen de entrada e ir operando matemáticamente (producto escalar) contra una pequeña matriz que se llama kernel. Ese kernel logra «visualizar» todas las neuronas de entrada (de izquierda-derecha, de arriba-abajo) y así logra generar una nueva matriz de salida, que será la nueva capa de neuronas ocultas.

Igualmente se incluyeron capas de MaxPooling con un tamaño de 2×2. Esto quiere decir que recorre cada una de las imágenes de características obtenidas anteriormente de izquierda-derecha, arriba-abajo PERO en vez de tomar de a 1 pixel, tomas de «2×2» (2 de alto por 2 de ancho = 4 pixeles) y va preservando el valor «más alto» de entre esos 4 pixeles (por eso lo de «Max»). En este caso, la imagen resultante es reducida «a la mitad» pero sigue almacenando la información más importante para detectar características deseadas.

La capa de Flatten "aplasta" todos los valores para que puedan ser introducidos en la última capa (capa densa), la cual es una capa regular y será la encargada de arrojar el resultado final, razón por la cual tiene 10 salidas (tenemos 10 clases).

Finalmente, como función de activación de la primera capa y las capas ocultas se utilizó la función relu, la cual transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como entran. Algunas carcaterísticas de esta función son las siguientes:

- Activación Sparse – solo se activa si son positivos.
- No está acotada.
- Se pueden morir demasiadas neuronas.
- Se comporta bien con imágenes.
- Buen desempeño en redes convolucionales.

Por otro lado, en la capa de salida la función de activación fue la función softmax, este cambio fue debido a que esta capa transforma las salidas a una representación en forma de probabilidades, de tal manera que la suma de todas las probabilidades de las salidas sea 1 (esto nos permite interpretar las salidas como la probabilidad que tiene la imagen de pertenecer a cada clase). Sus características son:

- Se utiliza cuando queremos tener una representación en forma de probabilidades.
- Esta acotada entre 0 y 1.
- Muy diferenciable.
- Se utiliza para para normalizar tipo multiclase.
- Buen rendimiento en las últimas capas.

2. Compilación

Antes de que el modelo se entrenara, fue necesario especificar algunas configuraciones durante el paso de compilación (`metodo.compile(optimizer,loss,metrics=[])`).

El optimizador que se ocupó fue el método del gradiente estocástico que dentro de `Keras` se llama `optimizer='adam'`, el cual actualiza con base a las imágenes.

Dentro de `Keras`, `loss` se refiere a la función de pérdida (función de costo) y se usó `loss='sparse_categorical_crossentropy'`; como su nombre lo dice categoriza la entropía cruzada entre las etiquetas, puesto que se trata de un problema que es de clasificación multiclase (labels) y sus predicciones. La diferencia de esta función y *categorical_crossentropy* es que en este caso no es necesario tener etiquetas categóricas.

Por último, `metrics=[]` se refiere a la función utilizada para ajustar el funcionamiento del modelo, la métrica utilizada fue el `metrics=['accuracy']`, el cual calcula que tan frecuente las predicciones y las etiquetas (labels) son iguales.

4. Entrenamiento

Se realizó el entrenamiento del modelo con `model.fit(x,y,epochs)` en el conjunto de entrenamiento `(train_ images,train_labels)` previamente definido.

En el método `.fit(x,y,epochs)`, `epochs` corresponde al número de veces que se va a iterar el modelo sobre el conjunto de entrenamiento; en este caso se tomó un valor de `epochs=10`.

El resultado del proceso de entrenamiento se muestra a continuación,

```
#Se entrenará durante 10 épocas
model.fit(train_images,train_labels, epochs=10)
```

```
Epoch 1/10
1563/1563 [==============================] - 72s 45ms/step - loss: 1.4719 - accuracy: 0.4704
Epoch 2/10
1563/1563 [==============================] - 70s 45ms/step - loss: 1.0855 - accuracy: 0.6188
Epoch 3/10
1563/1563 [==============================] - 70s 45ms/step - loss: 0.9377 - accuracy: 0.6723
Epoch 4/10
1563/1563 [==============================] - 70s 45ms/step - loss: 0.8453 - accuracy: 0.7056
Epoch 5/10
1563/1563 [==============================] - 70s 45ms/step - loss: 0.7745 - accuracy: 0.7334
Epoch 6/10
1563/1563 [==============================] - 70s 45ms/step - loss: 0.7149 - accuracy: 0.7539
Epoch 7/10
1563/1563 [==============================] - 70s 45ms/step - loss: 0.6692 - accuracy: 0.7688
Epoch 8/10
1563/1563 [==============================] - 70s 45ms/step - loss: 0.6203 - accuracy: 0.7845
Epoch 9/10
1563/1563 [==============================] - 69s 44ms/step - loss: 0.5796 - accuracy: 0.7983
Epoch 10/10
1563/1563 [==============================] - 69s 44ms/step - loss: 0.5446 - accuracy: 0.8094
<keras.callbacks.History at 0x7efef4c4dd90>
```

Fig. 7: Valores obtenidos en cada época

En la figura 7 podemos observar el tiempo en que cada `epoch` se realiza, una salida asociada al valor de `loss` y otra al valor del `accuracy`, mismo que fue creciendo conforme `epoch` va aumnetando y `loss` disminuye.

El valor del accuracy obtenido tras completar todas las épocas fue de `accuracy = 80.94` %.

5. Evaluación del modelo

Para este paso usamos el método `model.evaluate(x,y,verbose)` sobre el conjunto de prueba.

En el método `.evaluate()`, `x` es la el conjunto de imágenes de entrada `test_images`, mientras que `y` es la lista de las etiquetas asociadas `test_labels`, y `verbose` se refiere a la manera en que se quiera visualizar el proceso (puede tomar tres valores distintos, $\{0, 1, 2\}$), en donde 0 es conocido como *silencioso* pues no muestra la salida, 1 tiene como salida una barra de progreso animada y 2 genera una fila de registros de cada iteración. El valor que se tomó para el verbose fue de 2.

Como resultado de éste método se obtiene el valor de pérdida (o costo), el valor de `accuracy` (exactitud) así como cuánto tiempo le tomó al modelo calcular estos valores en cada iteración (`epochs`).

A continuación se muestra lo obtenido:

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

```
313/313 - 4s - loss: 0.8831 - accuracy: 0.7191 - 4s/epoch - 13ms/step
```

Fig. 8: Proceso de evaluación del modelo

```
print('Test de exactitud = ', test_acc)
```

```
Test de exactitud =  0.7190999984741211
```

Fig. 9: Accuracy para el conjunto de prueba

La diferencia observada en el `accuracy` respecto al conjunto de entrenamiento se debe al *overfittin* (o sobreajuste) que se da al sobre-entrenar el modelo.

4. PREDICCIONES

Una vez completados todos los pasos anteriores, el modelo ya se encuentra listo para hacer predicciones.

El método `model.predict(x)` tiene como salida lo que el modelo genera para el argumento `x`, que en este caso corresponde a las etiquetas predichas sobre las imágenes de entrada (`test_images`). La salida resulta ser una matriz n-dimensional debido a que `test_images` es una matriz n-dimensional, en donde cada entrada corresponde a las probabilidades asociadas a cada imagen.

```
predicciones[1]
```

```
array([1.0683586e-02, 6.7753266e-03, 8.4815230e-09, 1.5398994e-09,
       1.2069992e-09, 7.8979193e-12, 5.4637874e-11, 4.0792381e-10,
       9.8250520e-01, 3.5804183e-05], dtype=float32)
```

Fig. 10: Arreglo de predicciones para la imágen 1

En otras palabras, cada entrada consta de un `array` de 10 números que son el nivel de seguridad (confianza) que tiene el modelo de que la imágen pertenezca a alguna clase (’avion’,’carro’,’pájaro’,.. etc), por lo que para conocer el resultado final de la predicción es necesario aplicar el método `argmax()`.

Con el fin de observar las predicciones en `test_images` se graficaron los resultados del modelo mostrando el elemento en `test_images` y una gráfica de barras que pinta de azul (si el modelo hace una buena predicción) la clase a la que

pertenece este elemento, o una barra roja si el modelo no hizo una buena predicción. Para esto, se construyeron dos funciones con la librería `matplotlib`. A continuación se muestra una de estas predicciones individual y un conjunto de varias predicciones:

```
i = 1
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predicciones[i], test_labels_final, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predicciones[i],  test_labels_final)
plt.show()
```

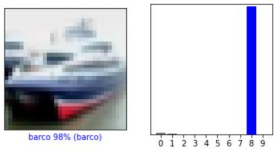

Fig. 11: Predicción individual

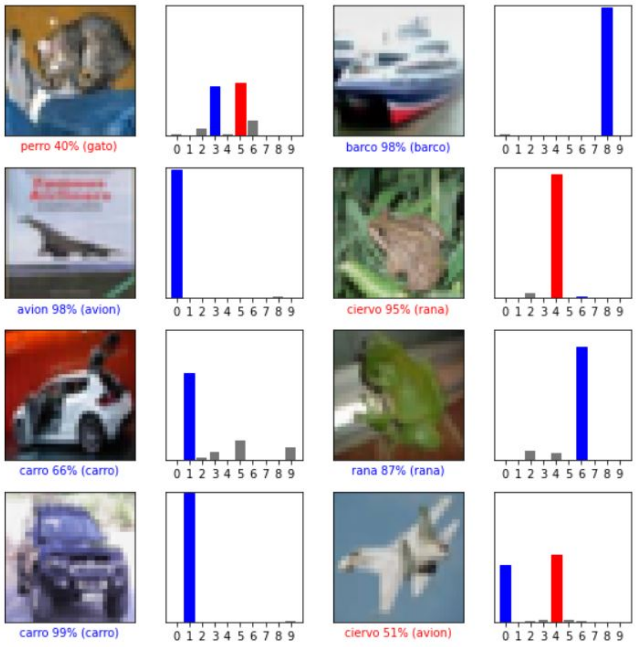

Fig. 12: Múltiples predicciones

Para terminar esta sección se calculó la matriz de confusión, misma que permite ver el desempeño del modelo en el conjunto de prueba (`test_images`) al comparar las etiquedas predichas.

La `matriz_confusion` obtenida fue la siguiente:

```
tf.Tensor(
[[808  28  33   7  31   3   5  16  44  25]
 [ 22 882   7   2   7   3   9   3  14  51]
 [ 73  11 619  31  96  61  65  25   9  10]
 [ 32  21  85 372 112 180 110  51  16  21]
 [ 21   6  51  28 735  26  52  68  10   3]
 [ 19   4  62 108  74 617  41  56  10   9]
 [  4   9  53  19  59  16 819  10   8   3]
 [ 16   4  24  23  74  55   7 782   5  10]
 [101  33  13   6   9   7   6   4 795  26]
 [ 40 120   8   6   7   8   8  15  26 762]], shape=(10, 10), dtype=int32)
```

Fig. 13: Matriz de confusión

En donde, la cantidad de predicciones correctas del modelo en el conjunto de prueba es la traza de la matriz, por lo que el resultado final fue el siguiente,

```
pred_correctas= np.sum(np.diagonal(matriz_confusion))
pred_incorrectas=len(test_images)-pred_correctas
print("Total de imágenes: ", len(test_images))
print("Total de predicciones correctas: ",pred_correctas)
print("Total de predicciones incorrectas: ",pred_incorrectas)
```

```
Total de imágenes:  10000
Total de predicciones correctas:  7191
Total de predicciones incorrectas:  2809
```

Fig. 14: Número de predicciones

5. CONCLUSIONES

Después de entrenar, evaluar y ajustar el modelo en la base de datos CIFAR10 se obtuvieron buenos niveles de accuracy (exactitud) tanto para el entrenamiento como para el conjunto de prueba, ambos cerca del 80 %, que nos permite concluir que el modelo aprendió correctamente y funciona bien para realizar predicciones sobre imágenes nuevas. Esto es en parte por la gran cantidad de imágenes con las que se contaba pues redujo el nivel de overfitting, y por la elección de una red convolucional como modelo.

Se notó que el nivel de exactitud del modelo cambiaba cada vez que se compilaba de nuevo, porque el orden de los conjuntos de prueba y entrenamiento es aleatorio, entonces no se puede esperar obtener exactamente el mismo valor siempre, sin embargo, la variación no es significativa y en todos los casos se obtuvieron buenos resultados.

Finalmente, para mejorar el modelo se propone aumentar el número de épocas en el proceso de entrenamiento con el fin de que la red pueda mejorar el aprendizaje de los parámateros aún más.

Nota: El notebook de Google Colab referente a este reporte puede ser consultado en el siguiente link: `https://colab.research.google.com/drive/1grLPstuAzMTRnROi4dM4auut_2Pe7oNZ?usp=sharing`

REFERENCIAS

[1] Página en GitHub del código de Keras. URL: https://github.com/keras-team/keras

[2] Página oficial de la librería Keras. URL: https://keras.io/

[3] Sayak Paul. (30/06/2021). *Compact Convolutional Transformers*, 20/01/2022. Recuperado de https://keras.io/examples/vision/cct/

[4] Keras API Reference. Built-in small datasets. *CIFAR10 small images classification dataset*. Recuperado de: https://keras.io/api/datasets/cifar10/

[5] TensorFlow. Aprende. TensorFlow Core. Instructivos. Principiante. Conceptos básicos de AA con Keras. Clasificación de imágenes básicas. *Clasificación Básica: Predecir una imagen de moda*. Recuperado de: https://www.tensorflow.org/tutorials/keras/classification?hl=es-419

[6] Chollet, F. (2021). *Companion Jupyter notebooks for the book "Deep Learning with Python"*. Recuperado de: https://github.com/fchollet/deep-learning-with-python-notebooks