Sistemas de Aprendizaje Automático

Conforme a contenidos del «Curso de Especialización en Inteligencia Artificial y Big Data»

Universidad de Castilla-La Mancha

Escuela Superior de Informática
Ciudad Real

Índice general

Listado de acrónimos

Capítulo 9

Deep Learning

Dentro de las redes neuronales hay una gran variedad de tipos desde Redes Neuronales Recursivas (RNN)[DBDJH14] hasta Máquinas de Turing Neuronales (NTM)[GWD14]. Todos estos tipos de Redes Neuronales se engloban dentro del término Deep Learning[LeC15].

Una de las más utilizadas son las redes neuronales recurrentes [MJ01] se diferencian de otras debido a que están completamente conectadas y son capaces de retroalimentarse entre todos los elementos de las que están compuestas. De esta manera una neurona está unida a las neuronas posteriores de la siguiente capa, a las neuronas de la capa anterior y a la misma neurona. Están conectadas a través de pesos de variables que son modificadas por cada salida con el fin de alcanzar los parámetros o metas de la operación. Teniendo en cuenta esta arquitectura de red neuronal, algunas de las características más importantes son:

- La complejidad computacional es más alta en comparación a un perceptrón multicapa.
- Estas redes por su arquitectura pueden propagar la información hacia delante en el tiempo, lo que las convierte en ideales a la hora de predecir eventos.
- Las RNN son especialmente útiles a la hora de resolver problemas de reconocimiento de patrones secuenciales y cambiantes en el tiempo como son las series temporales, por esta razón son ideales para resolver este problema.

Existe una evolución de estas redes que poseen más capacidad de memoria, estas son las: **Recurrent Neural Networks - LSTM (Long Short-Term memory) - Memoria a largo y corto plazo**. Estas redes identifican información relevante a corto y largo plazo y desechan la demás. Incorporan un elemento llamado célula de memoria que permite añadir o eliminar datos de la memoria y que actúa como una autopista a lo largo de toda la red neuronal. La información se modifica en la

célula de memoria a través de puertas que están compuestas por una red neuronal, una sigmoide, que es la válvula, ya que al moverse entre 1 y cero es equivalente a decir 1 totalmente abierto y o totalmente cerrada y un elemento multiplicador. Las puertas que controlan la celda de memoria son la puerta de olvido que determina qué información es eliminada de la célula de memoria, la puerta de entrada que decide qué valores se emplean para actualizar el estado de memoria y la puerta de salida que decide la salida de la neurona basándose en la entrada y el estado de memoria. Sus limitaciones se concentran en el uso inadecuado de la memoria, mantener de forma forzada información a lo largo de muchas observaciones puede llevarle a un rendimiento limitado.

Otra de las más populares en la actualidad son las Redes Neuronales Convolucionales (CNN)[LHL16] ya que ofrecen muy buen rendimiento en clasificación de audio y de imagen.

Las CNNs son muy similares a las redes neuronales comunes. Están formadas de neuronas que tienen biases y pesos sinápticos. Cada neurona recibe ciertas entradas, hace un producto matricial y aplica una función no lineal. De hecho, las últimas capas de una CNN son densamente conectadas, como lo son todas las de las Redes Neuronales comunes.

La gran diferencia entre las CNN y las Redes Neuronales comunes es que las CNN aprenden los filtros más adecuados para resolver cada problema en la parte de convolución. Existen otras dos grandes diferencias entre las CNN y las Redes Neuronales comunes que las hacen increíblemente más rápidas. La primera es que mientras estas últimas tienen las neuronas distribuidas bidimensionalmente, las neuronas de una CNN son tridimensionales. Y la segunda es que las neuronas de las capas de convolución dentro de una CNN no están densamente conectadas entre sí y los pesos de todas las neuronas dentro de una misma capa se actualizan simultáneamente.

9.1. Modelos para clasificación de imágenes

La clasificación de imágenes consiste agrupar imágenes en categorías semánticamente similares utilizando características visuales. En base a estas agrupaciones, se pueden percibir índices efectivos para una base de datos de imágenes. Para un aprendizaje supervisado de modelos de redes neuronales dedicados a la clasificación de imágenes esta base de datos se convierte en el *dataset* que posteriormente será dedicado al aprendizaje de los pesos.

Por norma general estos modelos tendrán las tres capas anteriormente citadas: capa de entrada, capa de salida y capa oculta.

La capa de entrada de estos modelos será cada píxel de una imagen con su correspondiente valor. El valor de cada píxel puede ser diferente en cada caso de clasificación, como por ejemplo para imágenes a color, RGB, o para imágenes en blanco y negro, escala de grises o binaria (negro o blanco). Por tanto, el número de perceptrones de la capa de entrada estará limitado a el número de píxeles por el valor del color.

La capa de salida se construirá con un vector del tamaño del número de clases. Esta capa es una capa de activación siendo la más sencilla y la encargada de dar una función a la entrada del perceptrón. Estas funciones pueden variar, pero para el caso de la clasificación de imágenes las más comunes son Softmax (Ver Eq. 9.1) para clasificar varias categorías y Sigmoid (Ver Fig. 9.1[1]) para saber si existe solo una categoría.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{9.1}$$

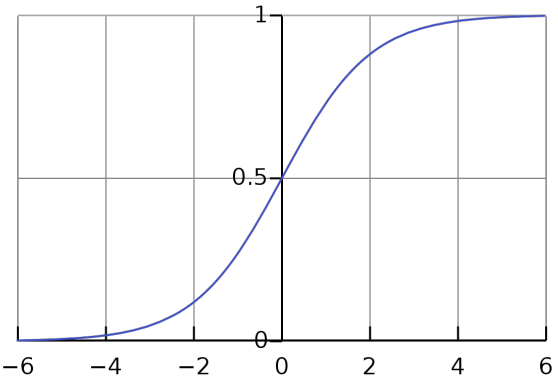

Figura 9.1: Función Sigmoide.

Los modelos habituales dedicados a la clasificación de imágenes son modelos pesados que están considerados dentro de técnicas de *deep learning*, por ello su capa oculta estará compuesta de multitud de capas y perceptrones.

Elementos fundamentales en las capas ocultas para la clasificación de imágenes son las capas de convolución. Estas capas se usan para cambiar los valores de entrada aplicando sus propios filtros que irán variando según el valor de los pesos. Una vez se ha realizado el filtro se realiza una reducción en la dimensión del vector (Ver Fig. 9.2[2]), aunque puede haber excepciones donde no se aplica esta reducción.

También existen otros tipos de capas habitualmente usadas en modelos de clasificación de imágenes. Estas capas no se ven influidas por los pesos que se ajustan durante el proceso de entrenamiento:

[1] https://en.wikipedia.org/wiki/Sigmoid_function#/media/File:Logistic-curve.svg
[2] https://miro.medium.com/max/875/0*QS1ArBEUJjjySXhE.png

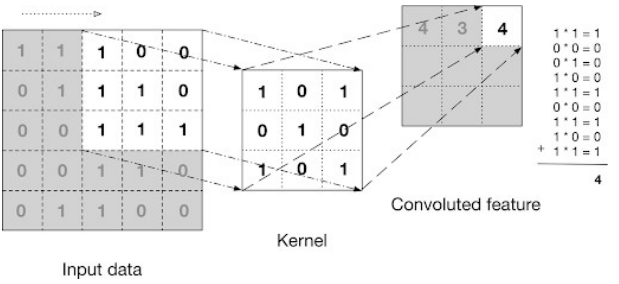

Figura 9.2: Imagen de una capa de convolución actuando sobre una imagen.

- **Capa de agrupación**: también llamada capa de *pooling*, se usa para reducir la dimensión de las imágenes. Consiste en dividir las imágenes en segmentos de un tamaño determinado y se aplica una función para transformar ese segmento a un solo valor. Esta función puede ser: el máximo valor (Ver Fig. 9.3[3]), el mínimo valor, la media de los valores o la suma de los valores de los segmentos.

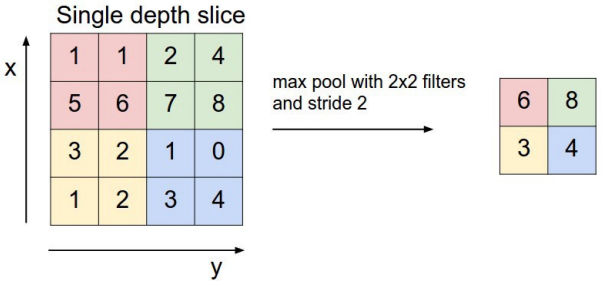

Figura 9.3: Imagen de una capa de agrupación con la función de máximos valores actuando.

- **Capa de normalización:** esta capa normaliza los valores del vector.
- **Capa de concatenación:** esta capa se usa para unir los valores de diferentes capas, por tanto su salida tendrá un tamaño igual a la suma de las entradas de la capa.

El uso de redes neuronales en clasificación de imágenes se ha extendido después de que en 2012 la red neuronal AlexNet sobrepasara los resultados obtenidos en el reto ImageNet (ILSVRC) [SRM+20] siendo anteriores ganadores y competidores algoritmos de procesamiento de imágenes sin uso de técnicas de *deep learning*. Desde esa rotunda victoria los siguientes años fueron dominados por multitud

[3] https://cdn-images-1.medium.com/max/1000/1*GksqN5XY8HPpIddm5wzm7A.jpeg

de tipos de redes neuronales. Además, ImageNet se convirtió en un reto referente en la investigación de nuevos modelos de redes neuronales capaces de mejorar la clasificación de imágenes. Desde 2013 realiza retos de segmentación semántica de objetos y desde 2015 de segmentación de objetos en vídeo.

Los resultados de algunos modelos de clasificación que recoge ImageNet se ven reflejados en la gráfica 9.4[4].

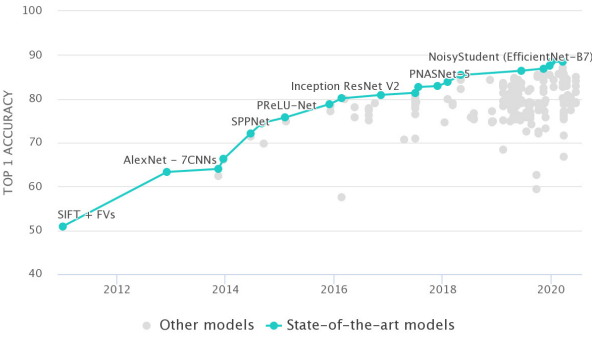

Figura 9.4: Ganadores de ImageNet según el año.

ResNet

ResNet, acrónimo de *Residual Neural Network*, está desarrollado por Microsoft y fue el ganador del reto ImageNet en 2015 con solo un error del 3,57 %. Este modelo tiene variantes con diferentes números de capas. El modelo que resultó ganador tenía 152 capas, aunque existen variantes con 50 y 101 capas más eficientes que consiguen buenos resultados.

La diferencia de este modelo respecto a un modelo plano de capas de convolución son los accesos directos de capas anteriores, *shortcut connections*. Como se ve en la figura 9.5[5] la red residual recibe señal tanto de su capa anterior como la que tuvo la capa predecesora a su predecesora y posteriormente pasada por una capa de activación. Su diseño final es más profundo y se basa en la construcción de modelos a partir de un bloque de capas llamado "Bottleneck" (Ver Fig. 9.6).

[4] https://paperswithcode.com/sota/image-classification-on-imagenet

[5] https://arxiv.org/pdf/1512.03385.pdf

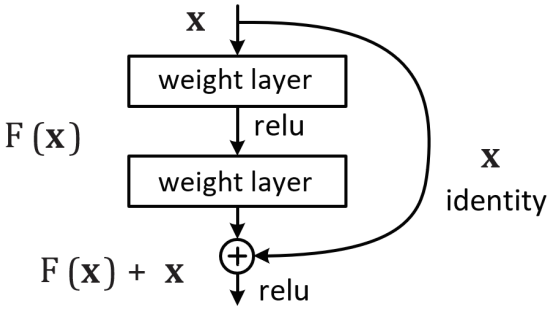

Figura 9.5: Imagen de un bloque residual.

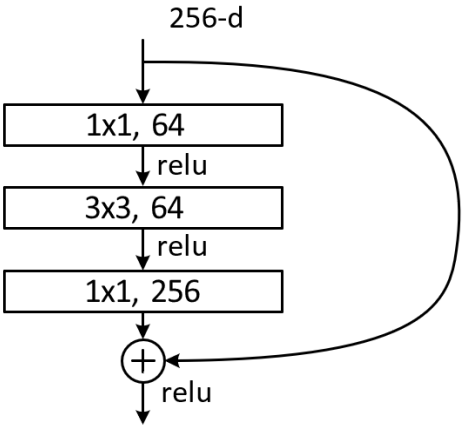

Figura 9.6: Imagen de un bloque Bottleneck.

Inception

InceptionNet está desarrollado por Google y constituye una serie de versiones de modelos (InceptionNetV3, InceptionNetV4...) [SIVA17]. Estos modelos se construyen a partir de un bloque de capas llamado "Inception" que consiste en realizar diferentes convoluciones variando el tamaño del segmento a filtrar y capas de *maxpooling* con la misma entrada en paralelo para, posteriormente, realizar una concatenación de todas las salidas (Ver Fig. 9.7[6]).

[6] https://arxiv.org/pdf/1602.07261.pdf

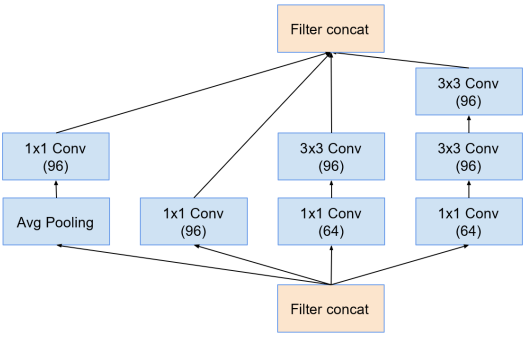

Figura 9.7: Imagen de un bloque Inception.

Inception-ResNet

Se constituye de una combinación de bloques Inception y Bottleneck (ResNet). Con su segunda versión llegó a ser ganador del reto ImageNet de 2017 con un error de tan solo 4,9 %.

EfficientNet

También investigadores de Google son los autores del modelo EfficientNet el cual es ganador de ImageNet 2019 y 2020 [TL19]. Además de conseguir una precisión mejor que ResNet e Inception-ResNet en ImageNet, es un modelo con menor cantidad de parámetros y de capas, por tanto, mucho más eficiente como su propio nombre indica. Actualmente es considerado el mejor modelo de clasificación de imágenes por sus resultados.

El modelo EfficientNetB0 se puede ver en la figura 9.8[7]. Los demás modelos se basan en la misma arquitectura, pero con diferentes valores de configuración en cada capa.

[7] https://arxiv.org/pdf/1905.11946.pdf

Stage i	Operator $\hat{F}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224× 224	32	1
2	MBConv1, k3x3	112× 112	16	1
3	MBConv6, k3x3	112× 112	24	2
4	MBConv6, k5x5	56× 56	40	2
5	MBConv6, k3x3	28× 28	80	3
6	MBConv6, k5x5	14× 14	112	3
7	MBConv6, k5x5	14× 14	192	4
8	MBConv6, k3x3	7× 7	320	1
9	Conv1x1 & Pooling & FC	7× 7	1280	1

Figura 9.8: Imagen del modelo EfficientNetB0.

9.2. Modelos para segmentación de imágenes

La segmentación de imágenes consiste en clasificar cada píxel de una imagen en una categoría. Se podría explicar también como la generación de una imagen con la profundidad que indica cada categoría. Para un aprendizaje supervisado de redes neuronales dedicadas a la segmentación de imágenes será necesario un conjunto de datos con imágenes originales e imágenes de máscaras binarias por cada categoría.

La capa de entrada de estos modelos será igual que en clasificación mientras que la capa de salida se construirá con un vector del tamaño del número de píxeles de la imagen original con la profundidad del número de clases.

Además de las capas explicadas en los modelos para clasificación de imágenes, para la segmentación se utilizan dos tipos de capas más:

- **Capa de deconvolución**: al igual que la capa de convolución esta capa realiza un filtro en los valores de entrada variando según el valor de los pesos, pero esta capa aumenta la dimensión del vector (Ver Fig. 9.9[8]), aunque, al igual que la convolución, puede haber excepciones donde no se aplica este aumento. Su uso reiterado en la estructura de una red se conoce como decodificación *transpose*.
- **Capa de *upsampling***: esta capa aumenta la dimensión de las imágenes. Hace una redimensión rellenando los píxeles con valores iguales (Ver Fig. 9.10[9]).

U-Net

U-Net es un modelo de segmentación de imágenes que fue creada para el reto de Detección de Caries en Radiografías de Mandíbulas en 2015 [KPIB19].

[8] https://miro.medium.com/max/679/1*AbCrAqPBfkqGRdhKtiZQqA.png

[9] https://www.oreilly.com/library/view/deep-learning-for/9781788295628/assets/a4df8c96-4e64-450f-b891-9efb18fc7368.png

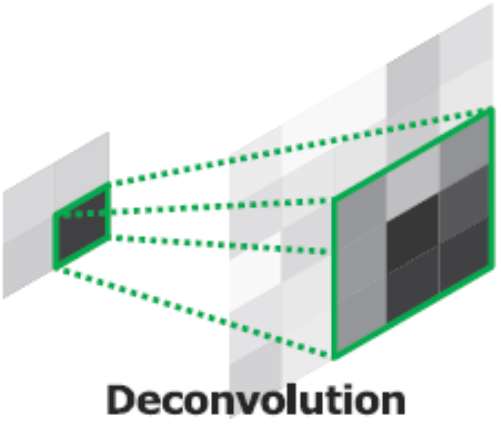

Figura 9.9: Imagen de una capa de deconvolución actuando.

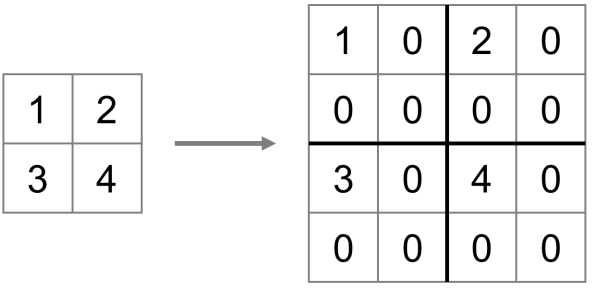

Figura 9.10: Imagen de una capa de *upsampling* actuando.

Su estructura se trata de un primer modelo de clasificación variante, después deconvoluciona de forma exponencial hasta llegar a la dimensión deseada. Además, concatena los valores que la red de clasificación calcula antes de reducir la dimensión junto con los valores de la capa de deconvolución previa para posteriormente deconvolucionar. La figura 9.11[10] muestra el modelo completo de una U-Net.

[10] https://arxiv.org/pdf/1505.04597.pdf

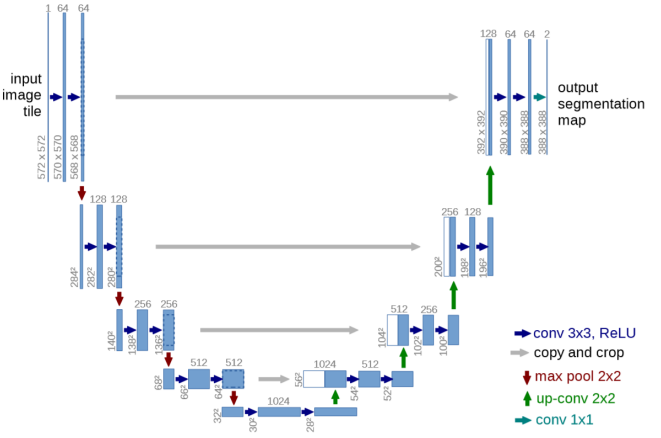

Figura 9.11: Imagen del modelo U-Net

9.3. Data Augmentation

Un problema común a la hora de abordar un caso de uso en el ámbito de *machine learning* es la falta de una gran cantidad de datos para llevar a cabo el proceso de entrenamiento/aprendizaje. *Data augmentation* sirve para ampliar la base de datos o más bien dar aleatoriedad a los datos en un entrenamiento supervisado.

Realmente es una herramienta de utilidad cuando no existen multitud de datos y nuestro caso de uso no es englobado por todos ellos. Aunque no se pretenderá crear nuevas categorías que abordar con solo *data augmentation*, éste también ayuda a controlar el sobreaprendizaje.

En el análisis de imágenes mediante técnicas de *deep learning* el *data augmentation* [PW17] es comúnmente usado, esto es debido a que, la edición de imágenes a partir de filtros automatizados y aleatorios son una gran herramienta para aumentar la cantidad de datos (Ver Fig. 9.12[11]).

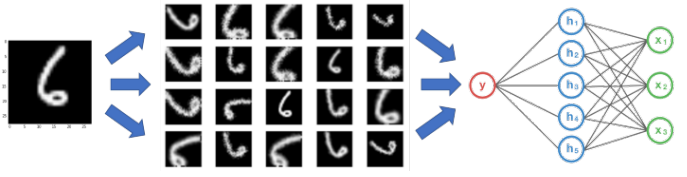

Figura 9.12: Imagen ejemplo de Data Augmentation.

[11] https://nanonets.com/blog/content/images/2018/11/1_dJNlEc7yf93K4pjRJL55PA--1-.png

Dentro de este proceso se utilizan herramientas como filtros para voltear imágenes, efectuar rotaciones o incluso el escalado de las mismas. Por otro lado, según el caso de uso concreto, se pueden llegar utilizar diferentes filtros como, por ejemplo, variaciones de brillo en un caso de uso que haya posibilidad de imágenes con diferente brillo, o un cambio de color en un caso de uso que los colores de cada categoría sean variantes y no sea una característica útil.

9.4. Plataformas de Deep Learning

Para implementar los distintos tipos de redes de Deep learning existen diferentes herramientas entre las que se pueden destacar las siguientes:

9.4.1. TensorFlow

TensorFlow [ABC+16] es una librería Open Source disponible para Python, C++, Java y GO. Se utiliza para computación numérica que usa grafos de flujo de datos. Cada nodo del grafo representa operaciones matemáticas, mientras que las aristas representan matrices multidimensionales de datos, también llamadas tensores, comunicadas entre ellas. Está desarrollada por Google y cuenta con una amplia documentación. Los cálculos computacionalmente pesados están implementados en C principalmente, esto la hace más eficiente que si estuvieran en Python. También tiene código optimizado para CUDA que es el lenguaje que utilizan las GPUs de NVDIA. Finalmente, destacar que cuenta con una herramienta visual muy potente llamada TensorBoard. Utilizando esta, se puede visualizar la evolución del entrenamiento de las distintas redes neuronales simplemente añadiendo unas pocas líneas de código además de analizar los distintos grafos que forman las distintas configuraciones.

TensorFlow tiene un conjunto de características que lo hacen jugar un rol muy importante en el área del Deep Learning:

- Portabilidad: implementa la computación en uno o más CPUs o GPUs en un escritorio, servidor o dispositivo móvil utilizando únicamente una API.
- Flexibilidad: Para diferentes dispositivos y sistemas como Android, iOS, Windows, Linux.
- Otras capacidades: Visualización, Puntos de parada, Auto-diferenciación, etc.

Una gran cantidad de proyectos de compañías muy importantes ya han utilizado TensorFlow. Compañías que utilizan TensorFlow como son (AMD, Nvidia, ebay, intel, Snapchat, Dropbox) en proyectos que han utilizado TensorFlow como: Neural Style Translation, Generative Handwriting, WaveNet.

9.4.2. Keras

Keras [Cho17] es una API de alto nivel enfocada a las Redes Neuronales. Está escrita en Python y se puede ejecutar sobre TensorFlow o Theano indistivamente. Fué desarrollada con la idea de crear prototipos rápidamente para diversos y pequeños experimentos. Lo que, a la vez que la hace fácil de aprender y experimentar, la resta de control sobre las mismas redes neuronales y resta algo de libertad a la hora de definir distintas arquitecturas. Esto la hace una estupenda herramienta para el campo de la investigación.

De esta manera, la API de Keras, nos permite expresar redes neuronales de una manera muy modular, considerando el modelo como si fuera una secuencia o un grafo. Estas estructuras pude ser debido a que un modelo de Deep Learning es un elemento discreto que se puede combinar de manera arbitraria.

Los nuevos componentes son fácilmente modificables y fáciles de agregar dentro del marco diseñado, para que los ingenieros prueben y exploren nuevas ideas. Además el lenguaje utilizado es el lenguaje Python, lenguaje muy útil y sencillo para el desarrollo de todo tipo de modelos.

Keras utiliza como backend TensorFlow, devolviendo así los resultados facilitados por dicha herramienta.

1. **Definir el modelo**: La estructura de datos principal utilizada en Keras es un modelo, el cual nos facilita la organización de las capas, ya que en Keras es básicamente una secuencia de capas. Hay capas que están totalmente conectadas o fully connected, max pool, capas de activación, etc. Se puede agregar fácilmente nuevas capas a nuestro modelo. Keras deducirá automáticamente la forma de todas las capas del modelo tras la primera capa. Esto significa que el ingeniero solo debe de establecer las dimensiones de la primera capa.

2. **Aprendizaje**: Una vez nuestro modelo haya sido diseñado y ya esté construido, pasamos al proceso de aprendizaje. Para compilar nuestro modelo debemos de especificar algunas propiedades adicionales que se requieren para entrenar a nuestra red, como puede ser la función de perdida o *loss*, evaluar el conjunto de pesos y el optimizador que recompilaremos durante el entrenamiento.

3. **Entrenamiento**: Una vez hemos definido y compilado nuestro modelo y ya esta listo para ser computado, es el momento de ejecutarlo con algunos datos, los datos que tengamos deben de tener las características de entrada y las de salida de nuestra red. De esta forma entrenaremos nuestro modelo para que nos de los resultados esperados más adelante. El proceso de formación se ejecutará un número bajo de iteraciones a través del conjunto devdatos llamados épocas o *epochs*. Definiremos el número de instancias que se evaluarán antes de quevse realice una actualización de pesos en la red mediante el argumento batch_size. Finalmente se podrá definir un atributo optativo denominado *verbose* para controlar la cantidad de información que queremos mostrar sobre la ejecución en la salida.

4. **Evaluar el modelo**: Una vez que nuestra red esté entrenada podemos evaluar la eficiencia del modelo modelo, mediante un conjunto de datos de test.

5. **Generar Predicciones** Una vez tenemos nuestro modelo seleccionado, aquel que nos ha aportado mejores resultados, pasaremos a realizar nuestras predicciones de los nuevos datos que vayan entrando.

Listado 9.1: Red Neuronal en Keras.

```
1
2  # Definir el Modelo
3
4  import tensorflow as tf
5  from tensorflow import keras
6
7  """default neural network"""
8
9  from tensorflow.keras import layers
10
11 inputs = keras.Input(shape=(256,), name='img')
12 x = layers.Dense(64, activation='relu')(inputs)
13 x = layers.Dense(64, activation='relu')(x)
14 outputs = layers.Dense(34, activation='relu')(x)
15
16 model = keras.Model(inputs=inputs, outputs=outputs, name='model')
17
18 model.summary()
19
20 keras.utils.plot_model(model, 'my_first_model_with_shape_info.png', show_shapes=True)
21
22 # Compile model
23 model.compile(loss= "mean_squared_error" , optimizer="adam", metrics=["mean_squared_error
       "])
24
25 # Fit Model
26 x = np.asarray(input).reshape(15, 256).astype('float32')
27 y = np.asarray(output).reshape(15,34).astype('float32')
28 model.fit(x, y, epochs = 10)
```

9.4.3. Theano

Theano apareció en 2007 en la universidad de Montreal y es una librería para Python que permite definir, optimizar y evaluar expresiones matemáticas. Está especialmente enfocado a cálculos con matrices multidimensionales. Combina aspectos de un sistema algebraico computacional con aspectos de un compilador optimizador Tiene una amplia documentación por su larga trayectoria de uso. Por esta misma razón también cuenta con una amplia comunidad, aunque bastante menos activa que la de Tensor- Flow. Cuenta con soporte nativo en Windows. Finalmente añadir que es el framework más rápido sobre una única GPU.

9.4.4. Caffe

Caffe[JSD+14] es un framework desarrollado por la universidad de Berkeley. También soporta CUDA como los demás frameworks vistos anteriormente. Mientras que, otros frameworks están más enfocados a la investigación, Caffe se decanta más por el lado del desarrollo de aplicaciones. Tampoco está orientado a nada fuera de la visión por computador, como NLP o análisis de audio y es ligeramente más lento que Theano y que TensorFlow. Lo que lo limita bastante. Con el fin de superar estas limitaciones existe Caffe2. Es una versión avanzada que se enfoca en el mismo sentido que Caffe, es decir, aplicaciones móviles y escalabilidad, pero mejora con respecto a Caffe aún más el despliegue en aplicaciones móviles, mejor soporte de Hardware y más flexibilidad para mejoras futuras. A pesar de todo, su mayor cambio respecto a Caffe son los Operadores. Estos son versiones más flexibles de las capas de Caffe.

9.4.5. Pytorch

PyTorch [JG20] es una librería de Python de código abierto implementada por Facebook. Esta librería tiene varios módulos interesantes para el desarrollo de redes neuronales. Gracias al uso de estos módulos, PyTorch, a diferencia de otras librerías, calcula los gradientes de los pesos de las distintas capas a la vez que aplica la fase forward a los datos de entrenamiento durante el entrenamiento de la red.

Esto supone una gran ventaja frente a otras librerías como Keras que necesitan calcular de manera estática los gradientes para aplicar posteriormente el algoritmo. Otra de las ventajas de esta librería es la posibilidad de crear tus propios módulos personalizados, lo cual permite al programador tener una mayor libertad a la hora de diseñar las redes neuronales, permitiendo la creación de éstas de forma más específica para resolver problemas concretos

Listado 9.2: Red Neuronal en pytorch.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.autograd import Variable
print(torch.__version__)

use_cuda = True

# transformar las imágenes en vectores/tensores para su manejo
transform = transforms.Compose(
    [transforms.ToTensor()])

# descargar tanto el training como el test datasets en ./data del colab
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
testset = torchvision.datasets.MNIST(root='./data', train=False,
```

```
                                        download=True, transform=transform)

# cargar y organizar en batches para facilitar el trabajo en paralelo
batch_size = 32 # se puede probar con 32 o 64

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

# Definiendo nuestra red Neuronal

class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        # 1st Full-Connected Layer: 784 (input data) ->500 (hidden node)
        self.fc1 = nn.Linear(input_size, hidden_size)
        # Non-Linear ReLU Layer: max(0,x)
        self.relu = nn.ReLU()
        # 2nd Full-Connected Layer: 500 (hidden node) ->10 (output class)
        self.fc2 = nn.Linear(hidden_size, num_classes)

    # Forward: apilar/organizar ls capas
    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out


input_size    = 784   # tamaño de la imagen = 28 x 28 = 784
hidden_size   = 500   # número de los nodos de la capa oculta
num_classes   = 10    # número de clases
net = Net(input_size, hidden_size, num_classes)

# Habilitamos el uso de Cuda para esta red

torch.cuda.device_count()

device = torch.device("cuda:0")
net.to(device)
net = nn.DataParallel(net)
net

# Entrenando nuestra Red Neuronal

# función de pérdida
criterion = nn.CrossEntropyLoss()

# optimizador
learning_rate = 1e-3
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)

num_epochs = 5
for epoch in range(num_epochs):
    train_running_loss = 0.0
    train_acc = 0.0
    net = net.train()
```

```
77      # Cargar batch de imágenes
78      for i, (images, labels) in enumerate(trainloader):   #
79          # De torch tensor a Variable: de vector(784) a matriz of 28 x 28
80          images = Variable(images.view(-1, 28*28))
81          labels = Variable(labels)
82
83          if use_cuda and torch.cuda.is_available():
84              images = images.cuda()
85              labels = labels.cuda()
86
87          # Inicializar los pesos ocultos a cero
88          optimizer.zero_grad()
89          # Forward: calcular la salida a partir de cierta imagen
90          outputs = net(images)
91          # Calcular la pérdida: diferencia entre la salida y la etiqueta prevista
92          loss = criterion(outputs, labels)
93          # Backward: calcular los pesos
94          loss.backward()
95          # Optimizador: actualizar los pesos de los nodos ocultos
96          optimizer.step()
97
98          train_running_loss += loss.detach().item()
99          corrects = (torch.max(outputs, 1)[1].view(labels.size()).data == labels.data).sum
                ()
100         train_acc += 100.0 * corrects/batch_size
101
102     net.eval()
103     print('Epoch: %d | Loss: %.4f | Train Accuracy: %.2f' \
104 #
105
106 correct = 0
107 total = 0
108 for images, labels in testloader:
109     images = Variable(images.view(-1, 28*28))
110
111     if use_cuda and torch.cuda.is_available():
112         images = images.cuda()
113         labels = labels.cuda()
114
115
116     outputs = net(images)
117     _, predicted = torch.max(outputs.data, 1)  # Choose the best class from the output:
            The class with the best score
118     total += labels.size(0)                    # Increment the total count
119     correct += (predicted == labels).sum()     # Increment the correct count
120
121 print('Exactitud de la red en 10k imagenes: %d %% % (100 * correct / total))
122
123 # exportacion del modelo
124 torch.save(net.state_dict(), 'fnn_model.pkl')
```

Bibliografía

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[Cho17] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2017.

[DBDJH14] Howard B Demuth, Mark H Beale, Orlando De Jess, and Martin T Hagan. *Neural network design*. Martin Hagan, 2014.

[GWD14] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[JG20] Shruti Jadon and Ankush Garg. *Hands-On One-shot Learning with Python: Learn to implement fast and accurate deep learning models with fewer training samples using PyTorch*. Packt Publishing Ltd, 2020.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[KPIB19] Thorbjørn Louring Koch, Mathis Perslev, Christian Igel, and Sami Sebastian Brandt. Accurate segmentation of dental panoramic radiographs with u-nets. In *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*, pages 15–19. IEEE, 2019.

[LeC15] Yann LeCun. Deep learning & convolutional networks. In *Hot Chips Symposium*, pages 1–95, 2015.

[LHL16] Yandong Li, ZB Hao, and Hang Lei. Survey of convolutional neural network. *Journal of Computer Applications*, 36(9):2508–2515, 2016.

[MJ01] Larry R Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 5:64–67, 2001.

[PW17] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[SIVA17] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[SRM+20] Vaishaal Shankar, Rebecca Roelofs, Horia Mania, Alex Fang, Benjamin Recht, and Ludwig Schmidt. Evaluating machine accuracy on imagenet. In *International Conference on Machine Learning*, pages 8634–8644. PMLR, 2020.

[TL19] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.