UNIVERSIDAD
COMPLUTENSE
MADRID

Desarrollo de Aplicaciones y Sistemas Inteligentes (DASI)

Redes neuronales artificiales

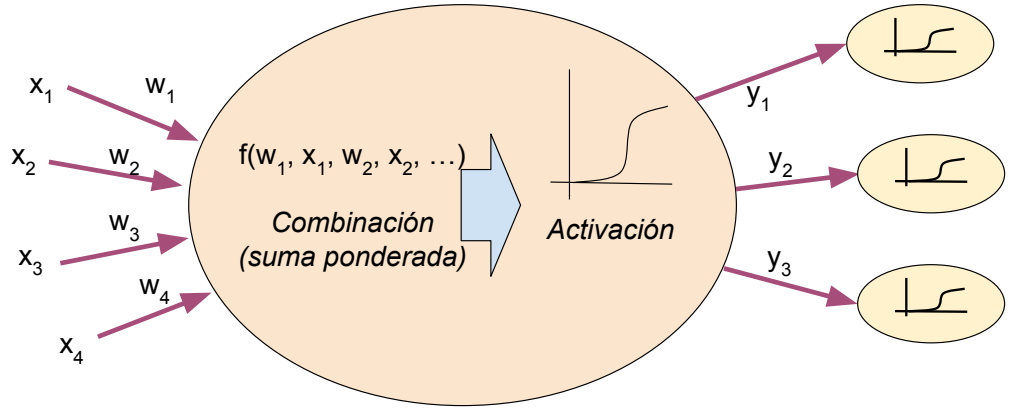
Juan Pavón Mestras
Dep. Ingeniería del Software e Inteligencia Artificial UCM

Redes Neuronales Artificiales

- Las Redes Neuronales Artificiales son un enfoque conexionista de la Inteligencia Artificial
  - Vagamente inspirado en sus homólogos biológicos
- Una RNA es una estructura de **grafo dirigido**:
  - Un conjunto de **nodos** llamados neuronas artificiales
  - **Conectados** entre sí para transmitir señales por medio de enlaces
  - La salida de la neurona la determina una **función de activación**
    - Que modifica el valor resultado
    - O impone un límite que se debe sobrepasar para propagar el valor a otra neurona
  - Los enlaces multiplican el valor de salida de la neurona anterior por un valor de **peso**
  - Estos pesos pueden incrementar o inhibir el estado de activación de las neuronas
  - La información de **entrada** atraviesa la ANN (donde se somete a diversas operaciones) produciendo unos valores de salida

Neurona artificial

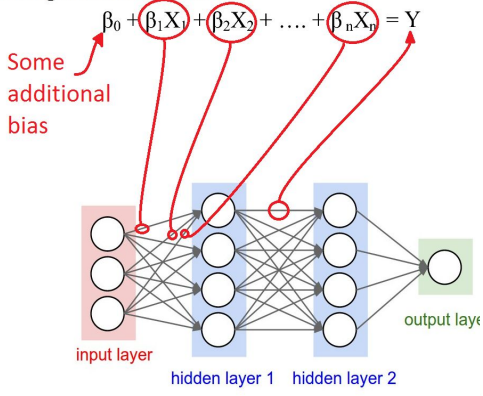
- Una neurona artificial es un modelo muy simplificado de una neurona biológica
  - Tiene unas **entradas**, cuyo valor está ponderado por unos pesos w_i así como un valor de sesgo (bias) w_0
  - Aplica una **función de activación** a la suma de estas entradas
  - El **resultado** se transmite a otras neuronas

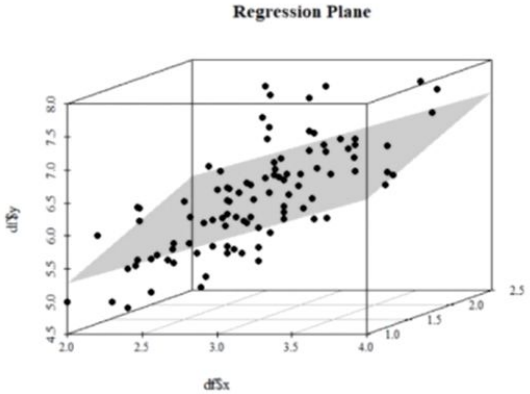


Regresión lineal en las neuronas artificiales

Multiple Regression Formula :

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n = Y$$



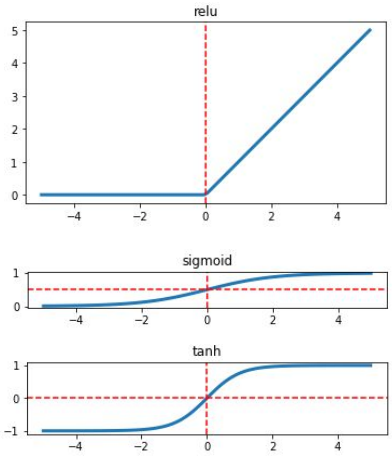


$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

Función de activación

- **ReLU**: y = max(0, x)
  - La más habitual para capas ocultas en redes profundas
  - Su derivada es simple: cuando x <0, dy / dx = 0; y cuando x> = 0, dy / dx = 1
- **sigmoid**: $y = 1/(1 + e^{-x})$
  - La más utilizada para la capa de salida
  - Su derivada es simple y' = 1/(1 - y)
  - Y punto de inflexión en x=0
- **tanh**: $y = 2 / (1 + e^{-2x}) - 1$

  ⇒ En la capa de salida, sigmoide suele ser mejor que tanh, porque la probabilidad de que la salida esté más cerca de 0 o 1 es mayor (véase por ejemplo para la clasificación binaria)



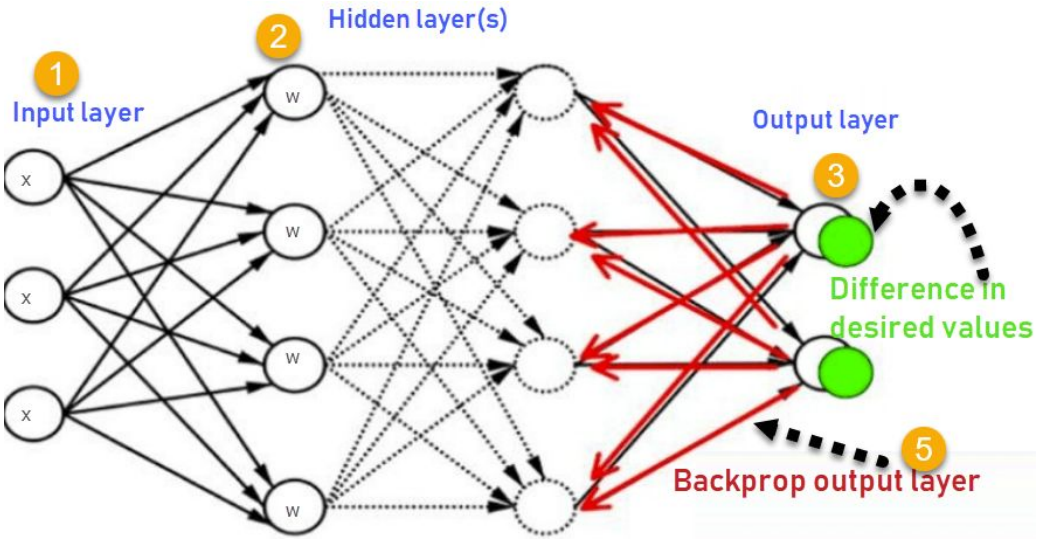
- Otras funciones: **Softmax**, **Linear**, personalizada

  ⇒ https://keras.io/api/layers/activations/

Backpropagation

- Propagación hacia atrás de errores o retropropagación

  David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams. *Learning representations by back-propagating errors*. Nature volume 323, pages533–536(1986)



Backpropagation

- Para obtener los pesos óptimos del modelo, de manera similar a como se hace con la regresión logística se trata de minimizar la función de coste $J(W)$ (el error cuadrático medio)
- Los gradientes ΔW se calculan usando ***backpropagation***
  - Primero se hace una pasada hacia delante
  - Viendo el resultado final se van calculando los gradientes de los pesos hacia atrás
    - Desde la capa de salida hasta la capa de entrada
    - Los cálculos parciales de los gradientes de cada capa se utilizan para calcular los de la capa anterior ⇒ La información del error se va propagando capa a capa en vez de calcularla por separado
- Este proceso se repite hasta que la función de coste converge o un número máximo de iteraciones

RNA de una capa

- Veamos el caso de una RNA de una sola capa:
  - Una capa de entrada (*input layer*): características de entrada (x)
  - Una capa oculta (*hidden layer*)
  - Una capa de salida (*output layer*): variables objetivo (y)
    - En un clasificador: un nodo si es binario que indica la probabilidad de la clase positiva y n nodos para clasificador multiclase indicando la probabilidad de cada una
    - En regresión tendrá un nodo cuyo valor es la predicción

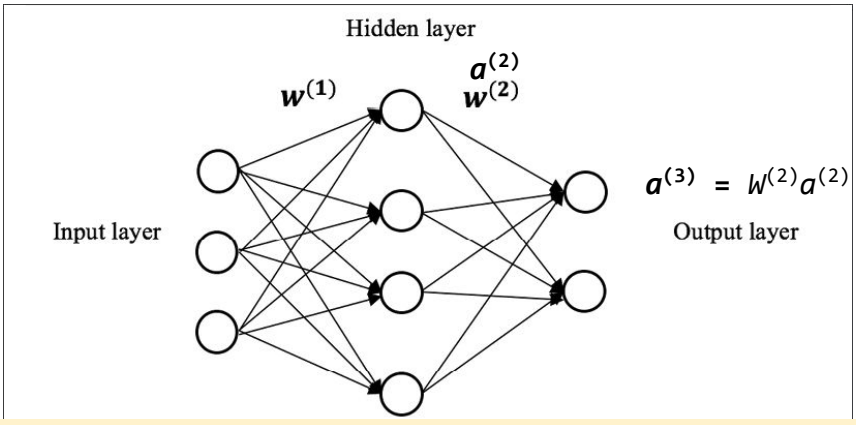


Figure 8.1: A simple shallow neural network, del libro Yuxi (Hayden) Liu: *Python Machine Learning By Example, Third Edition*. Packt Publishing, 2020

Backpropagation para una RNA de una capa

- En este caso se trata de encontrar los pesos `W = {W(1), W(2)}`
  1. Usando los pesos existentes se hace una pasada hacia delante y se calculan los valores de salida de la capa oculta, $\boldsymbol{a}^{(2)}$, y los de la capa de salida, $\boldsymbol{a}^{(3)}$
  2. Se calcula la derivada de la función de coste para la capa de salida respecto a su entrada $\delta^{(3)} = \frac{\partial}{\partial z^{(3)}} J(W) = -(y - a^{(3)}) \cdot f'(z^{(3)}) = a^{(3)} - y$
  3. Se calcula la derivada de la función de coste para la capa oculta respecto a su entrada $\delta^{(2)} = \frac{\partial}{\partial z^{(2)}} J(W) = \frac{\partial z^{(3)}}{\partial z^{(2)}} \frac{\partial}{\partial z^{(3)}} J(W) = ((W^{(2)})\delta^{(3)}) \cdot f'(z^{(2)})$
  4. Se calculan los gradientes usando la regla de la cadena:

     $\Delta W^{(2)} = \frac{\partial J(W)}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial W^{(2)}} = \delta^{(3)} a^{(2)}$ $\quad\quad$ $\Delta W^{(1)} = \frac{\partial J(W)}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(1)}} = \delta^{(2)} x$
  5. Y se actualizan los pesos con estos gradientes con una tasa de aprendizaje α y siendo m el número de ejemplos:

     $W^{(1)} := W^{(1)} - \frac{1}{m} \alpha \Delta W^{(1)}$ $\quad\quad$ $W^{(2)} := W^{(2)} - \frac{1}{m} \alpha \Delta W^{(2)}$

Ejemplo de implementación

- Código:

  https://colab.research.google.com/drive/1s0DzrDjqlfKelsmqvlpsElJPpXi5e6Cj?usp=sharing

Revisión de las funciones de activación

- **Lineal**: f(z) = z.
  - Es como que no hubiera función de activación
  - Se suele utilizar en la capa de **salida en redes de regresión**, ya que no necesitamos ninguna transformación en las salidas
- **Sigmoide (logística)**
  - Se puede interpretar como la probabilidad de una predicción de salida
  - Se suele utilizar en la capa de **salida** en las redes de **clasificación binarias**
- **Softmax**
  - Es una función logística generalizada que se utiliza en la salida para la **clasificación multiclase**

Revisión de las funciones de activación

- **Tanh**
  - Versión mejorada de la función sigmoidea con gradientes más fuertes, las derivadas de la función tanh son más pronunciadas que las de la función sigmoidea. Tiene un rango de -1 a 1
  - Es habitual utilizar la función tanh en las **capas ocultas**
- **ReLU**
  - La más utilizada "por defecto" en las capas ocultas de las redes feedforward
  - Su rango va de 0 a infinito, y tanto la propia función como su derivada son monótonas
  - Un inconveniente de la función ReLU es la incapacidad de asignar adecuadamente la parte negativa de la entrada, ya que todas las entradas negativas se transforman en cero
- **Leaky ReLU**
  - Soluciona el problema anterior introduciendo una pequeña pendiente en la parte negativa. Cuando z < 0, f(z) = az, donde a suele ser un valor pequeño, como 0,01

DASI

TensorFlow

TensorFlow

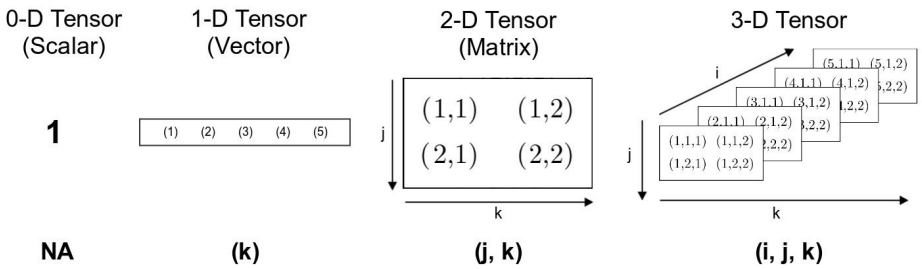
- Google Machine Learning Tools (2ª generación)
- Biblioteca de métodos de ML para construir aplicaciones inteligentes
  - Deep learning y otros algoritmos de ML (clasificación, regresión, SVM)
  - Otras: Theano, Torch, CNTK (Microsoft), MXNet (Apache)
- Sistema de programación que representa las computaciones como grafos de flujos de datos, y que se puede ejecutar en sistemas de computación distribuida y paralela
- Se puede probar (sin necesidad de programar): https://playground.tensorflow.org/

Tensores

- Las computaciones numéricas se expresan como grafos:
  - Nodos como **operaciones** con cualquier número de entradas y salidas
  - Arcos como **tensores** que fluyen entre los nodos
- Cada nodo es una implementación de una capa de neuronas artificiales que tienen como entrada tensores (matrices) y producen otros tensores de la misma u otras dimensiones
- Los tensores son similares a los arrays de NumPy, pero
  - Un tensor es inmutable (no se modifican, se crean nuevos)
  - Los tensores están optimizados para procesarlos en GPUs y TPUs (*Tensor processing units*), con mayor eficiencia que los arrays NumPy
  - También se pueden usar arrays NumPy como entrada y salida
- Los tensores no son exclusivos de TensorFlow
  - Son un recurso matemático para redes de Deep Learning
  - Otras librerías, como Pytorch, también usan el mismo concepto

Tensores

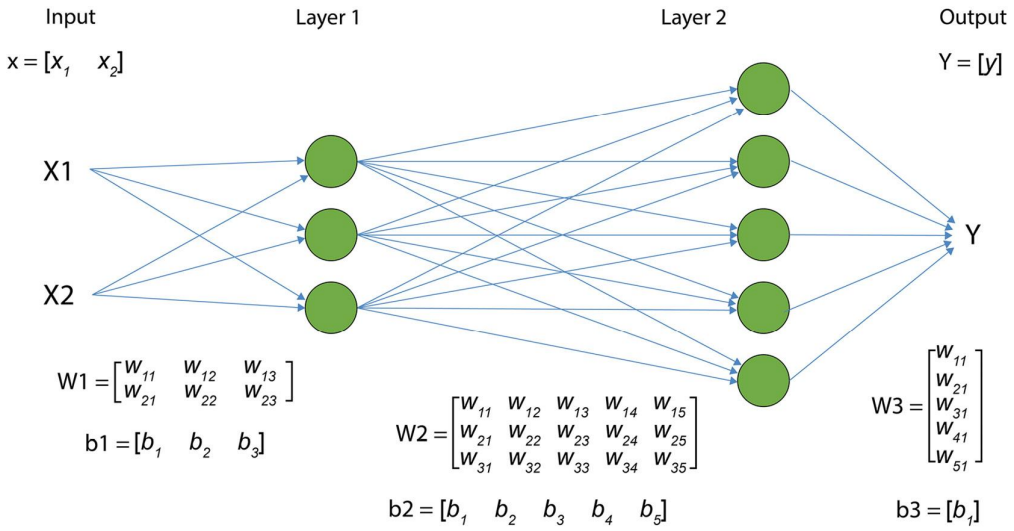
- Un tensor está caracterizado por:
  - Rank: número de dimensiones



  - Shape: forma en la que se organizan los datos
  - Type: tipo de los datos (numéricos)
- Operaciones
  - Suma (y resta): `C = A + B`, siendo `cij = aij + bij`
  - Multiplicación: se multiplica cada fila de A por cada columna de B
  - Reshaping: cambio de forma o de dimensiones
  - Transposición (T): (nxm)T → (mxn)

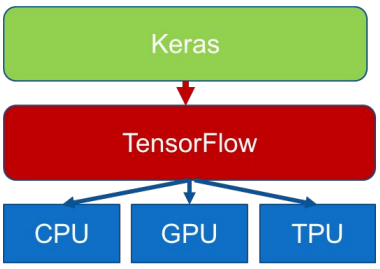
Capas y tensores

- Como entrada un tensor bidimensional y como salida un tensor unidimensional (un escalar)
- Para cada capa hay que determinar sus pesos y sesgos



Keras API

- Para facilitar el desarrollo de modelos de deep learning con TensorFlow hay varias APIs, **Keras es la API oficial**
  - Pero también se puede utilizar sobre otras herramientas de aprendizaje máquina, como Theano, CNTK, MXNet
  - Y TensorFlow puede ejecutar sobre distintos tipos de plataforma computacional (CPU, GPU, TPU)



- Ofrece funcionalidad para
  - Preprocesar datos
  - Crear modelos como un grafo de capas
  - Entrenamiento
  - Validación y optimización de modelos

Instalación

- Asumimos que tenemos instalado Anaconda
- Como suele haber cambios habituales en las versiones de Keras y sobre todo de TensorFlow, con distintas dependencias de librerías, es conveniente instalarlo en un entorno específico de Anaconda
  - Con conda:

    Usa el nombre que mejor te parezca

    ```
    $ conda create --name curso --clone root
    ```
- Una vez creado este entorno, activarlo para trabajar ahí:

  ```
  $ conda activate curso
  ```
- Y posteriormente instalar la última versión de tensorflow, que también instalará keras:

  ```
  $ pip install --upgrade tensorflow
  ```

  ⇒ Si dispones de una computadora con GPU, instala **tensorflow-gpu**
- Si se quisiera eliminar el entorno con todos sus ficheros:

  ```
  $ conda deactivate    # si el entorno activo fuera curso
  $ conda remove -n curso --all
  ```

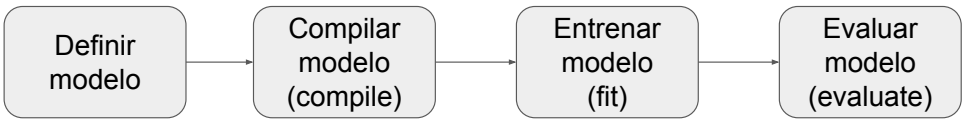

Modelos

- El módulo **keras.models** implementa varios modelos de RNA:
  - **Sequential**: Una pila lineal de capas (esto es, una tras otra)
    - Cada capa tiene exactamente un tensor de entrada y uno de salida
    - Tiene el método **add()** para añadir una capa encima de las demás
    - Y el método **pop()** para eliminar la última capa
  - **Functional API**: conjunto de métodos para construir modelos con cualquier topología (como un grafo dirigido acíclico) https://keras.io/guides/functional_api/
    - A cada capa se le puede indicar cual tiene de entrada
    - Se pueden unir varias con **concatenate()**
- Métodos más usados para trabajar con modelos:
  - **summary()** - permite ver una descripción textual de la red
  - **compile()** - configura el modelo para ser entrenado
  - **fit()** - entrena el modelo por un número de iteraciones (épocas)
  - **evaluate()** - devuelve valores de pérdida y métricas del modelo en modo prueba
  - **predict()** - genera predicciones a partir de ejemplos de entrada

Capas

- Son conjuntos de nodos que procesan los tensores
- Tipos de capas más comunes:
  - **Dense**: Todos los nodos de la capa están conectados a todos los nodos de las capas anterior y posterior
    - Muy usados en tareas de clasificación y regresión con datos tabulares
  - **Convolutional**: Implementa convoluciones en una o varias dimensiones
    - Habituales en clasificación de imágenes
  - **Pooling**: Se utiliza para reducir las dimensiones del tensor producido por la capa anterior
    - Hay varios tipos de pooling: max pooling (la salida es el valor máximo de una ventana dada), average pooling (la salida es la media)
    - Se utilizan conjuntamente con una capa convolucional para reducir la dimensión en las siguientes capas
  - **Recurrent**: Permite extraer patrones de secuencias de datos temporales (esto es, la salida depende de los resultados del paso anterior)
    - Útiles para proceso de lenguaje natural o series de datos temporales
  - **Dropout**: excluye aleatoriamente algunas de las neuronas en cada iteración del proceso de aprendizaje, para evitar el efecto de adaptación conjunta

Pasos para crear y entrenar una red con Keras



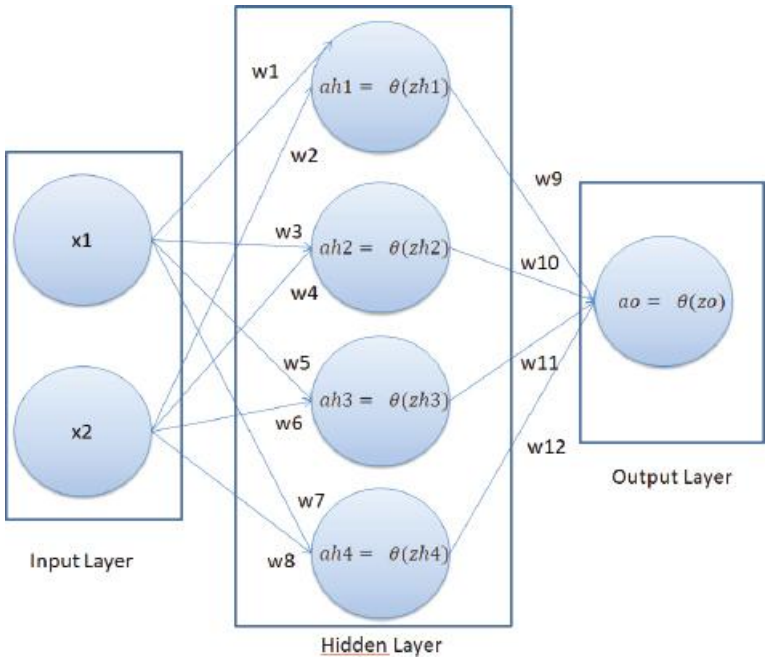
DASI

Redes Neuronales Densamente Conectadas (Perceptrón Multicapa)

Redes neuronales densamente conectadas

- Densely Connected Neural network (DNN) o **Perceptrón Multicapa**
  - Todos los nodos de una capa están conectados a todos los nodos de la siguiente capa
- Constan de
  - Una capa de entrada (*input layer*): las entradas (*features*)
  - Una o más capas ocultas (*hidden layers*)
  - Una capa de salida (*output layer*): la respuesta
- El entrenamiento se encarga de determinar los pesos que permiten un error mínimo entre la salida producida y la real
- Se aplican normalmente para hacer predicciones con datos tabulares

Ejemplo de red neuronal



Caso de estudio - Preparación de los datos

- Se va a desarrollar un problema de clasificación binaria
  - Se extraen las variables numéricas y las categóricas
  - Se convierten a numéricas las categóricas con el método **get_dummies()**
    - Este método crea una columna por cada valor posible de cada categoría y asigna el valor 0 o 1 si la fila tiene o no ese valor
    - Para las categorías con solo dos valores, bastaría con una (0 o 1), y esto se consigue con `drop_first=True`
  - Finalmente se concatenan el dataframe resultante con las numéricas

```
numericas = X.drop(['sex', 'smoker', 'day', 'time'], axis = 1)
categoricas = X.filter(['sex', 'smoker', 'day', 'time'])
cat_numericas = pd.get_dummies(categoricas, drop_first=True)
X = pd.concat([numericas, cat_numericas], axis = 1)
X.head()
```

con X.info() se puede ver cuáles son o no numéricas

	total_bill	size	sex_Female	smoker_No	day_Fri	day_Sat	day_Sun	time_Dinner
0	16.99	2	1	1	0	0	1	1
1	10.34	3	0	1	0	0	1	1
2	21.01	3	0	1	0	0	1	1
3	23.68	2	0	1	0	0	1	1
4	24.59	4	1	1	0	0	1	1

Caso de estudio - Preparación de los datos

- Generación del conjunto de entrenamiento y de pruebas a partir del conjunto inicial

```
# Generar conjunto de entrenamiento y de pruebas
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
    random_state=0)

# Escalar los datos
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

20% de los datos para test y el resto para entrenamiento

Calcula media y desviación estandar (**fit**) y los guarda en el objeto StandardScaler
Luego los usa para hacer **transform**()

Aquí basta con transform() porque ya se hizo fit() en sc

Funcionamiento de la red neuronal artificial

- Normalmente funcionan en dos pasos
  - Feed Forward: Propaga los resultados de la ejecución de las neuronas desde la capa de entrada hasta la de salida
    - Para cada neurona artificial, la salida es una función (de *activación*, se detallará más adelante) sobre la suma ponderada de sus entradas y el sesgo

      $zh1 = b + w1*x1 + w2*x2$

      $ah1 = 1 / (1 + e^{-zh1})$

      ... similar para ah2, ah3 y ah4

      $z_o = w9*ah1 + w10*ah2 + w11*ah3 + w12*ah4$

      $a_o = 1 / (1 + e^{-Zo})$
  - Backpropagation: trata de minimizar la pérdida global encontrando mejorando los valores de los pesos w_i y del sesgo aplicado
    - Como función de pérdida (loss function) se puede usar la del error cuadrático medio del valor predicho por la red respecto al valor real
    - Primero se calculará la derivada de la función de coste respecto a los pesos de salida (los que se aplican en las entradas de las neuronas de la capa de salida) y respecto al sesgo (b)
    - Luego se calculará la derivada del coste respecto a los pesos y sesgo para las capas ocultas

Creación de la red neuronal

- Crear un objeto de la clase Sequential del módulo **`tensorflow.keras.models`**
- Añadir capas con el método **`add(capa)`**
- Para crear una capa de red densa, usar la clase **`Dense`** con los siguientes parámetros
  - Número de nodos de la capa
  - Dimensión de la entrada (si es la primera capa densa, las siguientes capas densas pueden calcularlo por los nodos de la capa anterior)
  - Función de activación
- Compilar el modelo con el método **`compile()`** indicando
  - Función de pérdida (*loss function*)
  - Optimizador
  - Métricas

Creación del modelo de red neuronal artificial

```
def create_model (learning_rate , dropout_rate ):
    # crea un modelo Secuencial
    model = Sequential()

    # añade las capas
    model.add(Dense(12, input_dim=X_train.shape[1 ], activation="relu"))
    model.add(Dropout(dropout_rate))
    model.add(Dense(6, activation="relu"))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, activation="sigmoid"))

    # compila el modelo
    adam = Adam(lr=learning_rate)
    model.compile (loss="binary_crossentropy", optimizer=adam,
                   metrics=["accuracy"])

    return model
```

Sequential porque es una secuencia de capas, cada una con un tensor de entrada y uno de salida

En la primera se indica la dimensión, que será la de los datos de entrenamiento
Para las demás no hace falta especificarlo, será la misma

Función de activación relu para las capas de entrada y ocultas

En la última capa sigmoid

Capa Dropout

- Después de cada capa Dense se suele poner una Dropout
- El propósito es aumentar la convergencia de las redes neuronales, para que el aprendizaje sea más rápido
  - Al entrenar la red neuronal, resulta difícil valorar el impacto de cada una de las características de entrada a cada neurona, que son numerosas
  - Esto hace que los errores de algunas neuronas se suavizan con los valores correctos de otras y los errores se acumulan en la salida de la red neuronal
    ⇒ El entrenamiento se detiene en un cierto mínimo local con un error considerable (efecto de adaptación conjunta de características)
- La capa Dropout excluye aleatoriamente algunas de las neuronas en cada iteración del proceso de aprendizaje
  - La disminución en el número de características durante el entrenamiento aumenta la importancia de cada una, mientras que el cambio constante en la composición cuantitativa y cualitativa de las características reduce el riesgo de su adaptación conjunta

Compilación del modelo

- El método compile tiene varios parámetros que dependerán del tipo de problema de clasificación o regresión
  - La **función de pérdida** (loss function), que sirve para encontrar desviaciones en el proceso de aprendizaje
    - Para clasificación binaria, como es el caso del ejemplo: **binary_crossentropy**
    - Otras: **mean_squared_error, mean_absolute_error, hinge, huber_loss, poisson**, ...
    - Definidas en https://keras.io/api/losses/
  - El **optimizador**, método implementado para optimizar los pesos de entrada comparando la predicción y la función de pérdida
    - Adam: implementa el método de descenso del gradiente
    - Otros: SGD (optimizador de descenso de gradiente estocástico), Adagrad, Adadelta, Adamax, Nadam
  - **Métricas**, para juzgar el rendimiento del modelo, se pasa como un diccionario o una lista
    - Definidas en https://keras.io/api/metrics/
    - Ejemplos: accuracy (proporción de predicciones correctas), precision (proporción de etiquetas predichas correctamente), recall (proporción de positivos reales identificados correctamente), etc.

Métricas de clasificación

Para determinarlas se tienen en cuenta los resultados de un modelo:

VP: Verdaderos positivos
VN: Verdaderos negativos
FP: Falsos positivos
FN: Falsos negativos

se definen:

- **Exactitud** (*accuracy*) - Proporción de predicciones correctas sobre predicciones totales

  `(VP + VN) / (VN + VP + FN+ FP)`

- **Precisión** (*precision*) - Proporción de predicciones verdaderas correctas sobre todas las que se han identificado como verdaderas

  `VP / (VP + FP)`

- **Exhaustividad** (*recall*) - Proporción de predicciones positivas encontradas de todos los casos positivos posibles (esto es, tiene en cuenta los casos positivos no identificados)

  `VP / (VP + FN)`

⇒ Un modelo puede tener mucha exactitud pero pasar por alto muchos casos (poco preciso o poco exhaustivo)

Métricas de clasificación

- **F1 score** - Determina cuán robusto es el modelo, combinando precisión y exhaustividad

  `F1score = 2 / (1/precision + 1/recal)`

- En general habrá que tener en cuenta varias métricas, siendo unas más relevantes que otras según el problema que se esté abordando
  - Se querrá un modelo con precisión alta cuando se quiere asegurar que las predicciones etiquetadas como positivas son bastante acertadas
  - Se querrá un modelo con recall alto cuando se quieren detectar el máximo de casos positivos, aún incurriendo en muchos falsos positivos
  - Ejemplo: Diagnóstico médico
    - Si se tiene una enfermedad muy contagiosa es deseable tener un recall alto
    - Si el tratamiento requerido para una enfermedad es de alto riesgo, se querrá una gran precisión
    - Si se trata de una enfermedad rara y el modelo acierta cuando indica que un caso es positivo, pero pasa por alto muchos otros, será de poca utilidad

Configuración del modelo

- Habrá que parametrizar el modelo con la siguiente información:
  - Tasa de eliminación (*dropout rate*), para las capas de Dropout
    - Suele ser normal 0,1
  - Tasa de aprendizaje (*learn rate*)
- Con estos parámetros se puede invocar la función de creación del modelo

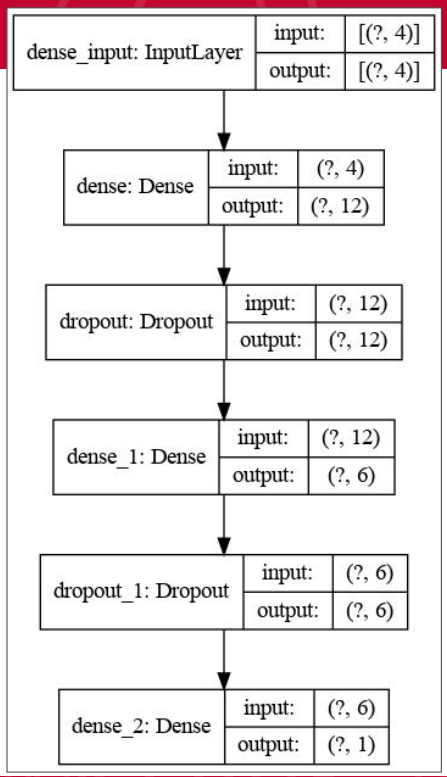
```
learn_rate = 0.001
dropout_rate = 0.1

model = create_model(learn_rate, dropout_rate)
```


Visualización del modelo

```
from tensorflow.keras.utils import plot_model

plot_model(model, to_file=mimodelo.png',
show_shapes=True, show_layer_names=True)
```



Entrenamiento del modelo

- Con el método **`fit()`**, al que se le pueden aplicar varios parámetros:
  - *epochs*: número de iteraciones (una época es una iteración sobre todo el conjunto de entrenamiento)
  - *batch size*: número de ejemplos de datos de entrenamiento a utilizar para cada actualización del gradiente
  - *validation_split*: proporción de datos de entrenamiento a utilizar en la evaluación de cada iteración (*epoch*)
  - *shuffle*: indica si barajar los datos de entrenamiento tras cada iteración

⇒ Conviene guardar la salida de la ejecución de fit() porque contiene la información sobre el rendimiento del modelo durante el entrenamiento, como la pérdida, que es evaluada tras cada época

```
epochs = 20
batch_size = 4

historia = model.fit(X_train, y_train, batch_size=batch_size,
                     epochs=epochs, validation_split=0.2, verbose=1)

plt.plot(historia.history['loss'])
plt.show()
```

Visualiza la pérdida del modelo en cada época

Evaluación del rendimiento del modelo

- El método **evaluate()** permite ver las pérdidas y métricas que se le hayan pasado al modelo para su entrenamiento

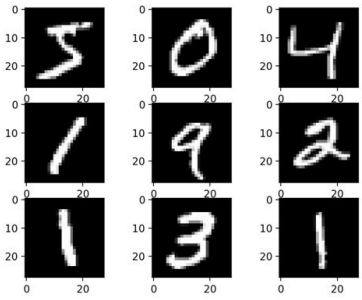
```
test_loss = model.evaluate(X_test, y_test['y'])
```

DASI

Reconocimiento de caracteres escritos

Identificación de dígitos escritos a mano

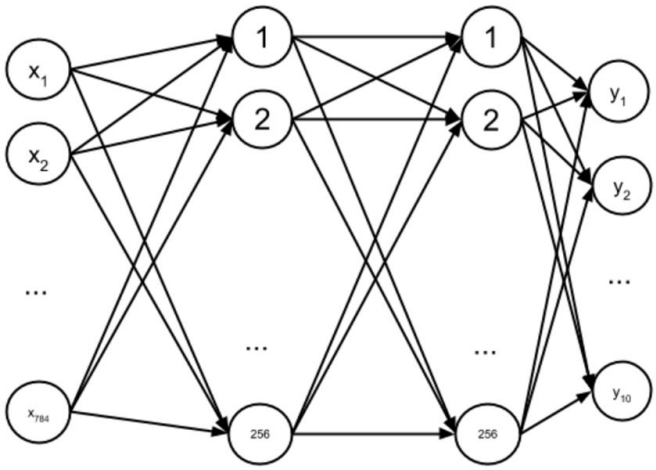
- Identificación de dígitos escritos a mano
  - Dígitos = {0..9}
  - Imágenes de 28 * 28 = 784 píxeles
- Ejemplo clásico de redes para clasificación entrenadas utilizando regresión lineal
- Utilizaremos el dataset del Modified National Institute of Standards and Technology (MNIST)
  - El Hello World! del aprendizaje automático
  - 70.000 ejemplos de dígitos escritos
    - 60.000 de entrenamiento y 10.000 de prueba
  - Imágenes de 28x28 píxeles en escala de grises
  - Keras incluye este dataset



License: Yann LeCun and Corinna Cortes hold the copyright of MNIST dataset, which is a derivative work from original NIST datasets. MNIST dataset is made available under the terms of the Creative Commons Attribution-Share Alike 3.0 license.

Esquema general de la red del ejemplo

- La capa de entrada son las imágenes como vectores de 784 números, cada uno representando un píxel de la imagen
  - Cada imagen es de 28x28 = 784 píxeles
- Las capas ocultas son perceptrones de 256 neuronas cada una, que implementan una regresión lineal ($\sum w_i x_i + b$)
- La capa de salida son las diez etiquetas (esto es, los dígitos de 0 a 9)



[Atienza 2020]

Módulos necesarios para el ejemplo

- En este ejemplo hará falta importar lo siguiente al principio del programa:

```
# La librería Numpy que implementa arrays n-dimensionales
# Numpy es la base de la mayoría de librerías de aprendizaje automático
import numpy as np

# El modelo de red que se va a utilizar:
from tensorflow.keras.models import Sequential

# Clases para configurar la red:
from tensorflow.keras.layers import Dense, Activation, Dropout

# Funciones de utilidad para convertir arrays en vectores y
visualización:
from tensorflow.keras.utils import to_categorical, plot_model

# El conjunto de datos (dataset) que se va a utilizar, MNIST:
from tensorflow.keras.datasets import mnist

# Módulo de Python utilizado frecuentemente para visualizar gráficos:
import matplotlib.pyplot as plt
```

Importación del juego de datos (dataset)

- Como este dataset ya está incluido en Keras, se puede cargar fácilmente:

```
# Carga el juego de datos:
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

- La función **load_data()** de mnist devuelve una tupla de arrays Numpy, para el conjunto de entrenamiento y el de prueba:
  - **x_train, x_test**: uint8 arrays of grayscale image data with shapes (num_samples, 28, 28), esto es, cada imagen es un array de píxeles cuyo valor es su color en una escala de grises
  - **y_train, y_test**: uint8 arrays of digit labels (integers in range 0-9) with shapes (num_samples), son las etiquetas correspondientes a cada imagen
- Se puede probar que ha ido bien viendo el número de casos que contiene de entrenamiento y prueba para cada número:

```
# cuenta el número de elementos de entrenamiento para cada dígito):
unique, counts = np.unique(y_train, return_counts=True)
print("Elementos de entrenamiento: ", dict(zip(unique, counts)))
```

  - La función **unique()** de Numpy devuelve los elementos únicos del array que se le pasa y en este caso el número de veces de cada valor único (return_counts=True)

Importación del juego de datos (dataset)

- De forma parecida se puede ver el número de casos que contiene de prueba para cada número:

```
# cuenta el número de elementos de prueba
unique, counts = np.unique(y_test, return_counts=True)
print("Elementos de prueba (test): ", dict(zip(unique, counts))
```

- Para imprimir los 10 primeros ejemplos del conjunto de entrenamiento:

```
for i in range(10):
    print(y_train[i],end = '    ')
    plt.subplot(1, 10, i + 1)
    plt.imshow(x_train[i], cmap='gray')
    plt.axis('off')
print()
plt.show()
plt.close('all')
```

Preparación del juego de datos (dataset)

- Para la red neuronal la salida tiene que ponerse como tensores (vectores o matrices):
  - La función **to_categorical()** de keras convierte un vector de enteros (las categorías en este caso) en una matriz dispersa de valores binarios (0, 1) con tantas filas como la longitud del vector, y tantas columnas como el número de clases

```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

- Por ejemplo, los 10 primeros dígitos anteriores del conjunto de entrenamiento quedarían así:

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]  [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]  [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]  [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```

  - Esto es, 1 en la posición correspondiente al dígito, empezado con el 0 en la primera, el 1 en la segunda, etc.:
    - el 5 es [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
    - el 0 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
    - etc.

Preparación del juego de datos (dataset)

- La entrada será un tensor de una dimensión, para lo cual habrá que:
  - Transformar los datos de las imágenes, que vienen en dos dimensiones (28x28 = 784 píxeles)
  - Y normalizar el color de cada pixel que es de 0 a 255 a un valor entre 0 y 1:

```
# obtiene la dimensión de las imágenes (asumiendo que son cuadradas)
image_size = x_train.shape[1]       # 28
pixeles = image_size * image_size   # 28x28=784 píxeles de cada imagen

# redimensiona y normaliza
x_train = np.reshape(x_train, [-1, pixeles]) # transforma a [60,000, 28 * 28]
x_train = x_train.astype('float32') / 255  # normaliza en el rango [0:1]
x_test = np.reshape(x_test, [-1, pixeles]) # transforma a [10,000, 28 * 28]
x_test = x_test.astype('float32') / 255
```

Construcción del modelo de red

- En este caso 3 capas con ReLU y dropout tras cada capa
  - Las dos primeras capas de tipo Dense
    - Con 256 elementos porque otras opciones (512 o 1024) no mejoran el resultado sensiblemente, y con 128 la red converge rápidamente pero tiene menos exactitud (accuracy)
    - ReLU para eliminar la linearidad (ReLU es la más comúnmente usada para esto)
  - El dropout para hacer más robusta la red, evitando que la red memorice demasiado los datos de entrenamiento. Por ejemplo, con dropout = 0.45 solo (1 - 0.45) * 256 elementos de la capa 1 participan en la capa 2

```
# parámetros de la red
batch_size = 128
hidden_units = 256
dropout = 0.45
# estructura sencilla de capas conectadas secuencialmente
model = Sequential()
model.add(Dense(hidden_units, input_dim=pixeles))
model.add(Activation('relu'))
model.add(Dropout(dropout))
model.add(Dense(hidden_units)) # segunda capa oculta
model.add(Activation('relu'))
model.add(Dropout(dropout))
```

La primera capa recibe la imagen

Construcción del modelo de red

- La tercera capa genera la salida
  - Al final se aplica una función de activación Softmax para mapear escalares a porcentajes:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{N-1} e^{x_j}}$$

  - Se aplica a todas las salidas (10)
  - La suma de todas las probabilidades resultantes es 1.0
  - Se genera un tensor de una dimensión
  - Se puede usar **numpy.argmax()** para determinar la categoría que tiene mayor probabilidad (esto es, el dígito correspondiente)
  - Otras posibles funciones de activación: linear, sigmoid, tanh

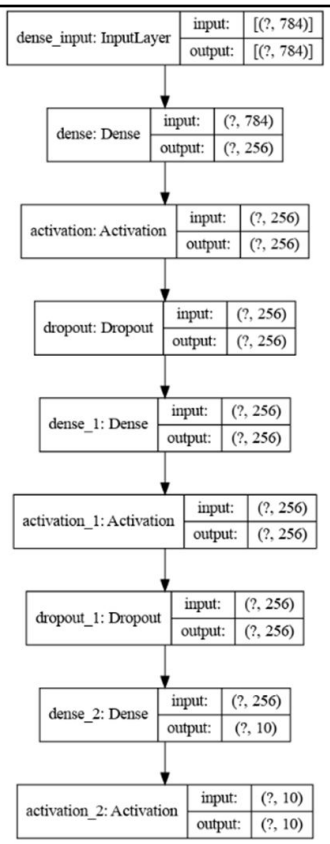
```
model.add(Dense(num_labels)) # tercera capa, genera la salida
model.add(Activation('softmax'))
```

Visualización del modelo de red

- Se puede ver un resumen y visualizar con:

```
model.summary()   # resumen
plot_model(model, # visualización
    to_file='mlp-mnist.png',
    show_shapes=True)
```

```
Model: "sequential"
Layer (type) Output Shape Param #
=================================================================
(Dense) (None, 256) 200960
_________________________________________________________________
(Activation) (None, 256) 0
_________________________________________________________________
(Dropout) (None, 256) 0
_________________________________________________________________
(Dense) (None, 256) 65792
_________________________________________________________________
(Activation) (None, 256) 0
_________________________________________________________________
(Dropout) (None, 256) 0
_________________________________________________________________
(Dense) (None, 10) 2570
_________________________________________________________________
(Activation) (None, 10) 0
=================================================================
params: 269,322 Trainable params: 269,322 Non-trainable params: 0
```



Parámetros de compilación del modelo

- El objetivo de la optimización es minimizar la función de pérdida (loss)
  - Hay que utilizar una métrica para determinar si el modelo ha aprendido
    - En este caso para clasificación se suele utilizar categorical_crossentropy
    - Otras métricas se pueden aplicar en el entrenamiento, la validación y las pruebas
  - Keras ofrece varios optimizadores: stochastic gradient descent (SGD), Adaptive Moments (Adam), y Root Mean Squared Propagation (RMSprop), cada uno con un conjunto de parámetros (especialmente la tasa de aprendizaje, learning rate)
    - En este caso se elige Adam por tener la mayor exactitud de prueba

```
# Selecciona la función de pérdida (loss),
# el optimizador Adam,
# y la métrica accuracy que es adecuada para tareas de clasificación
model.compile(loss='categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy'])
```

Entrenamiento del modelo

- El dataset y sus subconjuntos han de ser representativos:
  - Entrenamiento 80%-90%
  - Validación 5%-10%
  - Test 7%-15%
  - Ej. en el dataset cargado, 60000 ejemplos de entrenamiento y 10000 de test. Los de entrenamiento habrá que dividirlos para entrenamiento y validación
- El modelo se entrena llamando a la función **fit()** indicando
  - Los conjuntos de entrenamiento
  - Épocas: cuántas rondas de entrenamiento se hacen
  - El batch_size es el tamaño de la muestra del número de instancias de entrenamiento a las que se somete la red antes de una actualización de los pesos
    - Una época contendrá muchos lotes

```
# entrena la red
model.fit(x_train, y_train, epochs=20, batch_size=batch_size)
```


Validación del modelo

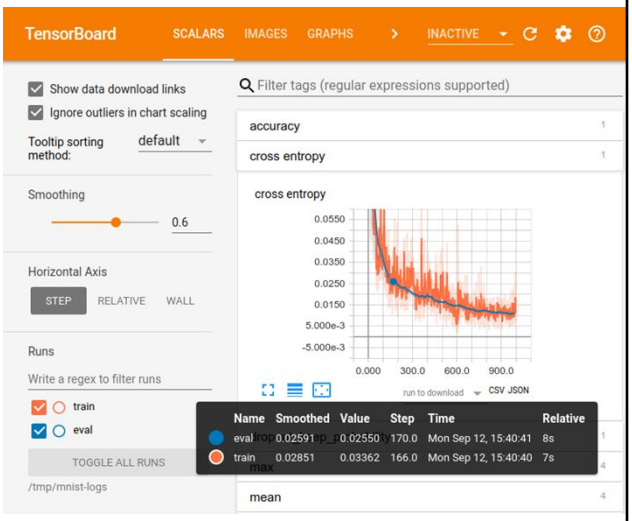
- Tras el entrenamiento ya se tiene el clasificador de dígitos listo y habrá que evaluar su rendimiento (performance)
  - Esta etapa permite medir los resultados que se obtienen variando los distintos parámetros que se han utilizado
  - La accuracy del test debería ser algo menor que la de validación

```
# valida el modelo con el conjunto de datos de prueba
test_loss, test_acc = model.evaluate(x_test, y_test,
                        batch_size=batch_size,
                        verbose=0)

print("Test accuracy: %.1f%%" % (100.0 * test_acc))
```

TensorBoard

- Las redes de Deep Learning (en general las ANN) pueden tener múltiples arquitecturas y parámetros para un mismo problema
  - Hay que probar y comparar
  - Se necesitan herramientas de depuración
- TensorBoard permite:
  - Seguir y visualizar métricas
  - Visualizar el grafo del modelo (operaciones y capas)
  - Ver histogramas de parámetros (ej. pesos y sesgos)
  - …



https://www.tensorflow.org/guide/summaries_and_tensorboard

El mismo ejemplo con redes convolucionales

- En vez de regresión lineal se puede procesar la imagen pasando una máscara (kernel) para extraer características
  - Esta máscara se va desplazando por la imagen, un pixel a la derecha y luego hacia abajo, por ejemplo, y generando
  - En el ejemplo abajo, es un kernel de 3x3 que se aplica a un input feature map (la imagen) y se genera otro feature map de menor tamaño
  - Esta operación se denomina convolución

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} & p_{15} \\ p_{21} & p_{22} & p_{23} & p_{24} & p_{25} \\ p_{31} & p_{32} & p_{33} & p_{34} & p_{35} \\ p_{41} & p_{42} & p_{43} & p_{44} & p_{45} \\ p_{51} & p_{52} & p_{53} & p_{54} & p_{55} \end{bmatrix} \otimes \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}$$

$$f_{11} = p_{11}a_{11} + p_{12}a_{12} + p_{13}a_{13} + p_{21}a_{21} + p_{22}a_{22} + p_{23}a_{23} + p_{31}a_{31} + p_{32}a_{32} + p_{33}a_{33}$$

[Atienza 2020]

El mismo ejemplo con redes convolucionales

- Los cambios al programa anterior serían los siguientes:
  - Añadir los elementos para crear las capas de convolución:
    from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
  - Redimensionar la entrada a tres dimensiones: 2 para los píxeles de la imagen y 1 para la escala de grises de cada punto
    x_train = np.reshape(x_train,[-1, image_size, image_size, 1])
    x_test = np.reshape(x_test,[-1, image_size, image_size, 1])
    x_train = x_train.astype('float32') / 255
    x_test = x_test.astype('float32') / 255
  - Cambiar los parámetros de configuración de la red:
    # la imagen es procesada como un cuadrado de escala de grises
    input_shape = (image_size, image_size, 1)
    batch_size = 128
    kernel_size = 3
    pool_size = 2
    filters = 64
    dropout = 0.2

El mismo ejemplo con redes convolucionales

- Y definir una nueva estructura de capas de la red:

```
model = Sequential()

model.add(Conv2D(filters=filters,kernel_size=kernel_size,
    activation='relu',input_shape=input_shape))

model.add(MaxPooling2D(pool_size))

model.add(Conv2D(filters=filters,kernel_size=kernel_size,
    activation='relu'))

model.add(MaxPooling2D(pool_size))

model.add(Conv2D(filters=filters,kernel_size=kernel_size,
    activation='relu'))

model.add(Flatten())

# dropout added as regularizer
model.add(Dropout(dropout))

# output layer is 10-dim one-hot vector
model.add(Dense(num_labels))
model.add(Activation('softmax'))
```

Validación del modelo

- Tras el entrenamiento ya se tiene el clasificador de dígitos listo y habrá que evaluar su rendimiento (performance)
  - Esta etapa permite medir los resultados que se obtienen variando los distintos parámetros que se han utilizado
  - La accuracy del test debería ser algo menor que la de validación

```
# valida el modelo con el conjunto de datos de prueba
test_loss, test_acc = model.evaluate(x_test, y_test,
                        batch_size=batch_size,
                        verbose=0)

print("Test accuracy: %.1f%%" % (100.0 * test_acc))
```


DASI

Conclusiones

Conclusiones

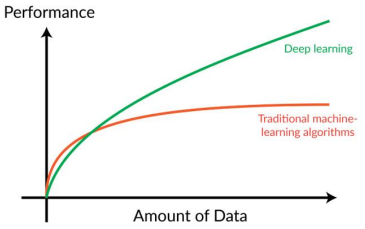
- Las RNA son un paradigma “clásico” de la IA
  - Existen desde los inicios: el Perceptrón
  - Pero no ha sido hasta ahora que la capacidad de procesamiento y la enorme cantidad de datos disponibles ha permitido su desarrollo
- El Deep Learning lleva las RNA tradicionales a una mayor complejidad y resuelve varios problemas de las RNA clásicas
  - Más capas, interconexiones y tipos de neuronas
  - Algoritmo de backpropagation
- Múltiples herramientas para aprendizaje automático
  - scikit-learn (https://scikit-learn.org/)
  - Keras (https://keras.io/)
  - Pytorch (https://pytorch.org/)
  - Scikit-learn (https://scikit-learn.org)
  - TensorFlow (https://www.tensorflow.org/)
  - PyCaret (https://pycaret.readthedocs.io/)

Conclusiones

- El **proceso de entrenamiento** de un modelo de aprendizaje automático tiene que abordarse de forma **iterativa**: empezar con un modelo sencillo y evaluar su eficiencia con métricas
- Una característica importante es el tratamiento de los datos **jerárquicamente**, esto es, de tratar elementos más simples en las primeras capas a otros de mayor nivel en las finales
  - Ejemplo: para tratamiento de imágenes, en las primeras capas se aprenden características como líneas, gradientes, para generar luego formas de objetos
  - La red puede **aprender estructuras latentes en datos no estructurados**

Conclusiones

- **Ventajas** sobre otros métodos de aprendizaje automático
  - Mejor **rendimiento**
    - Ejemplo: clasificación de imágenes (concurso ImageNet)
  - **Escalabilidad** de forma eficiente con los datos
    - La regresión logística o los árboles de decisión tienen un límite de eficiencia pero una RNA puede aprender más con mayor cantidad de datos (ImageNet usa 14 millones de imágenes etiquetas)
  - Las RNA no requieren ingeniería de características, esto es **pueden descubrir qué características son importantes** para el modelo
    - Por ejemplo no hace falta indicar el color o tamaño de animales para que se identifiquen en imágenes



  - **Adaptables y transferibles**: pesos y características se pueden aplicar en problemas similares
    - En tareas de visión por computador se suelen usar modelos de clasificación pre-entrenados como punto de partida para otras tareas de clasificación

Conclusiones

- **Ventajas de métodos de aprendizaje automático tradicionales** sobre RNA
  - **Mejor rendimiento cuando el conjunto de datos es pequeño**
    - Cuanto más profunda es la RNA más cantidad de datos requiere su entrenamiento
  - **Menor coste**
    - Computacionalmente las redes profundas pueden llegar a requerir demasiados recursos y tiempo para ajustarlas efectivamente
  - **Interpretables** (en general)
    - Muchos modelos de aprendizaje automático son fáciles de interpretar, por ejemplo, qué característica es la que tiene mayor impacto en la predicción del modelo
    - Las RNAs son una caja negra ⇒ Nuevo campo de estudio: la IA explicable (XAI)

Google Rules of Machine Learning: Best Practices for ML Engineering

https://developers.google.com/machine-learning/guides/rules-of-ml

- Regla n.o 1: No tengas miedo de lanzar un producto sin aprendizaje automático.
- Regla n.o 2: Primero, diseña e implementa métricas.
- Regla n.o 3: Elige el aprendizaje automático en lugar de una heurística compleja.
- Regla n.o 4: Mantén el primer modelo simple y haz la infraestructura correcta.
- Regla n.o 5: Prueba la infraestructura independientemente del aprendizaje automático.
- Regla n.o 6: Ten cuidado con la pérdida de datos cuando se copian las canalizaciones.
- Regla n.o 7: Convierte la heurística en elementos o gestiónalos de forma externa.
- Regla n.o 8: Conoce los requisitos de actualidad de tu sistema.
- Regla n.o 9: Detecta los problemas antes de exportar los modelos.
- Regla n.o 10: Busca los fallos silenciosos.
- Regla n.o 11: Proporciona documentación y propietarios a las columnas de atributos.
- Regla n.o 12: No pienses demasiado qué objetivo eliges optimizar directamente.
- Regla n.o 13: Elige una métrica simple, observable y con atributos para tu primer objetivo.
- ...

DASI

Bibliografía

Bibliografía

- Gonzalo Pajares Martinsanz, Pedro Javier Herrera Caro, Eva Besada Portas: *Aprendizaje Profundo*. RC Libros 2021.
- Matthew Moocarme, Mahla Abdolahnejad, and Ritesh Bhagwat: *The Deep Learning with Keras Workshop*. Packt Publishing 2020.
- Sridhar Alla & Suman Kalyan Adari: *Beginning MLOps with MLFlow. Deploy Models in AWS SageMaker, Google Cloud, and Microsoft Azure*. Apress, 2021.
- Yuxi (Hayden) Liu: *Python Machine Learning By Example, Third Edition*. Packt Publishing, 2020.

Recursos

- Documentación en línea de las distintas herramientas
  - scikit-learn User Guide. https://scikit-learn.org/stable/user_guide.html
  - keras: https://docs.scipy.org/doc/numpy/index.html
- Tutorial sobre aprendizaje automático
  - https://data-flair.training/blogs/machine-learning-tutorial/
- Los mejores videotutoriales, y además en español:
  - Canal Dot CSV: https://www.youtube.com/channel/UCy5znSnfMsDwaLlROnZ7Qbg