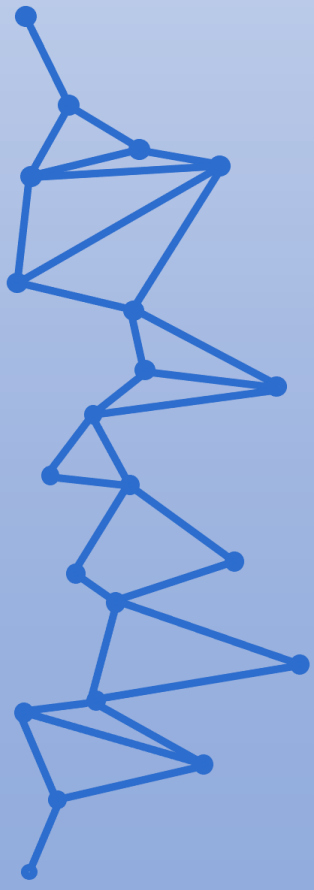
Curso de Especialización de
Inteligencia Artificial y Big Data (IABD)



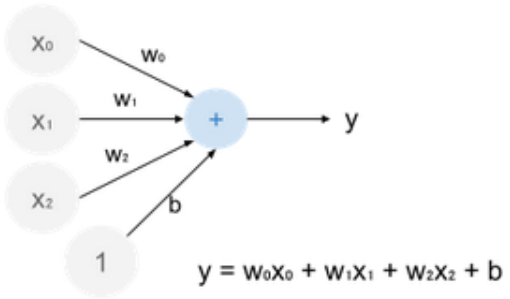
Programación de Inteligencia Artificial

UD05. Programación de redes neuronales profundas.
Resumen.

JUAN ANTONIO GARCIA MUELAS

En la técnica de un modelo de **varias capas de redes neuronales**, la clave de esta técnica es la **estructura por capas**, que en el caso de utilizar la librería Keras, viene dada por la **clase Sequential**.

La **unidad básica de una red neuronal es el perceptrón**. La **operación** que realiza éste, es tan sencilla como la **función lineal**. Algo así como y = ax + b. En el perceptrón, **tenemos tantas variables x como datos de entrada**, y en vez de usar el coeficiente "a" multiplicando a "x", usamos un tipo de coeficientes llamados "pesos" o "weights". Con esa nomenclatura, tendríamos que la ecuación lineal de un perceptrón sería:




```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(units=1,
input_shape=[3])
])
```

La capa o "**`layer`**" de tipo Dense, es la que representa verdaderamente una red neuronal con todos los nodos conectados con las diferentes variables de entrada.

El parámetro "**`units`**" se refiere al **número de neuronas que debe tener la capa.** Es **clave** en una capa **Dense**. Debe ser un número entero y positivo.

La **función de activación es opcional**. **Si no se activa** el resultado será un **cálculo lineal.** Para soluciones no lineales, debemos activarla.

El orden de las capas en un modelo de DNN (red neuronal profunda): Capa de entrada, capas internas y capa de salida.

Número de neuronas

Una de las dudas más corrientes es sobre el número de neuronas a definir en cada capa. En los **problemas de clasificación**, hay un criterio muy claro:

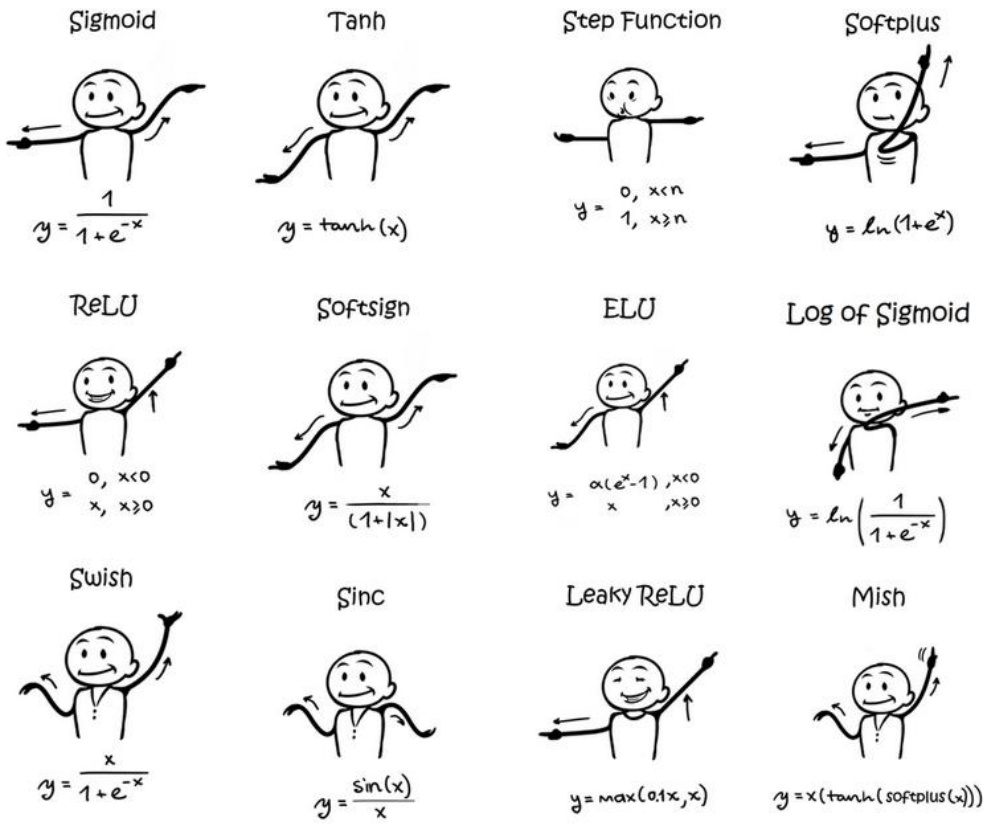
- ✓ Si es **clasificación binaria**, la capa de **salida** tendrá **una única neurona**
- ✓ SI es **clasificación múltiple**, la capa de **salida** debe tener **tantas neuronas como clases o categorías** de clasificación tenga el problema.

No hay criterio para las capas internas. Es recomendable empezar por configuraciones sencillas.

FUNCIONES DE ACTIVACIÓN

Hay muchas posibles funciones de activación, pero se suele utilizar siempre una de estas:

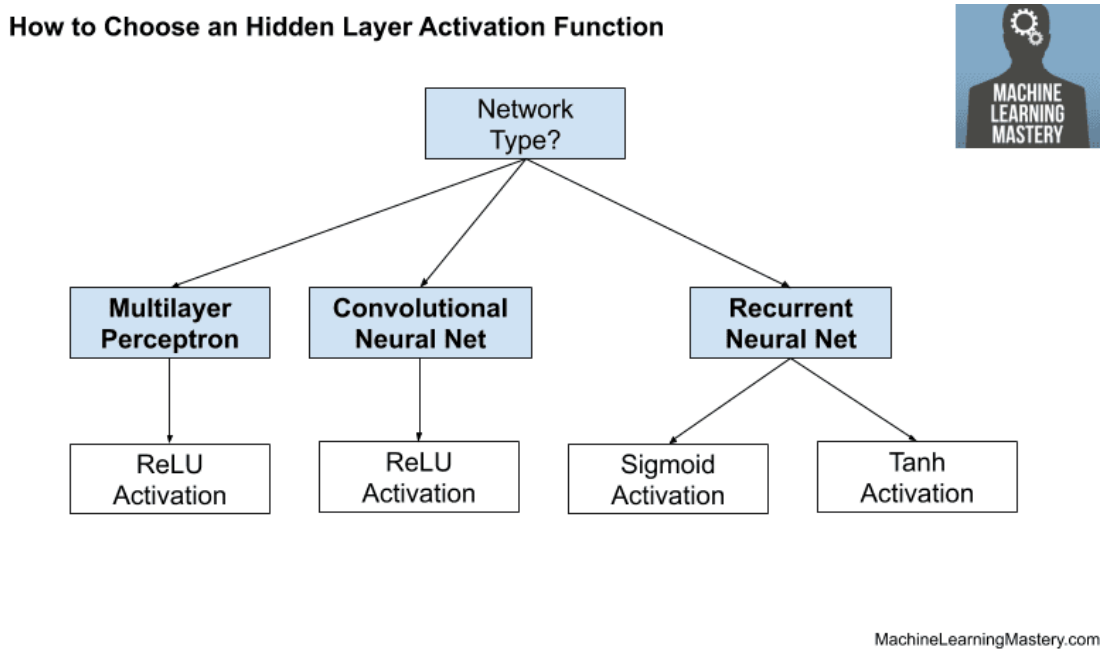
- ✓ ReLu
- ✓ Sigmoid
- ✓ Softmax
- ✓ Tanh

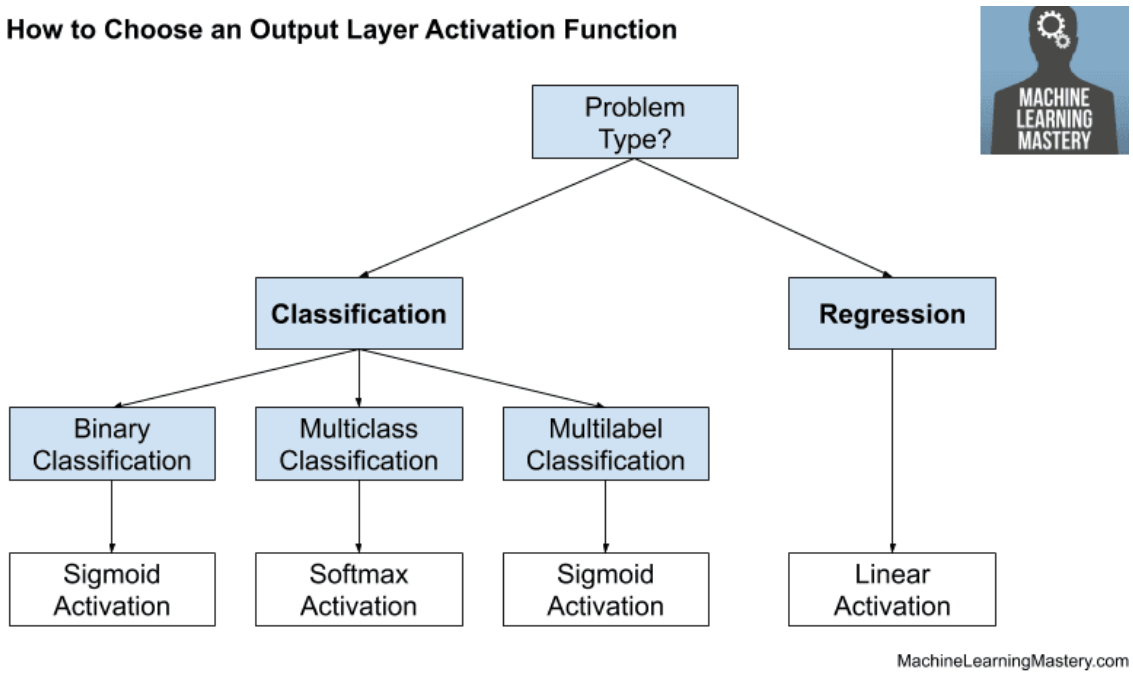


Algunos **consejos** para una configuración básica son:

- ✓ Utiliza **ReLu** para **capas internas**
- ✓ En la capa de **salida**, si estás en un problema de **clasificación binaria, utiliza Sigmoid**
- ✓ En la capa de **salida** de un problema de **clasificación múltiple, utiliza Softmax**

En estos dos esquemas de MachineLearningMastery.com tienes también una orientación para elegir función de activación según se trate de una capa interna o de la capa de salida.





Flatten

La **capa Flatten "aplana"** una estructura de datos de **entrada de más de una dimensión**, para que tengamos **un vector, o array de una dimensión**.

En el caso de **trabajar con imágenes**, lo normal es tener, como datos de entrada, una serie de **matrices o arrays de N x N pixels**. Por ejemplo, si tenemos un dataset con 1000 imágenes de 32 x 32 pixeles, la estructura de datos de entrada o dataset.shape sería: (10000, 12, 12). Al aplicar la capa Flatten, obtenemos una estructura de salida (1000, 144). Para este ejemplo, el código sería:

```
import keras

model = keras.Sequential()
model.add(keras.layers.Flatten(input_shape = (12,12))
model.add(keras.layers.Dense(64, activation = 'relu')
model.add(keras.layers.Dense(1,activation = 'sigmoid')
```

Tipos de función de coste (Loss)

La parte que se encarga de **configurar cómo será el entrenamiento**, **se controla con la función "compile".**

```
model.compile(optimizer= 'Adam', loss = 'sparse_categorical_crossentropy',
metrics=['accuracy'])
```

El **parámetro los** (El **valor medio del error entre** las **variables de salida** del modelo **y** las **etiquetas** reales.), representa lo que llamamos "**función de coste**" o "**función de pérdida**". Es una métrica necesaria para que el proceso de **ajuste de coeficientes o pesos de las redes** neuronales en las capas, se vayan actualizando hacia el modelo definitivo ya entrenado.

En general, para los **modelos basados en neuronas**, es necesario utilizar lo que se conoce como cálculo de la "**entropía cruzada**", en contraposición al error cuadrático medio o MSE.

Estas tres funciones hacen un cálculo del índice de error entre las clases reales dadas por las etiquetas y las predicciones que va dando el modelo. Pero cada una de ellas está programada para un tipo de problema de clasificación:

- ✓ **Binary Crossentropy**: es la función de coste indicada para trabajar con problemas de **clasificación binaria**, junto a la función de **activación sigmoide**.
- ✓ **Categorical Crossentropy**: adecuada para problemas de **clasificación múltiple**, pero con etiquetas o variables de **salida de tipo categórico en formato one-hot encoding**
- ✓ **Sparse Categorical Crossentropy**: es la función de coste para problemas **tanto binarios como múltiples,** pero con las etiquetas o clases de **salida dadas** como **números enteros**.

Debes conocer

En el caso de problemas de regresión, Keras proporciona las **funciones de error:**

- ✓ **MeanSquaredError: penaliza mucho los errores grandes**, y varía poco en la proximidad de los valores reales.
- ✓ **MeanAbsolutePercentageErro**r: es una métrica fácil de entender, basada en el porcentaje de error.
- ✓ **MeanSquaredLogarithmicError**: recomendable para **ignorar las anomalías o los grandes errores**.

Optimizadores

Para entrenar el modelo, necesitamos alcanzar la configuración de los pesos w en toda la red que hace que el error sea mínimo.

Como aproximación inicial al mundo de los optimizadores, te recomendamos que utilices, de momento, uno de estos dos:

- ✓ **RMSprop** presenta una **convergencia hacia el mínimo** más rápida.
- ✓ **Adam** presenta un **mejor comportamiento general**. Es una buena opción en tus **primeros entrenamientos**.

Ratio de aprendizaje o "Learning Rate"

Es el parámetro que **controla** la magnitud con la que **modificamos los pesos** en función de la pendiente de la función de coste. Es un factor que **amplifica el efecto de la pendiente de la función de coste.** Si el **learning rate** es **pequeño,** avanzará a **pasos pequeños**, lo que hará el **entrenamiento muy lento.** Si es **al revés**, hará un salto a la otra cara de la función de coste. El **proceso** de entrenamiento será **muy inestable** e incluso podría no converger. El **rango** de un learning rate de partida está entre **0,0001 y 0,001.**

En keras, al llamar a la función de optimización, podemos pasarle un valor de learning rate como argumento por clave:

```
opt = keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

Keras proporciona el método **`fit`** para el entrenamiento de un modelo basado en redes neuronales profundas. Un ejemplo de la aplicación de esta función sería:

```
model.fit(X_train, y_train, epochs = 20, batch_size = 40)
```

El método **`fit`** necesita **tres parámetros** ineludibles: los datos de **entrada X,** las etiquetas o **datos de salida** y, más, finalmente, el **número de epochs**.

El parámetro **epochs** representa el **número de iteraciones** del entrenamiento.

Las **muestras que se utilizan en cada iteración** o epoch, se gestionan con el **parámetro batch_size**. Si no se indica nada, este parámetro toma el valor, **por defecto, de 32 muestras**.

Descenso del gradiente es la técnica que nos permite encontrar o identificar el mínimo de la función de coste en una iteración del entrenamiento.