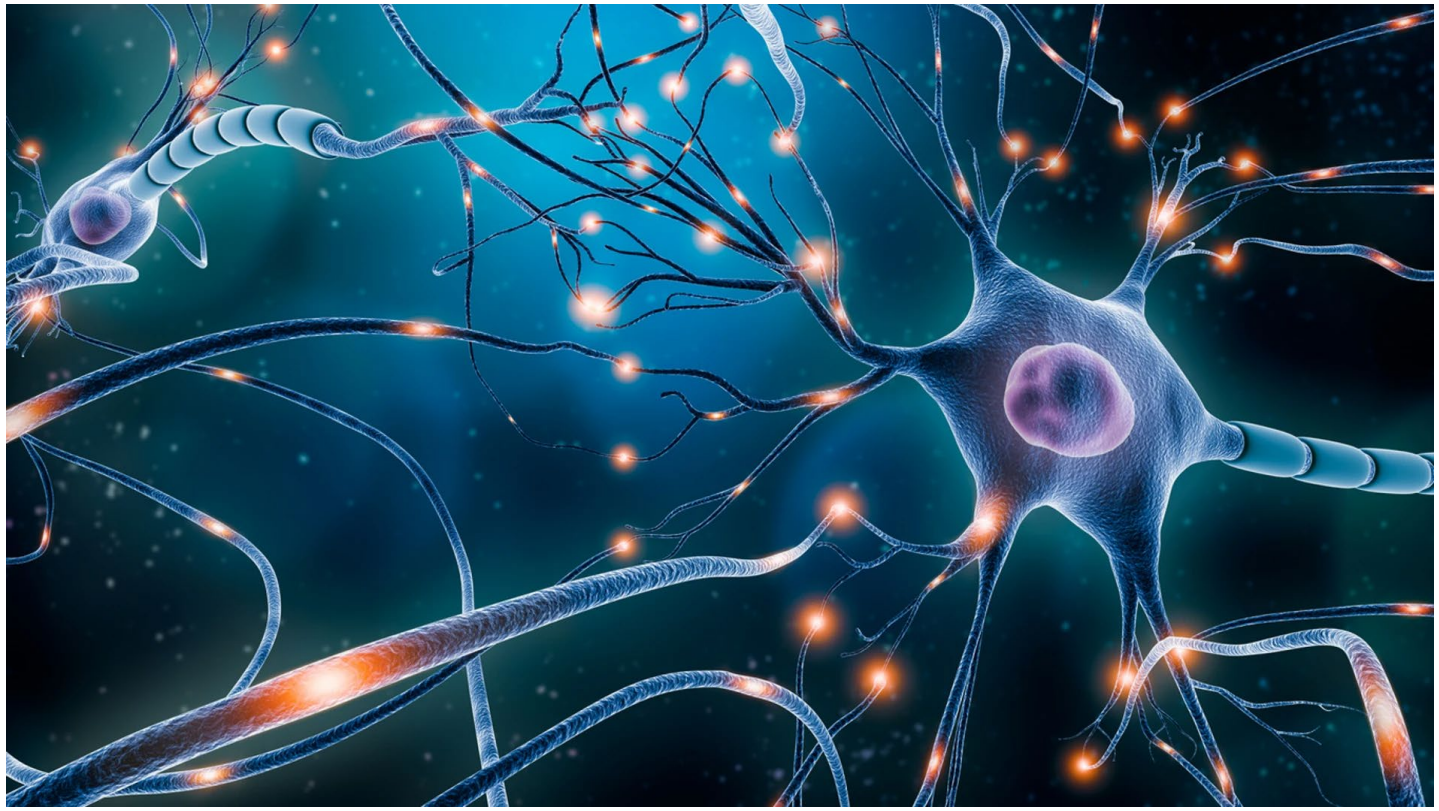


Redes Neuronales II - keras

Jonnatan Arias Garcia
jonnatan.arias@utp.edu.co
jariasg@uniquindio.edu.co

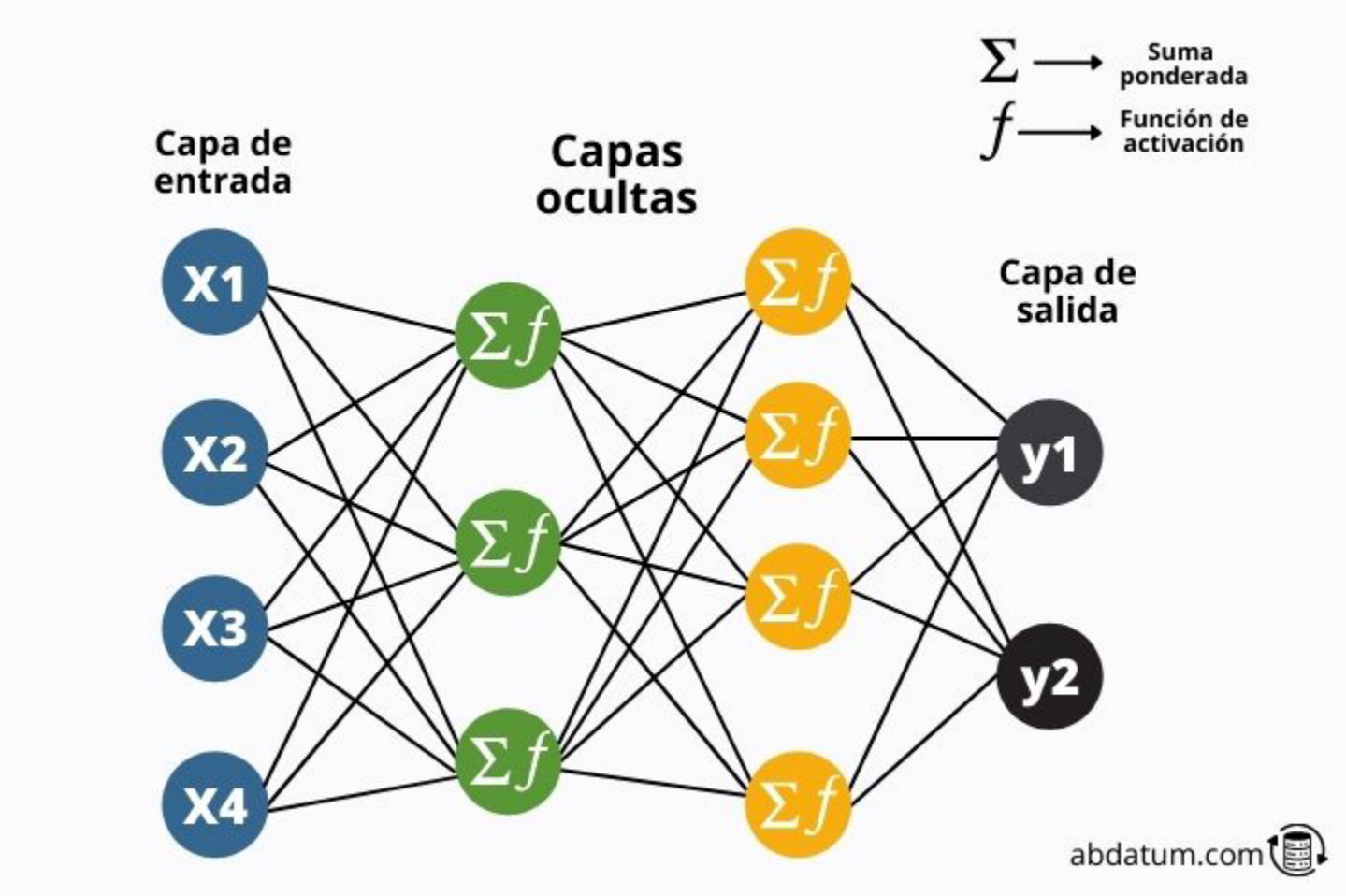
David Cardenas peña - dcardenasp@utp.edu.co
Hernán Felipe Garcia - hernanf.garcia@udea.edu.co

Contenido

- Model Sequential ->Capas
  - Core
  - Convolution
  - Pooling
  - Recurrent

- Módulo compile
- Fit, evaluate y predict
- Módulos generales

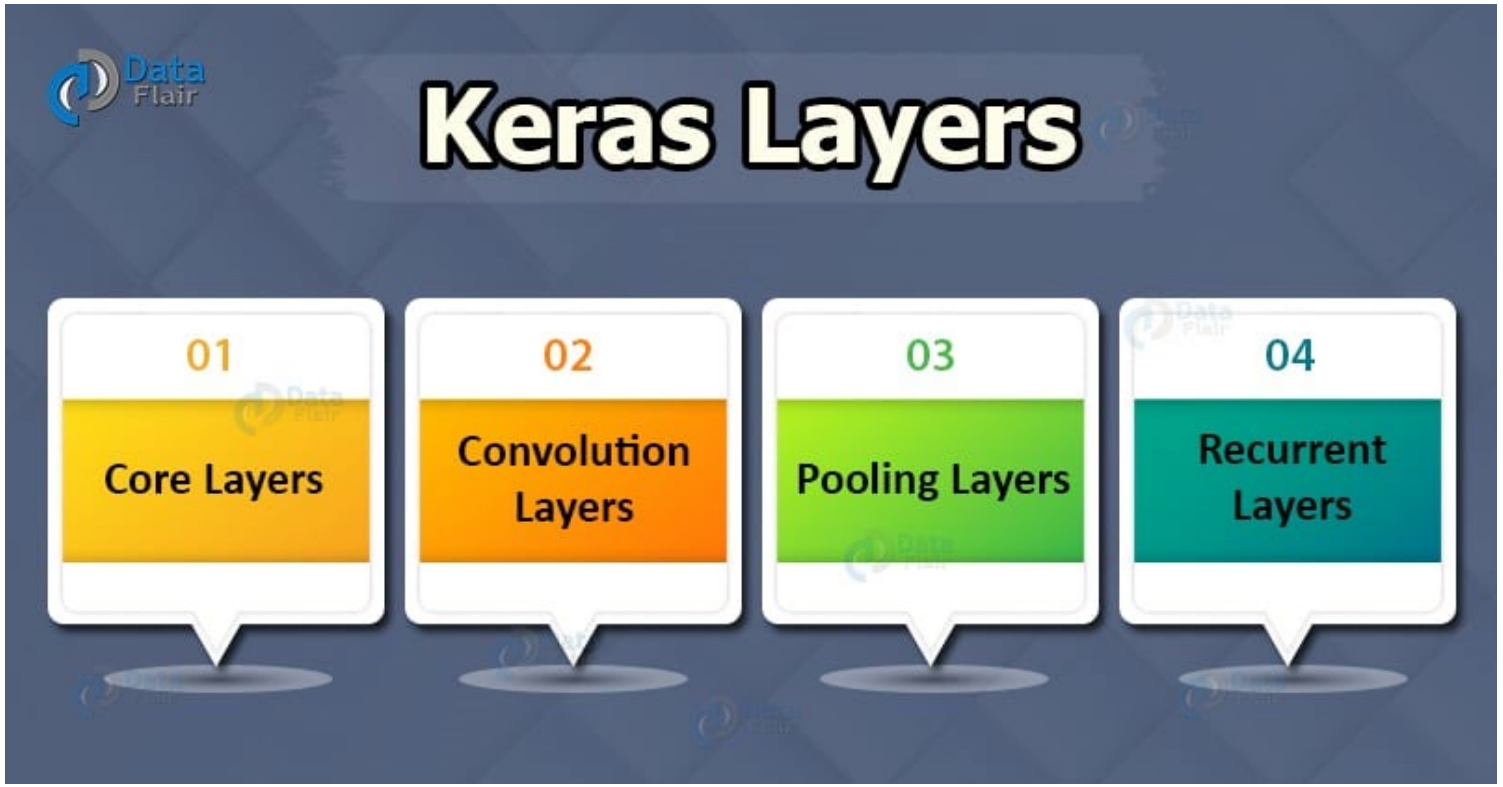
Capas de las NN



Capas I

Son la unidad principal para crear redes neuronales. Componemos una arquitectura de aprendizaje profundo mediante la adición de capas sucesivas.

Cada capa sucesiva realiza algún cálculo en la entrada que recibe. Luego, propaga la información de salida a la siguiente capa.



Capas II

Para definir o crear una capa Keras, necesitamos la siguiente información:

- **La forma de la entrada**: comprender la estructura de la información de entrada.
- **Unidades:** El número de nodos/neuronas en la capa.
- **Inicializador**: Los pesos de cada entrada para realizar el cálculo.
- **Activadores**: Para transformar la entrada en un formato no lineal, de modo que cada neurona pueda aprender mejor.
- **Restricciones**: Para poner restricciones a los pesos al momento de la optimización.
- **Regularizadores**: Para aplicar una penalización a los parámetros durante la optimización.

Capa central – Core Layer

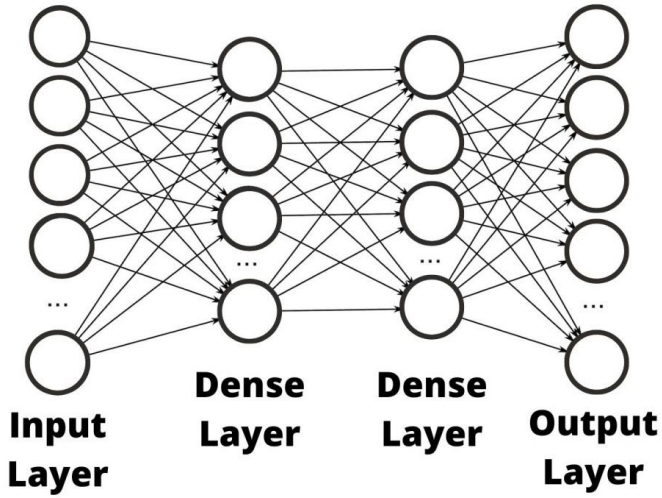
1. **Dense:** Calcula la salida de la forma

   output=activation(dot(input,kernel)+bias)

“Activation" es el activador,

“Kernel" es una matriz ponderada que aplicamos a los tensores de entrada y

“Bias" es una constante que ayuda a ajustar el modelo de la mejor manera.



La capa densa recibe información de todos los nodos de la capa anterior. Tiene los siguientes argumentos y sus valores predeterminados:

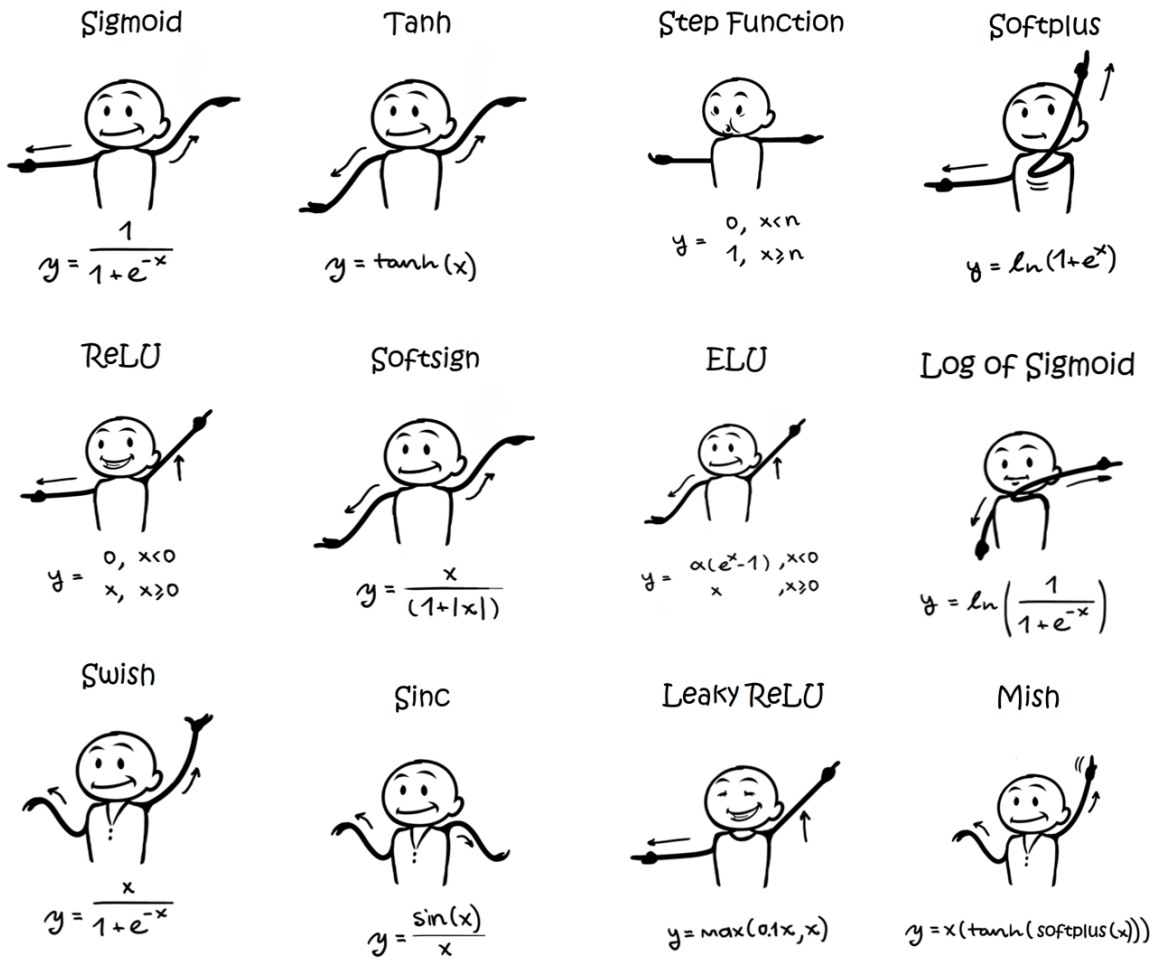
```
Dense(units, activation=NULL, use_bias=TRUE,
kernel_initializer='glorot_uniform', bias_regularizer=NULL,
activity_regularizer=NULL, kernel_constraint=NULL, bias_constrain=NULL)
```

Capa central – Core Layer

La Función de activación: Se usa para adicionar complejidad y robustez frente a invarianzas.

Por defecto: lineal

Dense(16, activation='softmax')

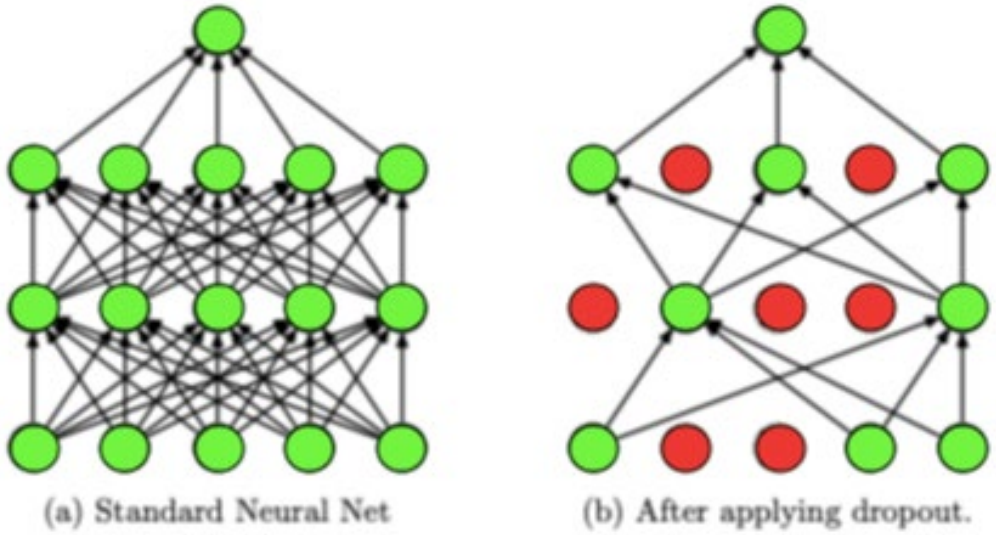


Capa central – Core Layer

Dropout: Se usa en nuestra red neuronal para evitar que se sobreajuste. Eligiendo aleatoriamente una fracción de unidades y se establece en 0 en cada actualización.

rate: Float between 0 and 1. Fraction of the input units to drop.

```
model = keras.Sequential()
model.add(layers.Dense(64, activation="relu", input_dim=25))
model.add(layers.Dropout(0.4))
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dropout(0.4))
model.add(layers.Dense(5, activation="softmax"))
```



(a) Standard Neural Net

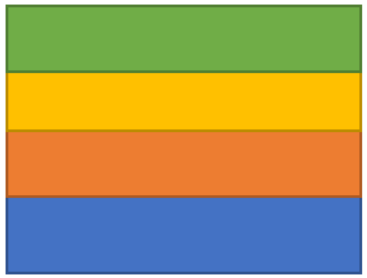
(b) After applying dropout.

Capa central – Core Layer

Flatten: Se usa para reducir la entrada en una dimensión mas baja.

rate: Float between 0 and 1. Fraction of the input units to drop.

```
model = Sequential()
model.add(Dense(16, input_shape=(3, 2)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(4))
```



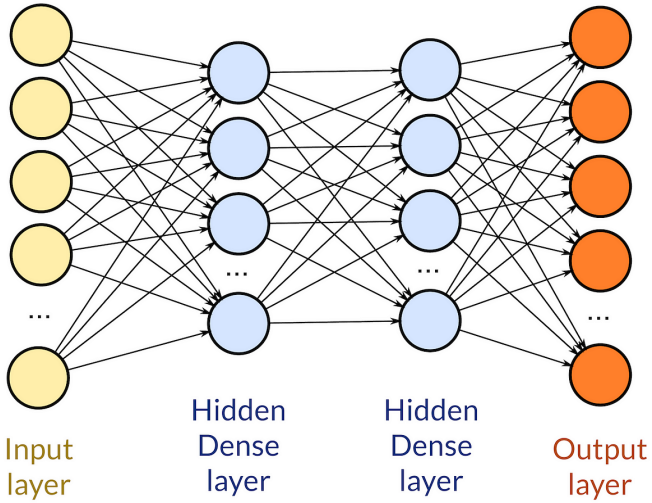
Capa central – Core Layer

Input: Sirve como punto de entrada a la NN y determina la forma y el formato de los datos de entrada.

Parametros:
shape: especifica la forma de los datos de entrada, como (height, width, channels) para imágenes o (sequence_length, input_dim) para datos secuenciales.
batch_shape: Representa el número de muestras por lote durante el entrenamiento.
name: Asigna un nombre a la capa de entrada para fines de visualización y depuración.
dtype: especifica el tipo de datos de entrada, como float32 o int32.
sparse: optimiza el uso de la memoria y los cálculos cuando los datos de entrada son escasos, es decir, contienen muchos ceros.

```
Input(shape, batch_shape, name, dtype, sparse=FALSE)
```

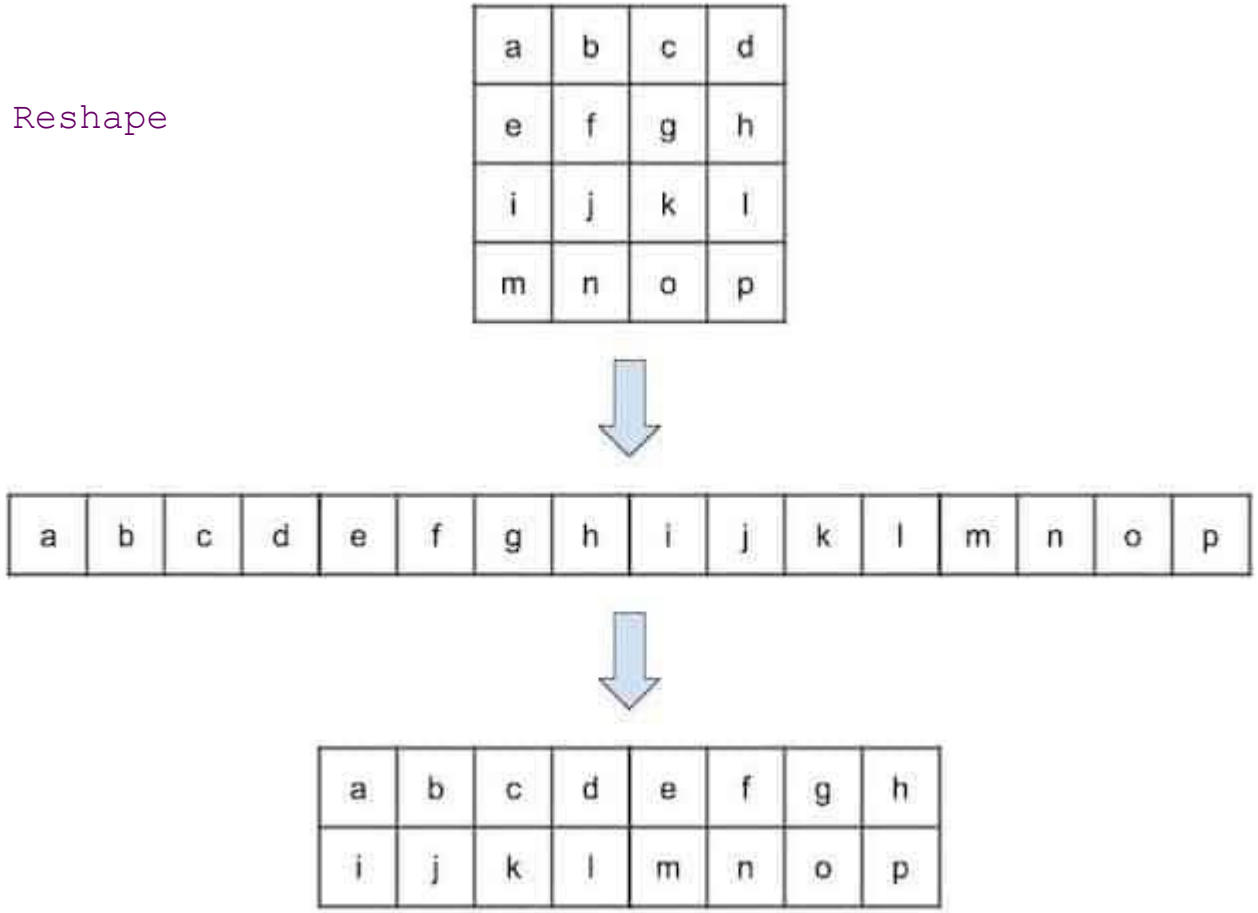
```
model = Sequential()
model.add(Dense(16, input_shape=(3, 2)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(4))
```



Capa central – Core Layer

Reshape: redimensiona a una dimensión en particulas

```
from keras.models import Sequential
from keras.layers import Activation, Dense, Reshape

model = Sequential()
layer_1 = Dense(16, input_shape = (4,4))
model.add(layer_1)
layer_2 = Reshape((2, 8))
model.add(layer_2)

layer_2.input_shape (None, 4, 4)
layer_2.output_shape (None, 2, 8)
```



Capa central – Core Layer

Permute: también se utiliza para cambiar la forma de la entrada usando un patrón. Por ejemplo, si se aplica Permute con el argumento (2, 1) a una capa que tiene una forma de entrada de (tamaño_lote, 3, 2), entonces la forma de salida de la capa será (tamaño_lote, 2, 3).

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
# now: model.output_shape == (None, 64, 10)
# note: `None` is the batch dimension
```

Capa central – Core Layer

Lambda: se utiliza para transformar los datos de entrada usando una expresión o función.

Por ejemplo, si se aplica Lambda con la expresión lambda x: x ** 2 a una capa, entonces sus datos de entrada serán elevados al cuadrado antes de procesarse.

keras.layers.Lambda(function, output_shape = None, mask = None, arguments = None)

- función representa la función lambda.
- forma_de_salida representa la forma de la entrada transformada.
- máscara representa la máscara a aplicar, si es necesario.
- argumentos representan el argumento opcional para la función lambda como un diccionario.

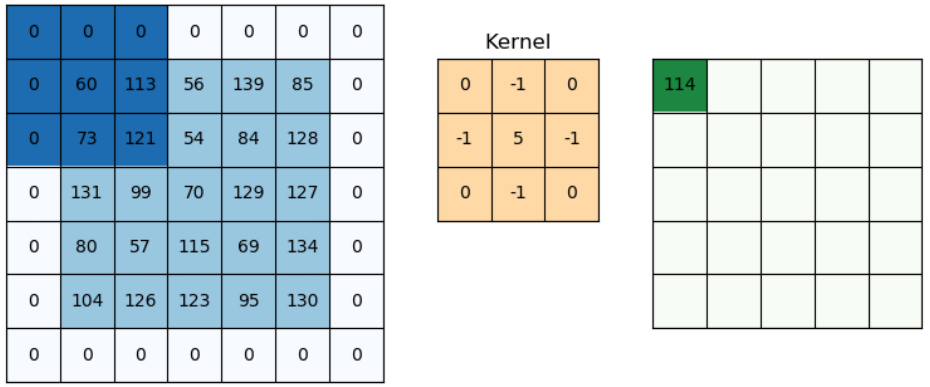
Capa Convolucional

Convolución: Operación de convolución para datos unidimensionales, bidimensionales y tridimensionales respectivamente.

Toda capa de convolución tendrá ciertas propiedades, que la diferencian de otras capas (por ejemplo, capa densa).

Conv1D, Conv2D, Conv3D

```
model.add(Conv2D(32, (3, 3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
```




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

Capas de pooling o agrupamiento

Se utiliza para realizar operaciones de agrupación máxima en datos temporales. La firma de la función ***MaxPooling1D*** y sus argumentos con el valor predeterminado es la siguiente:

```
keras.layers.MaxPooling1D ( pool_size = 2, strides = None, padding = 'valid',
data_format = 'channels_last' )
```

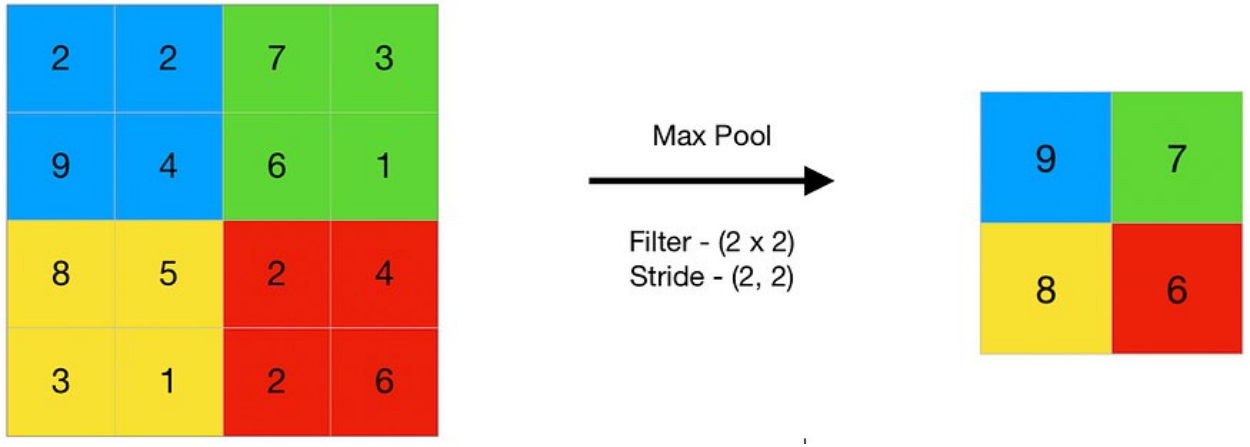
Aquí,

- **pool_size** se refiere a las ventanas máximas de agrupación.
- **Los avances** se refieren a los factores de reducción de escala.

De manera similar, MaxPooling2D y MaxPooling3D se utilizan para operaciones de agrupación Max para datos espaciales.

```
AveragePooling1D(pool_size=2, strides=None, padding='valid',
data_format='channels_last')
```

```
AveragePooling1D(pool_size=(2,2), strides=None, padding='valid',
data_format=None)
```



Capas Recurrentes

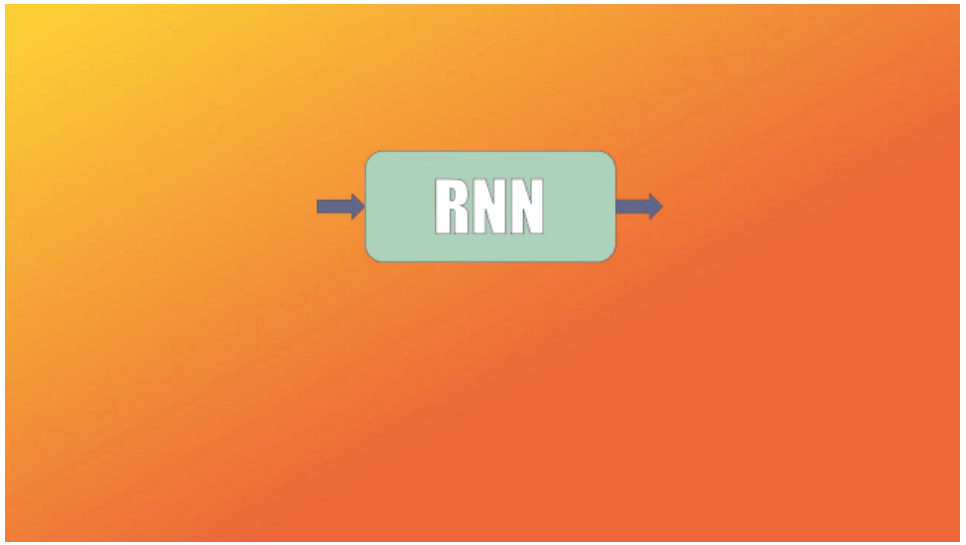
Red Neuronal Recurrente es una arquitectura que permite analizar secuencias (como texto, conversaciones o videos).

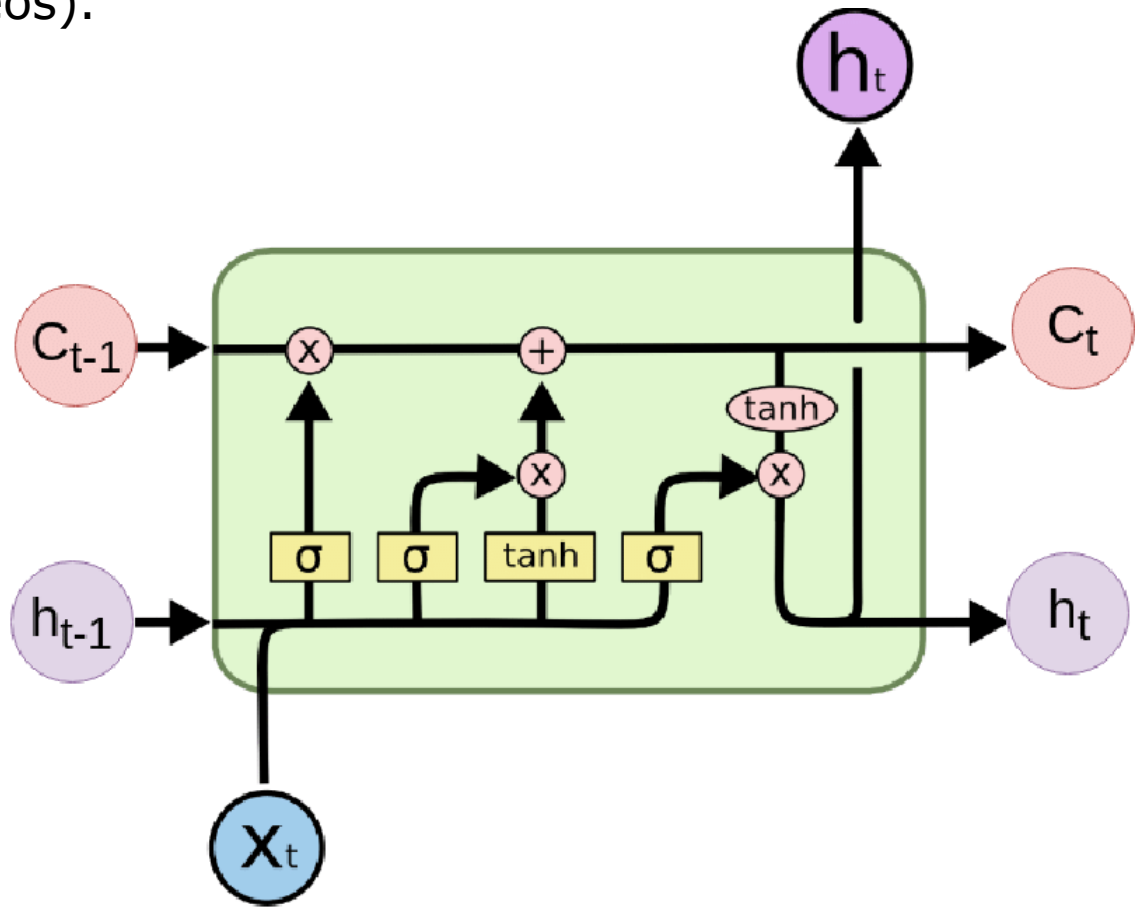
Existen dos principales:

*Simple RNN

*LSTM (modelo ajustado de RNN

LongShort Term Memory)





Contenido

- Modulo general de Compile
  - Loss
  - Optimizer
  - Metrics

Módulos Centrales (model.compile)

Keras proporciona funciones integradas relacionadas con redes neuronales para crear correctamente

Algunas de las funciones son las siguientes:

Módulo de Loss: Funciones de pérdida como mean_squared_error, mean_absolute_error, poisson, etc.

Módulo optimizador: El módulo optimizador proporciona funciones de optimización como adam, sgd, gradiente descendiente, etc.

Metrics : proporciona una lista de funciones de métricas. Podemos aprenderlo en detalle en el capítulo Entrenamiento de modelos .

Módulos loss

Loss function, es usado para encontrar el error o la desviación en el proceso de aprendizaje.

Keras, provee varias opciones, las cuales pueden funcionar bien dependiendo la tarea y los datos.

1. **mean_squared_error (MSE)**:
   Tareas de regresión: Es útil cuando se busca minimizar el error cuadrático medio entre las predicciones y los valores reales.
2. **mean_absolute_error (MAE)**:
   Tareas de regresión: Similar a MSE pero considera el error absoluto promedio. Es menos sensible a valores extremos que el MSE.
3. **mean_absolute_percentage_error (MAPE)**:
   Tareas de regresión: Útil para evaluar el rendimiento en términos de porcentaje de error absoluto medio.
4. **mean_squared_logarithmic_error (MSLE)**:
   Tareas de regresión: Es útil cuando las variables objetivo están en una escala logarítmica y se busca minimizar el error cuadrático medio entre los logaritmos de las predicciones y los logaritmos de los valores reales.

Módulos loss

5. Squared_hinge y **hinge**:

Tareas de clasificación binaria: Estas funciones de error se utilizan comúnmente en SVM (Support Vector Machine) y pueden ser útiles cuando se entrena una red neuronal para problemas de clasificación binaria.

6. categorical_hinge:

Tareas de clasificación multiclase: Similar a la función hinge, pero adaptada para problemas de clasificación multiclase.

7. logcosh:

Tareas de regresión: Se utiliza para abordar el problema de los valores atípicos en problemas de regresión, ya que penaliza menos los errores más grandes en comparación con MSE.

8. huber_loss:

Tareas de regresión: Funciona bien cuando se enfrenta a datos con ruido o valores atípicos, ya que proporciona una penalización menos severa para los errores grandes que MSE.

Módulos loss

9. Categorical_crossentropy y **sparse_categorical_crossentropy**:

Tareas de clasificación multiclase: La diferencia entre ellas radica en el formato de las etiquetas de destino: 'categorical_crossentropy' espera que las etiquetas estén codificadas en un formato one-hot, mientras que 'sparse_categorical_crossentropy' espera índices enteros de las clases.

10. binary_crossentropy:

Tareas de clasificación binaria: Es útil donde las etiquetas de destino están en formato binario.

11. kullback_leibler_divergence:

Tareas de clasificación o regresión con distribuciones de probabilidad: Mide la divergencia entre dos distribuciones de probabilidad. Se utiliza en problemas donde se desea que la salida de la red se parezca a una distribución de probabilidad específica.

Módulos loss

12. poisson:

Tareas de regresión: Es útil cuando se modelan datos de conteo que siguen una distribución de Poisson, como en problemas de predicción de recuentos.

13. cosine_proximity:

Tareas de regresión: Mide la similitud del coseno entre las etiquetas reales y las predicciones. Se puede usar en problemas de regresión donde la magnitud absoluta de las predicciones no es tan importante como la dirección.

14. is_categorical_crossentropy:

Tareas de clasificación multiclase: Similar a 'categorical_crossentropy' pero puede ser útil en situaciones donde las clases están desequilibradas y se quiere penalizar más los errores en las clases minoritarias.

Módulos Optimizer

Optimización es un proceso de ajuste “Optimo” comparando predicción mediante la función de perdida.

SGD – Stochastic gradient descent optimizer.
keras.optimizers.SGD(learning_rate = 0.01, momentum = 0.0, nesterov = False)

RMSprop – RMSProp optimizer.
keras.optimizers.RMSprop(learning_rate = 0.001, rho = 0.9)

Adagrad – Adagrad optimizer.
keras.optimizers.Adagrad(learning_rate = 0.01)

Adadelta – Adadelta optimizer.
keras.optimizers.Adadelta(learning_rate = 1.0, rho = 0.95)

Módulos Optimizer

Adam – Adam optimizer.
keras.optimizers.Adam(learning_rate = 0.001, beta_1 = 0.9, beta_2 = 0.999, amsgrad = False)

Adamax – Adamax optimizer from Adam.
keras.optimizers.Adamax(learning_rate = 0.002, beta_1 = 0.9, beta_2 = 0.999)

Nadam – Nesterov Adam optimizer.
keras.optimizers.Nadam(learning_rate = 0.002, beta_1 = 0.9, beta_2 = 0.999)

…. ETC ….

Módulos Metrics

Metrica es usada para evaluar el desempeño de nuestro modelo.

Es similar a **Loss Function**, pero no usa el proceso de entremaniento.

accuracy:

utilizada en problemas de clasificación donde se busca evaluar la precisión general del modelo en todas las clases. Es adecuada para problemas de clasificación binaria y multiclase.

binary_accuracy:

Similar a 'accuracy', pero específicamente diseñada para problemas de clasificación binaria.

categorical_accuracy:

clasificación multiclase con etiquetas codificadas en formato one-hot.

sparse_categorical_accuracy:

Similar a 'categorical_accuracy', pero diseñada para problemas de clasificación multiclase donde las etiquetas de destino son índices enteros en lugar de estar codificadas en formato one-hot. Es útil cuando las etiquetas son enteros en lugar de vectores binarios.

Módulos Metrics

top_k_categorical_accuracy:

Mide la tasa de éxito si la clase verdadera está dentro de las k clases más probables predichas por el modelo. Es útil cuando se quiere evaluar si el modelo es capaz de predecir la clase correcta entre un conjunto de clases candidatas.

sparse_top_k_categorical_accuracy:

Similar a 'top_k_categorical_accuracy', pero diseñada para problemas de clasificación multiclase donde las etiquetas de destino son índices enteros en lugar de estar codificadas en formato one-hot. Es útil cuando las etiquetas son enteros en lugar de vectores binarios.

cosine_proximity:

Se utiliza para problemas de regresión donde la similitud entre vectores de salida es importante. Mide la similitud del coseno entre las etiquetas reales y las predicciones.

Model Compile

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=False,
    steps_per_execution=1,
    jit_compile="auto",
    auto_scale_loss=True, )
```

```python
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    loss=keras.losses.BinaryCrossentropy(),
    metrics=[
        keras.metrics.BinaryAccuracy(),
        keras.metrics.FalseNegatives(),
    ],
)
```

Model Compile

Recomendaciones según el tipo de tarea:

Para tareas de predicción (regresión):

Optimizer: Adam es una buena opción para empezar debido a su buen rendimiento en una amplia variedad de problemas de regresión.

Loss: Dependiendo de la naturaleza del problema, se puede utilizar 'mse' (error cuadrático medio), 'mae' (error absoluto medio), o 'huber_loss' (pérdida de Huber).

Metrics: el loss ó ‘mse’, 'mae', 'r_squared' son métricas comunes para problemas de regresión.

Para tareas de clasificación:

Optimizer: Adam o RMSprop son opciones sólidas para comenzar en la mayoría de los casos.

Loss: Para clasificación binaria, 'binary_crossentropy'; para clasificación multiclase, 'categorical_crossentropy’.

Metrics: 'accuracy' es una métrica común para la clasificación, pero también se pueden considerar otras métricas como ‘Confusion Matix', ‘Recall', 'f1-score' dependiendo de los requisitos específicos del problema.

Módulos de entrenamiento fit

```
Model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose="auto",
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1, )
```

```
history = model.fit(
x_train,
y_train,
batch_size = 128,
epochs = 20, validation_data = (x_valid, y_valid)
)
```

Módulos de entrenamiento fit

- **X, Y** - Es una tupla con los datos a aprendes (x valores, y etiquetas)
- **epoch** : número de veces que es necesario evaluar el modelo durante el entrenamiento.
- **Batch_size**: instancias de entrenamiento.

Un lote es un subconjunto de muestras de entrenamiento que se utilizan para calcular el gradiente y actualizar los pesos del modelo

- **Validation**

validation_split:

•valor flotante entre 0 y 1 que representa la proporción de los datos de entrenamiento utilizados como validación.

•Por ejemplo, si validation_split=0.2, el 20% de los datos de entrenamiento se utilizarán como conjunto de validación, y el 80% restante se utilizará para entrenar el modelo.

•Esta opción divide automáticamente los datos de entrenamiento en un conjunto de entrenamiento y un conjunto de validación. La división se realiza de manera aleatoria, pero los datos se mantienen en el mismo orden.

validation_data:

•permite proporcionar explícitamente datos de validación para evaluar el modelo durante el entrenamiento.

•Este enfoque te da más control sobre los datos de validación, especialmente si deseas utilizar un conjunto de validación específico o si ya tienes los datos divididos en conjuntos de entrenamiento y validación

Módulos de evaluación

```
Model.evaluate(
    x=None,
    y=None,
    batch_size=None,
    verbose="auto",
    sample_weight=None,
    steps=None,
    callbacks=None,
    return_dict=False,
    **kwargs )
```

```
score = model.evaluate(x_test, y_test)

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Datos que no se han usado para entrenarse ni para validar, pero con etiquetas y.
Predice las salidas de X_test y mira si se ajusta al real y_test

Módulos predicción

```
Model.predict(
    x,
    batch_size=None,
    verbose="auto",
    steps=None,
    callbacks=None)
```

```
pred = model.predict(x_test)

pred = np.argmax(pred, axis = 1)[:5]
label = np.argmax(y_test,axis = 1)[:5]
print(pred)
print(label)
```

Le enviamos unos datos X de la misma dimensión y formato a como se entreno, pero sin etiqueta. El nos predice la etiqueta según lo que se entreno

Módulos Generales I

Módulos Generales

- **Activaciones**: La función de activación vista como elemento de una capa.
- **Regularizadores**: Funciones como regularizador L1, regularizador L2, etc.
- **Initializers** : proporciona una lista de funciones de inicializadores. Podemos aprenderlo en detalle en el capítulo de capas de Keras . durante la fase de creación del modelo de aprendizaje automático.
- **Constraints** : proporciona una lista de funciones de restricciones.

Módulos de Initializers

En Machine Learning, se asignará peso a todos los datos de entrada. El módulo de inicializadores proporciona diferentes funciones para establecer estos pesos iniciales. Algunas de las funciones del inicializador de Keras son las siguientes:

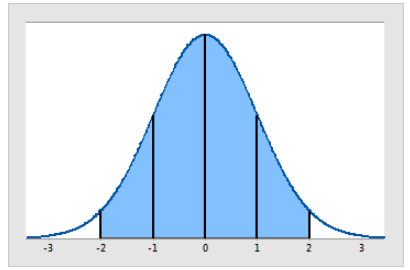
```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import initializers

my_init = initializers.Zeros()
model = Sequential()
model.add(Dense(512, activation = 'relu', input_shape = (784,),
   kernel_initializer = my_init))
```

Donde **kernel_initializer** representa el inicializador del kernel del modelo

Módulos de Initializers

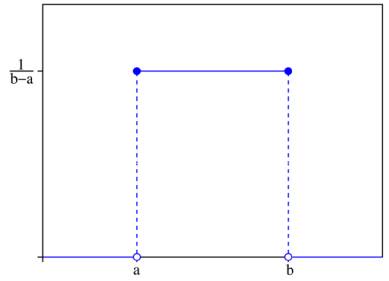
Inicializa con **Unos o zeros**

```
my_init = initializers.Ones()
my_init = initializers.Zeros()
```



Valor **constante** digamos 5

```
my_init = initializers.Constant(value=5)
```



Aleatorio **normal**

```
my_init = initializers.RandomNormal(mean=0.0, stddev=0.05)
```

Aletorio **uniforme**

```
my_init = initializers.RandomUniform(minval = -0.05, maxval =0.05)
```

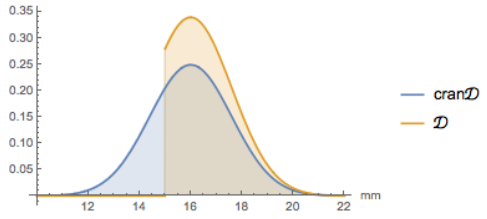
Módulos de Initializers

Normal Truncado

```
my_init = initializers.TruncatedNormal(mean=0.0, stddev=0.05)
```

Lecun_uniforme

```
my_init = initializers.lecun_uniform(seed=none)
```



Glorot_normal

```
my_init = initializers.glorot_normal(seed=none)
```

Glorot_uniforme

```
my_init = initializers.glorot_uniform(seed=none)
```

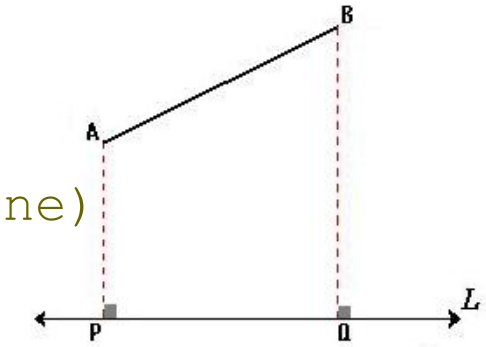
Módulos de Initializers

he_uniforme

```
my_init = initializers. He_normal(seed=none)
```

Ortogonal

```
my_init = initializers.Orthogonal(gain=1.0, seed=none)
```



Identidad

```
my_init = initializers.Identity(gain=1.0)
```

Módulos de Constraints

Se establecen restricciones en el parámetro (weight) durante la fase de optimización, para evitar overfit.

<>El módulo Restricciones proporciona diferentes funciones para establecer la restricción en la capa. Algunas de las funciones de restricción son las siguientes.

No negativo (pesos no negativos)

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import constraints

my_constrain = constraints.non_neg(axis = 0)
model = Sequential()
model.add(Dense(512, activation = 'relu', input_shape = (784,),
kernel_constraint = my_constrain))
```

Módulos de Constraints

Norma de Unidad (pesos de norma de capa sea unitaria)

```
my_constrain = constraints.UnitNorm(axis = 0)
```

Norma máxima (Norma inferior o igual a valor dado)

```
my_constrain = constraints.MaxNorm(max_value=2,axis=0)
```

Norma MinMax (pesos entre max y min)

```
my_constrain = constraints.MinMaxNorm(min_value=0.0,
                max_value=1.0,rate=1.0,axis=0)
```

Módulos de Regularizadores

los regularizadores se utilizan en la fase de optimización.

Aplica penalizaciones al parámetro de capa durante la optimización.

El módulo de regularización de Keras proporciona las siguientes funciones para establecer penalizaciones en la capa.

La regularización se aplica únicamente por capa.

Regularizador L1

```
from keras.models import Sequential
from keras.layers import Activation, Dense
from keras import regularizers


my_regularizer = regularizers.L1(0.)
model = Sequential()
model.add(Dense(512, activation = 'relu', input_shape = (784,),
kernel_regularizer = my_regularizer))
```

Módulos de Regularizadores

Regularizador L2

```
my_regularizer = regularizers.L2(0.5)
```

Regularizador L1L2

```
my_regularizer = regularizers.L1L2(l1=0.0, l2=0.0)
```

Test de clase

1. ¿Cuál de las siguientes no es una capa comúnmente utilizada en Keras?
    a) Dense
    b) Flatten
    c) Convolute
    d) LSTM
2. ¿Cuál de las siguientes funciones de activación se utiliza comúnmente para problemas de clasificación binaria en la capa de salida?
    a) ReLU
    b) Sigmoid
    c) Tanh
    d) LeakyReLu
3. ¿Cuál de los siguientes optimizadores no es parte de la biblioteca estándar de Keras?
    a) Adam
    b) SGD
    c) RMSprop
    d) Newton

Test de clase

4. ¿Cuál de las siguientes funciones de pérdida se utiliza comúnmente para problemas de regresión en Keras?

a) Binary Crossentropy
b) Categorical Crossentropy
c) Mean Squared Error
d) Hinge Loss

5. ¿Cuál de las siguientes capas se utiliza comúnmente para aplicar evitar conexiones de nerunas entre capas?

a) Dropout
b) BatchNormalization
c) L2Normalization
d) Regularization

6.¿Cuál de las siguientes funciones de activación se utiliza comúnmente en las capas ocultas de una red neuronal?

a) Sigmoid
b) ReLU
c) Softmax
d) Tanh

Test de clase

7.¿Cuál de los siguientes no es un argumento valido para el método fit() en Keras?

a) epoch
b) Loss_function
c) Validation_data
d) Batch_size

8. ¿Cuál de las siguientes capas se utiliza comúnmente para procesar datos secuenciales en keras?

a) Dense
b) Conv2D
c) LSTM
d) Dropout

9. ¿Cuál de los siguientes optimizadores se adapta a la tasa de aprendizaje dada?

a) Adam
b) adagrad
c) RMSprop
d) Newton

Test de clase

10. ¿Cuál de las siguientes funciones de pérdida se utiliza comúnmente para problemas de clasificación multiclase?
    a) Binary Crossentropy
    b) Mean Squared Error
    c) Categorical Crossentropy
    d) Huber Loss
11. ¿Cuál de las siguientes capas se utiliza comúnmente para procesar imágenes en Keras?
    a) Dense
    b) LSTM
    c) Conv2D
    d) Flatten
12. ¿Cuál de los siguientes no es un argumento válido para el método 'Compile()'?
    a) optimizer
    b) loss
    c) Validation_split
    d) epochs

Test de clase

13. ¿Cuál de las siguientes funciones de activación se utiliza comúnmente en la capa de salida de una red neuronal para problemas de regresión?
    a) ReLu
    b) Sigmoid
    c) Softmax
    d) Linear

14. ¿Cuál de las siguientes capas se utiliza comúnmente para reducir dimensionalidad de los datos en Keras?
    a) Dense
    b) Dropout
    c) Flatten
    d) MaxPooling2D

15. ¿Cuál de los siguientes optimizadores se basa en la estimación estocástica del gradiente en keras?
    a) Adam
    b) SGD
    c) RMSprop
    d) Adagrad