KIBERNUM
IT Academy

Módulo 7 – Fundamentos de Deep Learning

# Redes Neuronales Densas

Especialización en Ciencia de Datos

# Contenido

1. Framework Deep Learning.
2. Implementación Keras-Tensorflow.

KIBERNUM

# Frameworks Deep Learning

Como hemos visto, la implementación de una red neuronal, requiere mucho conocimiento y habilidad para la programación. La buena noticia es que existen frameworks y librerías eficientes que son maduras, con mucha documentación, cursos, tutoriales, y sin costo.

A continuación, nos referiremos a las librerías Tensorflow y Keras, que serán las que utilizaremos a lo largo de este curso.


KIBERNUM

# Frameworks Deep Learning


https://www.tensorflow.org/

KIBERNUM

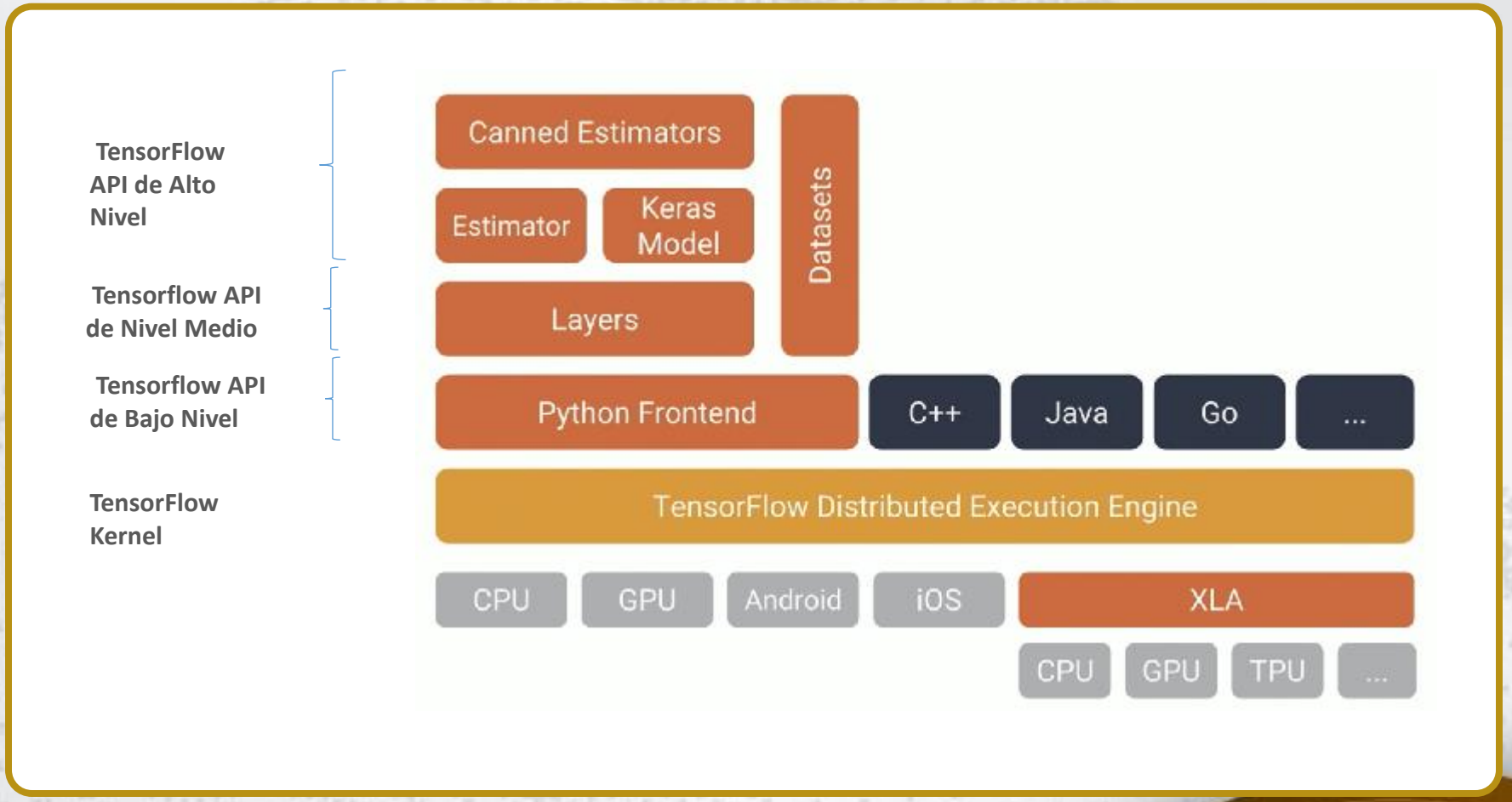
# Google Tensorflow

- Es una librería open source para cálculo numérico y machine learning de larga escala.
- Utilizado tanto para machine learning como deep learning (redes neuronales) con una metáfora común.
- Utiliza Python para proveer una interfaz para construir aplicaciones, mientras que su ejecución se realiza en C++ (alto performance).

# Google Tensorflow

- Tensorflow permite a los desarrolladores crear grafos de flujos de datos.
- Los grafos son estructuras que describen cómo se va a mover y procesar la data entre los distintos nodos de procesamiento.
- Cada nodo representa una operación matemática, y cada conexión entre nodos es un arreglo multidimensional, llamado Tensor.

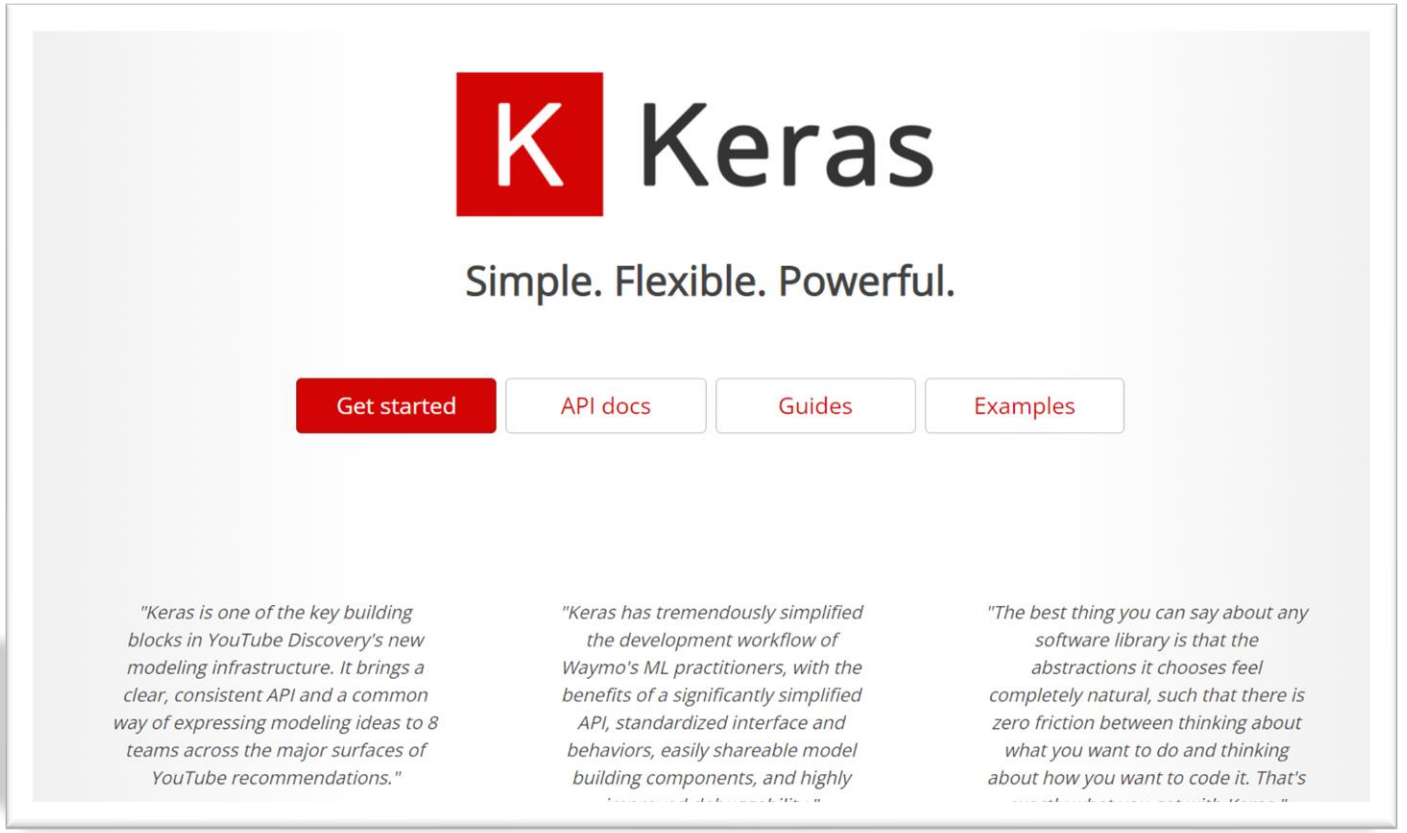
# Google Tensorflow




KIBERNUM

# Librería Keras

Keras es una librería escrita en Python que provee una API de alto nivel para interactuar múltiples backends de computación de redes neuronales.




https://keras.io/

# Librería Keras

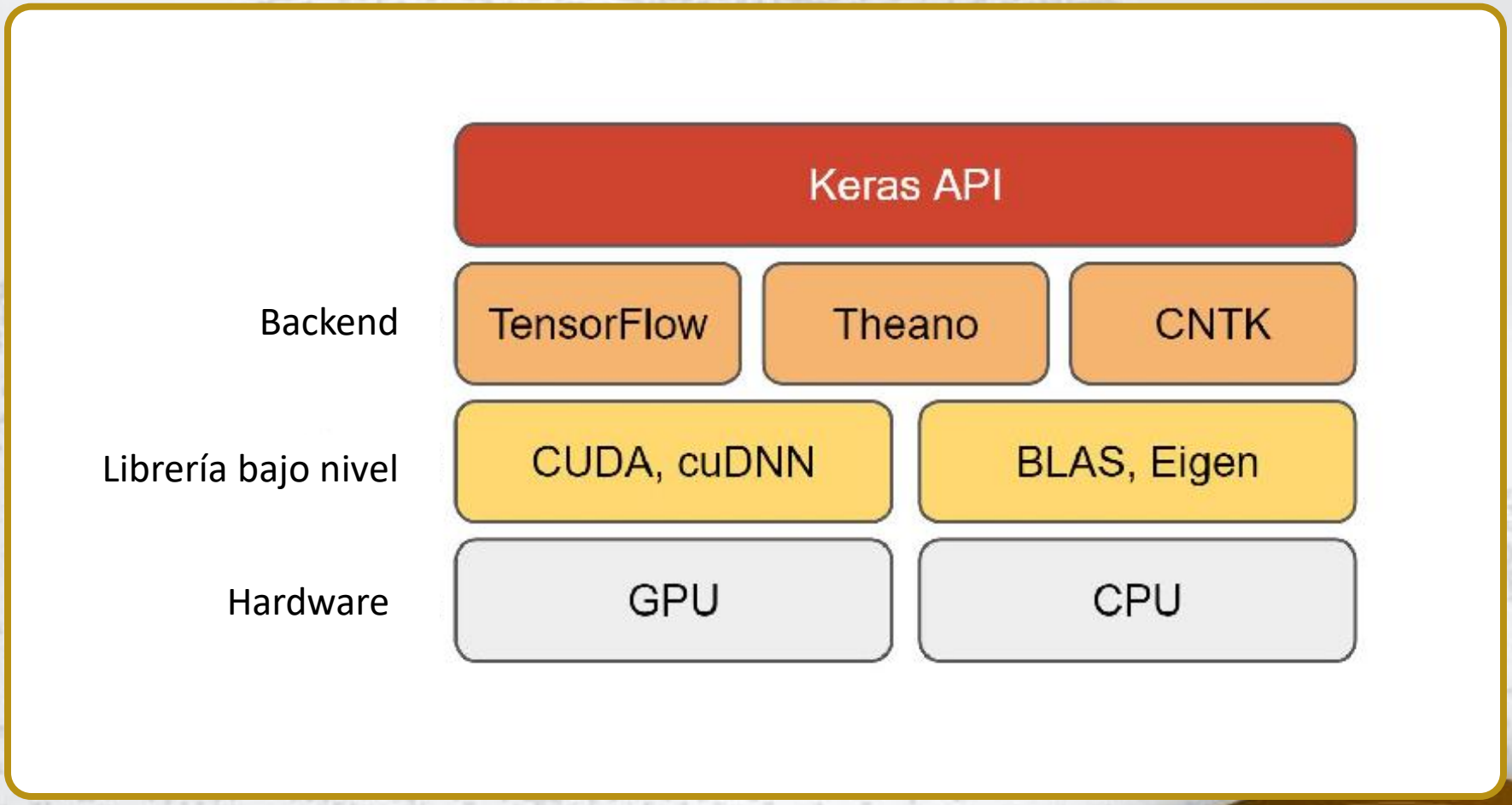
Hoy en día, ha sido adoptado por el proyecto Tensorflow como api de alto nivel en la versión 2.0 de Tensorflow.

Keras fue creado para ser amigable, modular, fácil de extender. Fue diseñada para “trabajar con humanos, no con máquinas”.

También, puede trabajar con otros backends, y soporta múltiples gpu y entrenamiento distribuido.

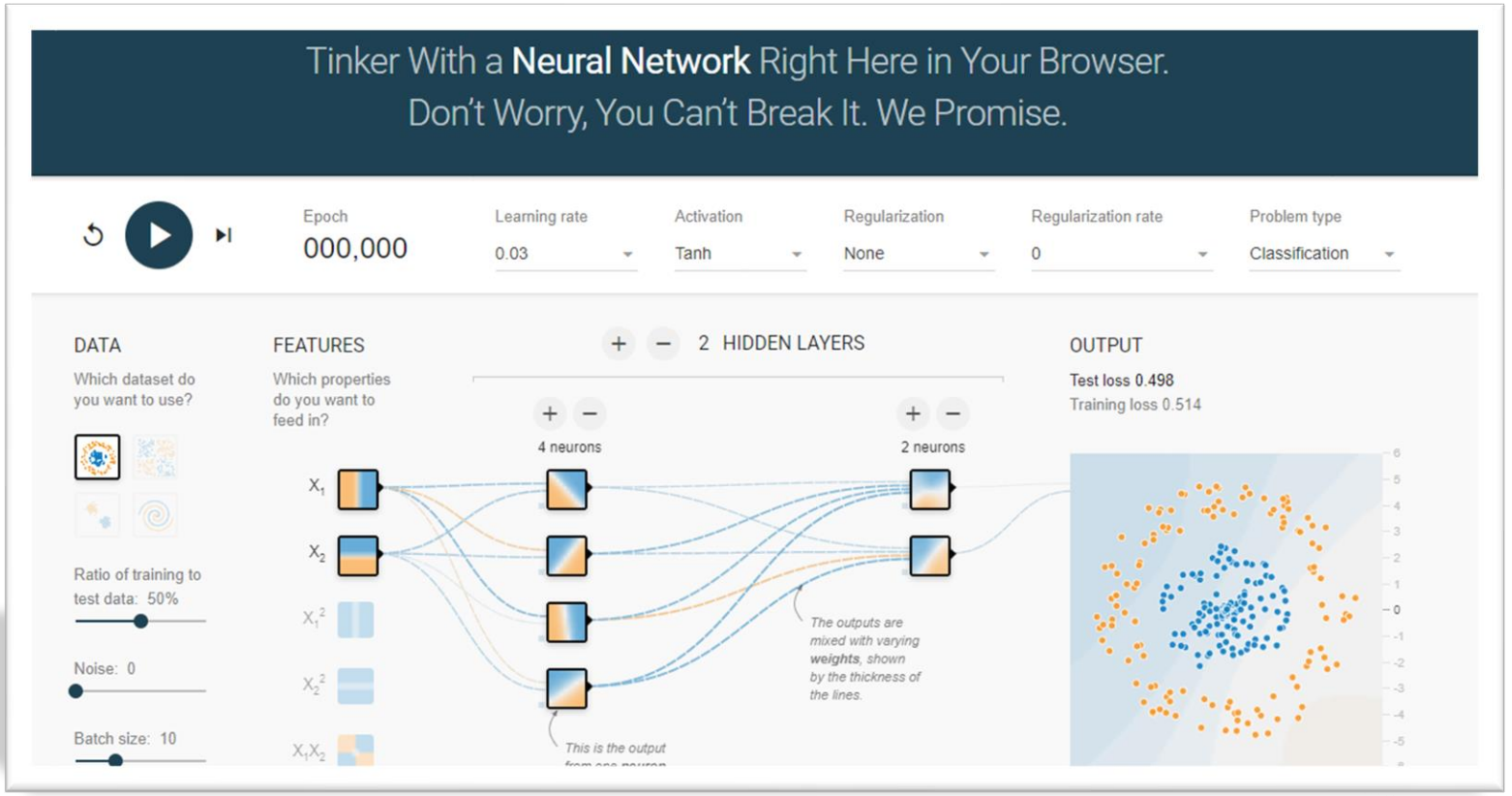

KIBERNUM

# Librería Keras



# Patio de Juegos de Tensorflow

TensorFlow Playground es un espacio de juegos con las redes neuronales.




https://playground.tensorflow.org/

KIBERNUM

# Implementación Tensorflow-Keras

KIBERNUM

# Entorno Tensorflow-Keras

Ahora implementaremos un modelo de red neuronal utilizando el stack Tensorflow-Keras. Para eso, es necesario instalar en el entorno las librerías correspondientes. Si se trabaja con Google Colab, éstas ya están instaladas.

```
# Requires the latest pip
$ pip install --upgrade pip

# Current stable release for CPU and GPU
$ pip install tensorflow

# Or try the preview build (unstable)
$ pip install tf-nightly
```


```
conda install -c conda-forge keras

conda install -c anaconda tensorflow
```

# Modelo Keras

Keras es una **biblioteca de aprendizaje profundo** de código abierto escrita en Python que permite construir, entrenar y evaluar modelos de aprendizaje profundo de manera eficiente. Un modelo de Keras es un grafo computacional que consta de capas interconectadas y puede ser utilizado para resolver una amplia variedad de tareas de aprendizaje profundo, como la clasificación de imágenes, el procesamiento de lenguaje natural y la detección de objetos.

En Keras, los modelos se construyen utilizando una API de alto nivel que permite a los usuarios definir y conectar capas de manera sencilla y flexible. Los modelos de Keras se pueden entrenar utilizando diferentes algoritmos de optimización y funciones de pérdida, y se pueden evaluar en conjuntos de datos de prueba para medir su rendimiento.

Existen varias formas de definir un modelo con la librería Keras, utilizaremos la más sencilla por ahora, que es el modelo Secuencial.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

# Definición de un Modelo Secuencial

Este es un modelo secuencial en donde se van agregando las distintas capas de la red neuronal.

Esta instrucción crea un contenedor de modelo vacío

Definimos la capa de entrada, shape=(6,) significa que hay 6 nodos de entrada

Acá se definen 2 capas ocultas densas (fully connected), con función de activación relu

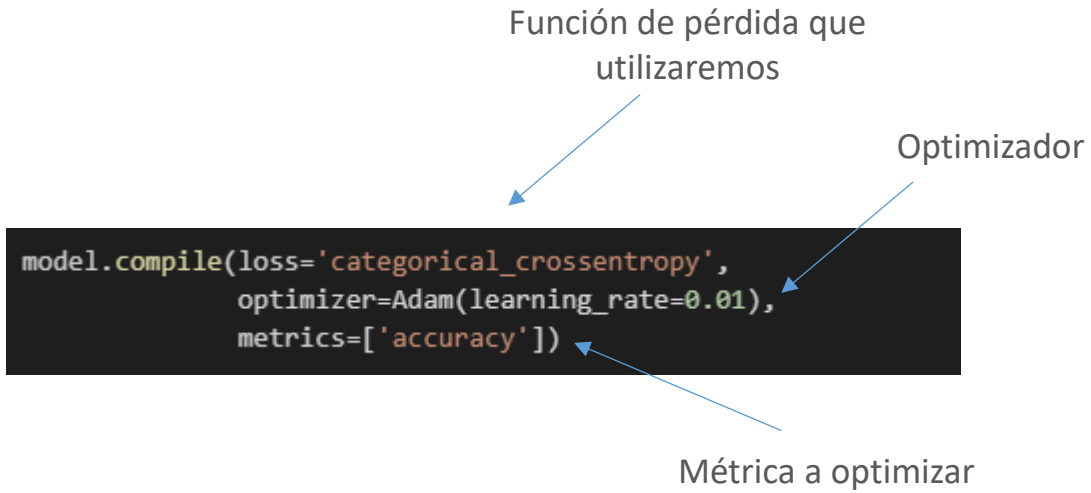
Esta es la capa de salida con dos nodos y con la función de activación softmax

```
model = Sequential()
model.add(Input(shape=(6,)))
model.add(Dense(20, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(2, activation='softmax'))
```

KIBERNUM

# Compilación Modelo Secuencial

**Definido el modelo, con sus respectivas capas, éste debe ser compilado. Para esto, se utiliza el método compile() y se le agregan nuevas instrucciones.**




KIBERNUM

# Visualización del Modelo

Este sumario resume la arquitectura del modelo que definimos anteriormente. Nótese que definimos capas densas de conexión, por lo tanto todas las conexiones entre nodos de distintas capas están activas. Este modelo tiene 372 parámetros para entrenar (pesos).

```
model.summary()
```

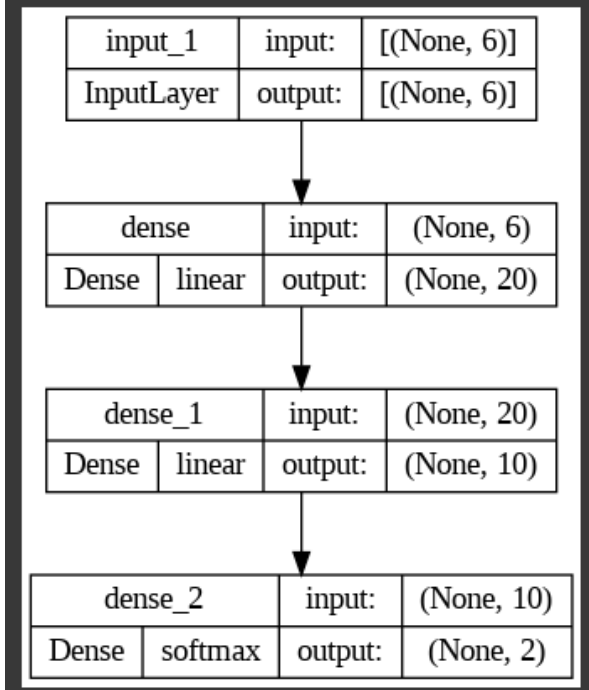
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                140

 dense_1 (Dense)             (None, 10)                210

 dense_2 (Dense)             (None, 2)                 22

=================================================================
Total params: 372
Trainable params: 372
Non-trainable params: 0
_________________________________________________________________
```

KIBERNUM

# Visualización del Modelo

```
from tensorflow.keras.utils import plot_model
```

```
plot_model(model, show_shapes=True, show_layer_activations=True)
```




Esta es otra forma de visualizar el modelo.

KIBERNUM

# Entrenamiento

Una vez definido el modelo, realizaremos el entrenamiento, en donde le presentaremos los datos del set respectivo. Acá definimos la cantidad de epochs, el tamaño de cada batch, y una proporción para validación interna.

```
history = model.fit(X_train, y_train, batch_size=128, epochs=50, validation_split=0.1)
```

```
Epoch 1/50
5/5 [==============================] - 2s 103ms/step - loss: 6.3223 - accuracy: 0.4049 -
Epoch 2/50
5/5 [==============================] - 0s 12ms/step - loss: 3.6015 - accuracy: 0.6586 - v
Epoch 3/50
5/5 [==============================] - 0s 12ms/step - loss: 2.3214 - accuracy: 0.6791 - v
Epoch 4/50
5/5 [==============================] - 0s 10ms/step - loss: 2.2729 - accuracy: 0.4496 - v
Epoch 5/50
5/5 [==============================] - 0s 14ms/step - loss: 1.0196 - accuracy: 0.5989 - v
Epoch 6/50
5/5 [==============================] - 0s 16ms/step - loss: 1.3031 - accuracy: 0.6903 - v
Epoch 7/50
5/5 [==============================] - 0s 10ms/step - loss: 0.8446 - accuracy: 0.6866 - v
Epoch 8/50
5/5 [==============================] - 0s 14ms/step - loss: 0.7462 - accuracy: 0.6157 - v
Epoch 9/50
5/5 [==============================] - 0s 18ms/step - loss: 0.6635 - accuracy: 0.6791 - v
Epoch 10/50
```

# Predicciones y Evaluación del Modelo

Como es habitual en los modelos de aprendizaje de máquina, podemos realizar predicciones sobre el set de test y posteriormente calcular las respectivas métricas de evaluación del desempeño.

```
from sklearn.metrics import accuracy_score
```

```
y_pred = model.predict(X_test)
```

```
10/10 [==============================] - 0s 2ms/step
```

```
y_pred_clase = np.argmax(y_pred,axis=1)
y_test_clase = np.argmax(y_test,axis=1)
```

```
accuracy_score(y_test_clase,y_pred_clase)
```

```
0.823728813559322
```

# Dudas y consultas

KIBERNUM

Fin Presentación