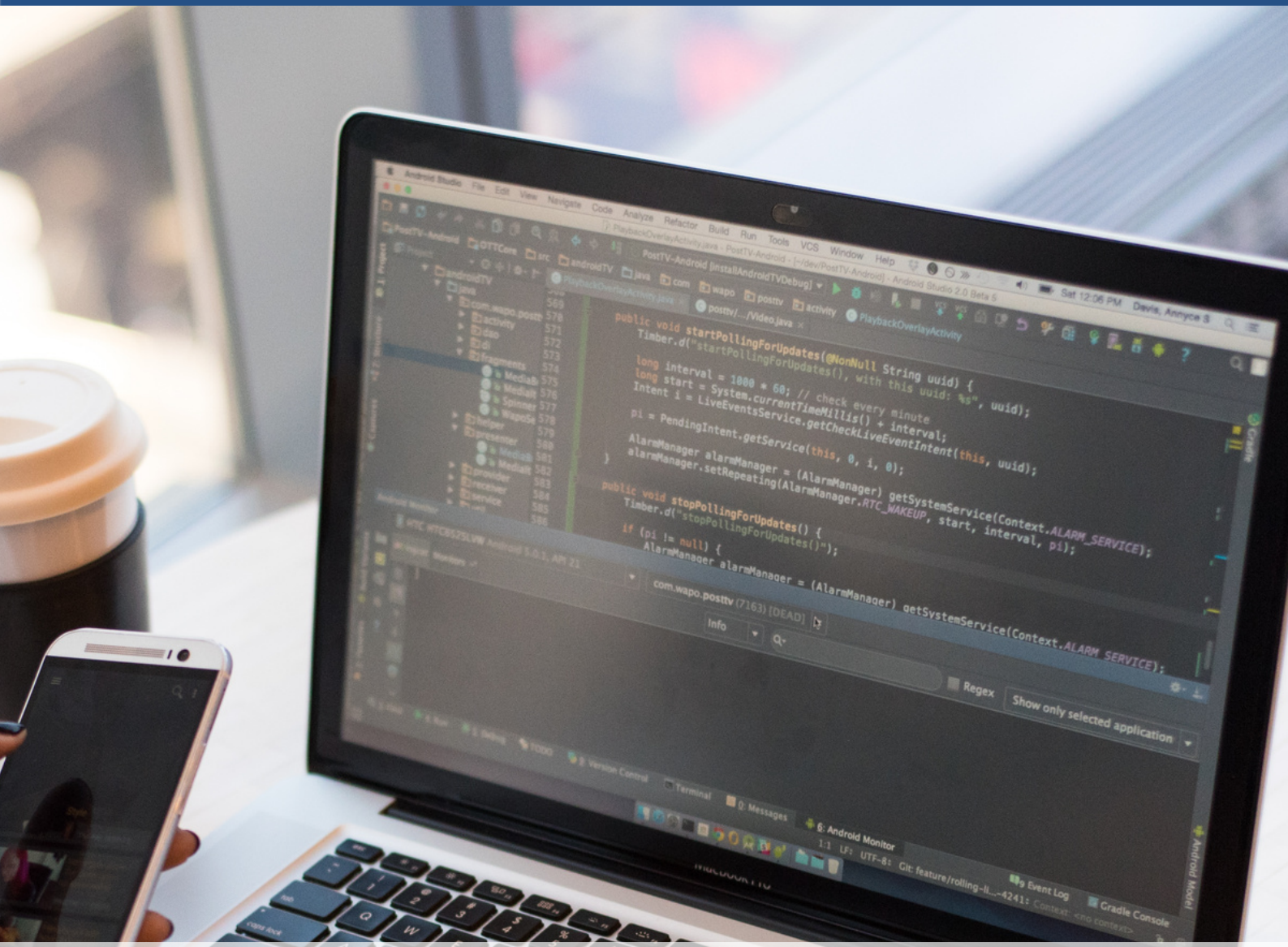


Modelos de redes neuronales en Keras

Modelos de redes neuronales en Keras

Para ver la documentación oficial de Keras:
https://keras.io/api/

Keras es una de las bibliotecas de aprendizaje profundo más populares y ampliamente utilizadas en el campo de las redes neuronales. Es una biblioteca de código abierto escrita en Python que proporciona una interfaz amigable y de alto nivel para la construcción, entrenamiento y evaluación de modelos de redes neuronales de manera rápida y eficiente.

Keras

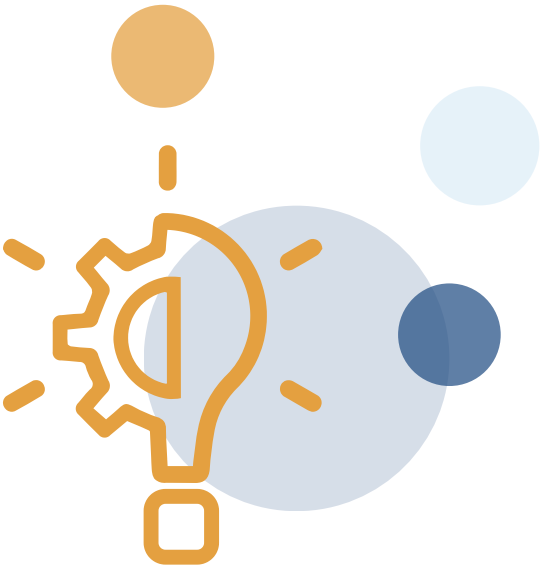
La importancia de Keras en el campo de las redes neuronales radica en varios aspectos:

Facilidad de uso

Keras ofrece una interfaz de programación simple y coherente que permite a los usuarios construir y experimentar con modelos de redes neuronales de forma intuitiva. Está diseñado para ser accesible tanto para principiantes como para expertos, lo que facilita la entrada al campo del aprendizaje profundo.

Flexibilidad

Keras es altamente flexible y permite la construcción de una amplia variedad de arquitecturas de redes neuronales, desde modelos simples hasta redes profundas y complejas. Permite la creación de modelos secuenciales (capa por capa) o modelos más complejos con múltiples entradas y salidas.



Compatibilidad con otras bibliotecas

Keras se integra perfectamente con otras bibliotecas populares de aprendizaje profundo, como TensorFlow y Theano. Esto significa que los usuarios pueden aprovechar toda la potencia de estas bibliotecas subyacentes mientras disfrutan de la simplicidad y la facilidad de uso de Keras.

Gran comunidad y soporte

Keras cuenta con una gran comunidad de usuarios y desarrolladores que contribuyen activamente con ejemplos, tutoriales, documentación y código abierto. Esto proporciona un valioso recurso para aquellos que están aprendiendo y trabajando en el campo del aprendizaje profundo.

Adopción industrial

Keras ha sido ampliamente adoptado por la industria y se utiliza en una variedad de aplicaciones y sectores, incluyendo tecnología, salud, finanzas, automoción y más. Su popularidad y versatilidad lo convierten en una opción atractiva para empresas que buscan implementar soluciones de inteligencia artificial.

Keras es una biblioteca esencial en el campo de las redes neuronales debido a su facilidad de uso, flexibilidad, compatibilidad con otras bibliotecas, gran comunidad y adopción industrial. Facilita la experimentación y el desarrollo de modelos de aprendizaje profundo, permitiendo a los usuarios centrarse en la construcción de soluciones innovadoras en lugar de preocuparse por los detalles de implementación.

El proceso para implementar una red neuronal utilizando Keras sigue varios pasos generales:

1 Preparación de los datos

Asegúrate de tener tus datos correctamente estructurados en matrices NumPy u otros formatos compatibles con Keras. Divide tus datos en conjuntos de entrenamiento, validación y prueba si es necesario.

2 Importación de la biblioteca

Importa las clases y funciones necesarias de Keras, incluyendo **Sequential** para modelos secuenciales y capas específicas como **Dense**, **Conv2D**, **LSTM**, etc., dependiendo del tipo de red neuronal que estés construyendo.

3 Configuración del modelo

Crea una instancia de un modelo secuencial (Sequential) o funcional de Keras. Añade capas al modelo, especificando el número de neuronas, la función de activación, la regularización, entre otros, según tus necesidades. Puedes hacerlo usando el método add del modelo. Puede ver la documentación oficial en https://keras.io/api/models/sequential/

Sequential class

```
keras.Sequential(layers=None, trainable=True, name=None)
```

add method

```
Sequential.add(layer, rebuild=True)
```

4 Compilación del modelo

Utiliza el método compile para compilar el modelo, especificando el optimizador, la función de pérdida y las métricas que se utilizarán para evaluar el rendimiento del modelo durante el entrenamiento. Puedes ver documentación oficial en https://keras.io/api/models/model_training_apis/

compile method

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

Donde:

optimizer:
Optimizador. Puede ser una cadena (nombre del optimizador) o una instancia de optimizador. Ver keras.optimizers.

loss:
Función de pérdida. Puede ser una cadena (nombre de la función de pérdida) o una instancia de keras.losses.Loss. Ver keras.losses.

loss_weights:
Lista opcional o diccionario que especifica coeficientes escalares (números flotantes de Python) para ponderar las contribuciones de pérdida de diferentes salidas del modelo.

metrics:
Lista de métricas que se evaluarán durante el entrenamiento y la prueba del modelo. Cada una puede ser una cadena (nombre de una función incorporada), una función o una instancia de keras.metrics.Metric. Ver keras.metrics.

weighted_metrics:
Lista de métricas que se evaluarán y ponderarán por sample_weight o class_weight durante el entrenamiento y la prueba.

run_eagerly:
Booleano. Si es True, el pase hacia adelante de este modelo nunca se compilará. Se recomienda dejar esto como False durante el entrenamiento (para obtener el mejor rendimiento) y establecerlo en True durante la depuración.

steps_per_execution:
Entero. El número de lotes que se ejecutarán durante cada llamada de función compilada.

jit_compile:

Booleano o "auto". Si es "auto", habilita la compilación XLA si el modelo lo admite, y se deshabilita de lo contrario.

5 Entrenamiento del modelo:

Utiliza el método **fit** para entrenar el modelo con los datos de entrenamiento. Especifica el número de épocas de entrenamiento y el tamaño del lote (batch size).

Ver documentación oficial https://keras.io/api/models/model_training_apis/

`fit` method

```
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,
)
```

Donde

x: Datos de entrada puede ser:

- Un array NumPy (o similar), o una lista de arrays (en caso de que el modelo tenga múltiples entradas).
- Un tensor, o una lista de tensores (en caso de que el modelo tenga múltiples entradas).
- Un diccionario que mapea nombres de entrada a los arrays/tensores correspondientes, si el modelo tiene entradas con nombres.
- Un tf.data.Dataset. Debería devolver una tupla de (entradas, objetivos) o (entradas, objetivos, pesos de muestra).
- Un keras.utils.PyDataset que devuelva (entradas, objetivos) o (entradas, objetivos, pesos de muestra).

y: Datos objetivo:

- Al igual que los datos de entrada x, puede ser arrays NumPy o tensores nativos del backend. Si x es un dataset, generador o instancia de keras.utils.PyDataset, y no debe especificarse (ya que los objetivos se obtendrán de x).

batch_size:

- Entero o None. Número de muestras por actualización de gradiente. Si no se especifica, batch_size se establecerá en 32.

epochs:

- Entero. Número de épocas para entrenar el modelo.

verbose:

- "auto", 0, 1 o 2. Modo de verbosidad.

callbacks:

- Lista de instancias de keras.callbacks.Callback. Lista de callbacks para aplicar durante el entrenamiento.

validation_split:

- Float entre 0 y 1. Fracción de los datos de entrenamiento a usar como datos de validación.

validation_data:

- Datos en los que evaluar la pérdida y cualquier métrica de modelo al final de cada época.

shuffle:

- Booleano, si mezclar los datos de entrenamiento antes de cada época.

class_weight:

- Diccionario opcional que mapea índices de clases a un valor de peso, usado para ponderar la función de pérdida durante el entrenamiento.

sample_weight:

- Array NumPy opcional de pesos para las muestras de entrenamiento, usado para ponderar la función de pérdida durante el entrenamiento.

initial_epoch:

- Entero. Época en la que comenzar el entrenamiento.

steps_per_epoch:

- Entero o None. Número total de pasos (lotes de muestras) antes de declarar una época finalizada y comenzar la siguiente.

validation_steps:

- Sólo relevante si se proporcionan datos de validación. Número total de pasos (lotes de muestras) para realizar la validación al final de cada época.

validation_batch_size:

- Entero o None. Número de muestras por lote de validación.

validation_freq:

- Solo relevante si se proporcionan datos de validación. Especifica cuantas épocas de entrenamiento correr antes de realizar una nueva validación. Por ejemplo, validation_freq=2 realiza validación cada 2 épocas.

6 Evaluación del modelo

Utiliza el método **evaluate** para evaluar el rendimiento del modelo en los datos de validación o prueba.

Ver documentación oficial https://keras.io/api/models/model_training_apis/

evaluate method

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

Donde

x: Datos de entrada puede ser:

- Un array NumPy (o similar), o una lista de arrays (en caso de que el modelo tenga múltiples entradas).
- Un tensor, o una lista de tensores (en caso de que el modelo tenga múltiples entradas).
- Un diccionario que mapea nombres de entrada a los arrays/tensores correspondientes, si el modelo tiene entradas con nombres.
- Un tf.data.Dataset. Debería devolver una tupla de (entradas, objetivos) o (entradas, objetivos, pesos de muestra).
- Un generador o keras.utils.PyDataset que devuelve (entradas, objetivos) o (entradas, objetivos, pesos de muestra).

y: Datos objetivo:

- Al igual que los datos de entrada x, puede ser arrays NumPy o tensores nativos del backend. Si x es una instancia de tf.data.Dataset o keras.utils.PyDataset, y no debe especificarse (ya que los objetivos se obtendrán del iterador/dataset).

batch_size:

- Entero o None. Número de muestras por lote de computación.
- verbose: "auto", 0, 1 o 2. Modo de verbosidad.

sample_weight:

- Array NumPy opcional de pesos para las muestras de prueba, usado para ponderar la función de pérdida.

steps:

- Entero o None. Número total de pasos (lotes de muestras) antes de declarar que la evaluación ha finalizado.

callbacks:

- Lista de instancias de keras.callbacks.Callback. Lista de callbacks para aplicar durante la evaluación.

return_dict:

- Si es True, los resultados de la pérdida y las métricas se devuelven como un diccionario, con cada clave siendo el nombre de la métrica. Si es False, se devuelven como una lista.

7 Predicciones

Utiliza el método **predict** para realizar predicciones en nuevos datos utilizando el modelo entrenado.

Ver documentación oficial https://keras.io/api/models/model_training_apis/

`predict` method

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

Donde

x: Muestras de entrada. Puede ser:

- Un array NumPy (o similar), o una lista de arrays (en caso de que el modelo tenga múltiples entradas).
- Un tensor, o una lista de tensores (en caso de que el modelo tenga múltiples entradas).
- Un tf.data.Dataset.
- Una instancia de keras.utils.PyDataset.

batch_size:

- Entero o None. Número de muestras por lote. Si no se especifica, batch_size se establecerá en 32.

verbose:

- "auto", 0, 1 o 2. Modo de verbosidad.

steps:

- Número total de pasos (lotes de muestras) antes de declarar que la predicción ha finalizado.

callbacks:

- Lista de instancias de keras.callbacks.Callback. Lista de callbacks para aplicar durante la predicción.

8 Ajuste de hiperparámetros

Opcionalmente, puedes realizar un ajuste de hiperparámetros utilizando técnicas como la búsqueda de cuadrícula (**GridSearchCV**) o la búsqueda aleatoria (**RandomizedSearchCV**) para encontrar la combinación óptima de hiperparámetros que maximice el rendimiento del modelo.

Es importante tener en cuenta que la implementación exacta puede variar según el tipo de red neuronal que estés construyendo (redes totalmente conectadas, convolucionales, recurrentes, etc.) y los requisitos específicos de tu problema. Sin embargo, estos pasos generales proporcionan una guía básica para implementar redes neuronales utilizando Keras.

Ejemplo básico de cómo implementar una red neuronal utilizando Keras en Python

En este ejemplo:

- **Importación de bibliotecas:**

```
import numpy as np
import keras
```

Importamos las bibliotecas necesarias. **keras** es una API de alto nivel para construir y entrenar modelos de redes neuronales.

- **Construcción del modelo:**

```
# Build a simple Sequential model+

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

Creamos un modelo secuencial utilizando **Sequential** de Keras, lo que significa que las capas se apilan en secuencia.

Agregamos una capa densa (**Dense**) al modelo. Esta capa tiene una sola unidad (**units=1**) y espera una entrada de una dimensión (**input_shape=[1]**).

- **Compilación del modelo:**

```
# Compile the model
model.compile(optimizer='sgd', loss='mean_squared_error')
```

Compilamos el modelo utilizando el optimizador de descenso de gradiente estocástico (**sgd**) y la función de pérdida de error cuadrático medio (**mean_squared_error**). Esto configura el modelo para la fase de entrenamiento.

- **Declaración de entradas y salidas para el entrenamiento:**

Para esta ejemplo vamos a asumir que tenemos como entradas X los números del 1 al 5 y como salidas Y vamos asumir que tenemos una función que toma cada x la multiplica por 10 y le suma 1 así:

x	y
1	11
2	21
3	31
4	41
5	51

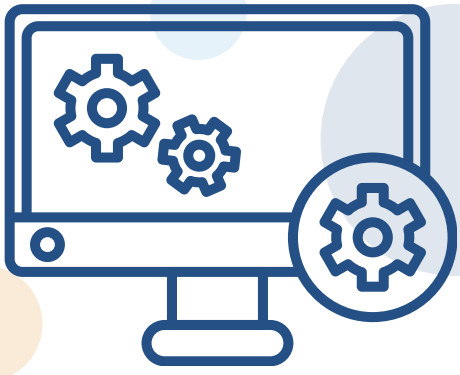
```
# Declare model inputs and outputs for training
X = np.array([1.0, 2.0, 3.0, 4.0, 5.0], dtype=float)
y = np.array([11.0,21.0,31.0,41.0,51.0], dtype=float)
```

Declaramos los datos de entrada (**x**) y los resultados deseados (**y**) para el entrenamiento del modelo.

- **Entrenamiento del modelo:**

```
# Train the model
model.fit(X, y, epochs=100)
```

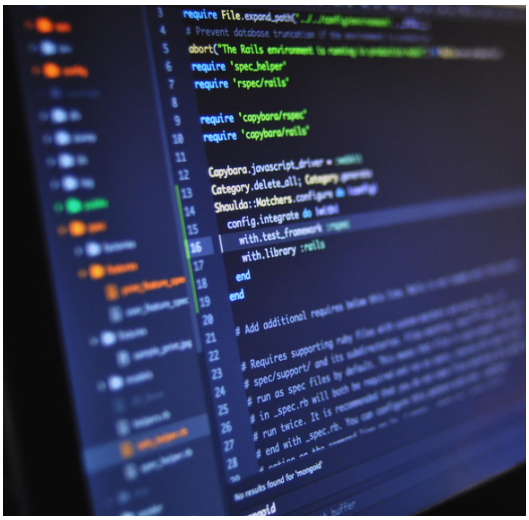
Entrenamos el modelo utilizando los datos de entrada (X) y los resultados deseados (y) durante 100 épocas. Durante cada época, el modelo ajusta sus pesos para minimizar la función de pérdida especificada en el paso de compilación.



- **Realización de una predicción:**

Para esta ejemplo vamos a asumir que tenemos como entradas X los números del 1 al 5 y como salidas Y vamos asumir que tenemos una función que toma cada x la multiplica por 10 y le suma 1 así:

```
# Make a prediction
print(model.predict([10.0]))
```



Realizamos una predicción utilizando el modelo entrenado, pasando una entrada de prueba de 10.0. El modelo utiliza los pesos ajustados durante el entrenamiento para predecir el resultado correspondiente. Este código implementa y entrena un modelo de red neuronal muy simple utilizando Keras. El modelo tiene una sola capa densa y se entrena para predecir una salida a partir de una única entrada.