HuggingFace

Hugging Face Transformers: Leverage Open-Source AI in Python – Real Python

As the AI boom continues, the Hugging Face platform stands out as the leading open-source model hub. In this tutorial, you'll get hands-on experience with Hugging Face and the Transformers library in Python.

https://realpython.com/huggingface-transformers/

Transformers de Hugging Face: cómo aprovechar la IA de código abierto en Python

Transformers es una potente biblioteca de Python creada por Hugging Face que le permite descargar, manipular y ejecutar miles de modelos de IA de código abierto previamente entrenados. Estos modelos cubren múltiples tareas en distintas modalidades, como procesamiento de lenguaje natural, visión artificial, audio y aprendizaje multimodal. El uso de modelos **de código abierto** previamente entrenados puede reducir costos, ahorrar el tiempo necesario para entrenar modelos desde cero y brindarle más control sobre los modelos que implementa.

Vamos a aprender a:

- Navegar por el ecosistema de Hugging Face
- Descargar, ejecutar y manipular modelos con Transformers
- Acelerar la inferencia de modelos con GPU

A lo largo de este tutorial, obtendrá una comprensión conceptual de las ofertas de IA de Hugging Face y aprenderá a trabajar con la biblioteca Transformers a través de ejemplos prácticos. Cuando termine, tendrá el conocimiento y las herramientas que necesita para comenzar a usar modelos para sus propios casos de uso. Antes de comenzar, será beneficioso para usted tener un conocimiento intermedio de Python y de bibliotecas de aprendizaje profundo populares como PyTorch y TensorFlow.

▼ El ecosistema de Hugging Face

Antes de usar Transformers, es conveniente que comprenda bien el ecosistema de HuggingFace. En esta primera sección, explorará brevemente todo lo que ofrece Hugging Face, con especial énfasis en las tarjetas de modelos.

▼ Exploración de Hugging Face

Hugging Face es un centro de modelos de IA de última generación. Es conocido principalmente por su amplia gama de modelos de código abierto basados en transformadores que se destacan en el procesamiento del lenguaje natural (PLN), la visión artificial y las tareas de audio. La plataforma ofrece varios recursos y servicios que atienden a desarrolladores, investigadores, empresas y cualquier persona interesada en explorar modelos de IA para sus propios casos de uso.

Hay muchas cosas que puede hacer con Hugging Face, pero las ofertas principales se pueden dividir en algunas categorías:

Modelos: Hugging Face alberga un vasto repositorio de modelos de IA previamente entrenados que son de fácil acceso y altamente personalizables. Este repositorio se llama **Model Hub** y alberga modelos que cubren una amplia gama de tareas, incluida la clasificación de texto, la

generación de texto, la traducción, el resumen, el reconocimiento de voz, la clasificación de imágenes y más. La plataforma está impulsada por la comunidad y permite a los usuarios contribuir con sus propios modelos, lo que facilita una selección diversa y en constante crecimiento.

Datasets - Conjuntos de datos: Hugging Face tiene una biblioteca de miles de conjuntos de datos que puede usar para entrenar, evaluar y mejorar sus modelos. Estos van desde evaluaciones comparativas a pequeña escala hasta conjuntos de datos masivos del mundo real que abarcan una variedad de dominios, como datos de texto, imágenes y audio. Al igual que Model Hub, 🤗 Datasets admite contribuciones de la comunidad y proporciona las herramientas que necesita para buscar, descargar y usar datos en sus proyectos de aprendizaje automático.

Espacios: *Spaces* le permite implementar y compartir aplicaciones de aprendizaje automático directamente en el sitio web de Hugging Face. Este servicio admite una variedad de marcos e interfaces, incluidos Streamlit, Gradio y cuadernos Jupyter. Es particularmente útil para mostrar las capacidades del modelo, alojar demostraciones interactivas o con fines educativos, ya que le permite interactuar con los modelos en tiempo real.

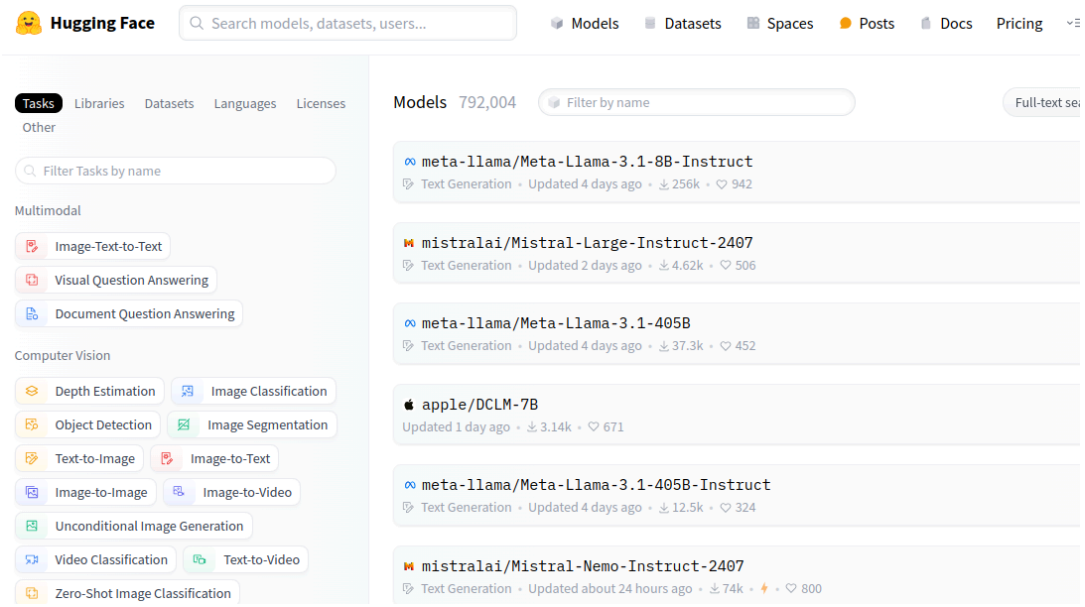
Servicios de pago: Hugging Face también ofrece varios servicios pagos para empresas y usuarios avanzados. Entre ellas se incluyen la cuenta Pro, el centro empresarial y los puntos finales de inferencia. Estas soluciones ofrecen alojamiento de modelos privados, herramientas de colaboración avanzadas y soporte dedicado para ayudar a las organizaciones a escalar sus operaciones de IA de manera eficaz.

Estos recursos le permiten acelerar sus proyectos de IA y fomentar la colaboración y la innovación dentro de la comunidad. Ya sea que sea un principiante que busca experimentar con modelos entrenados previamente o una empresa que busca soluciones de IA sólidas, Hugging Face ofrece herramientas y plataformas que satisfacen una amplia gama de necesidades.

Este tutorial se centra en Transformers, una biblioteca de Python que le permite ejecutar casi cualquier modelo en el centro de modelos. Antes de usar los transformers, deberá comprender qué son las tarjetas de modelo (Model Cards), y eso es lo que hará a continuación.

▼ Entender las Model Cards

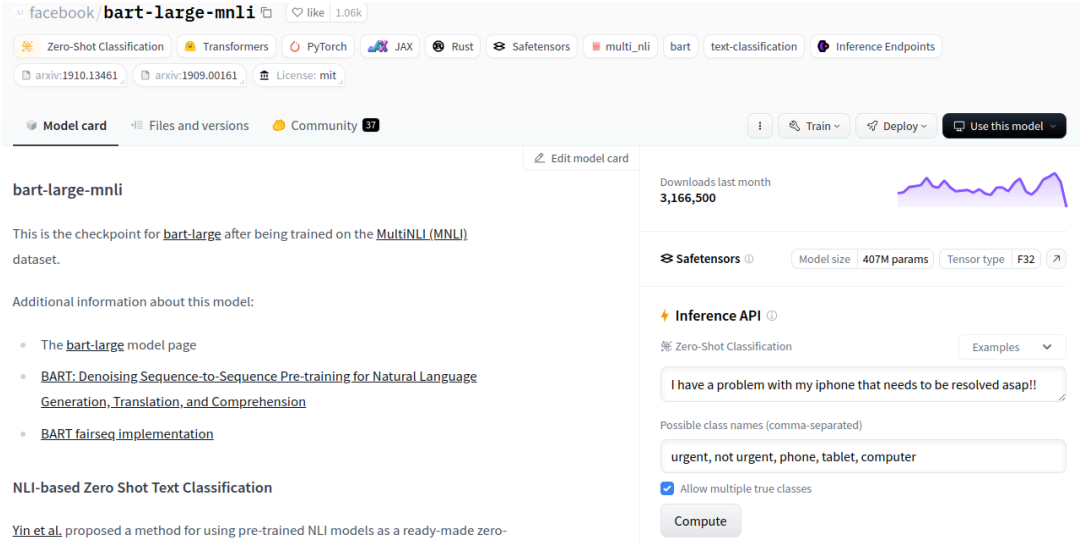
Las tarjetas de modelos son los componentes principales del Centro de modelos y necesitarás entender cómo buscarlas y leerlas para usar modelos en Transformers. Las tarjetas de modelos no son más que archivos que acompañan a cada modelo para brindar información útil. Puedes buscar la tarjeta de modelo que estás buscando en la página Modelos:



En el lado izquierdo de la página Modelos, puede buscar tarjetas de modelos según la tarea que le interese. Por ejemplo, si le interesa la clasificación de texto Zero-Shot, puede hacer clic en el botón Zero-Shot Classification Clasificación de disparo cero en la sección **Natural Language Processing**:

En esta búsqueda, puede ver 291 modelos de clasificación de texto de Zero-Shot diferentes, que es un paradigma en el que los modelos de lenguaje asignan etiquetas al texto sin entrenamiento explícito ni ver ningún ejemplo. En la esquina superior derecha, puede ordenar los resultados de la búsqueda en función de los "me gusta" del modelo, las descargas, las fechas de creación, las fechas de actualización y las tendencias de popularidad.

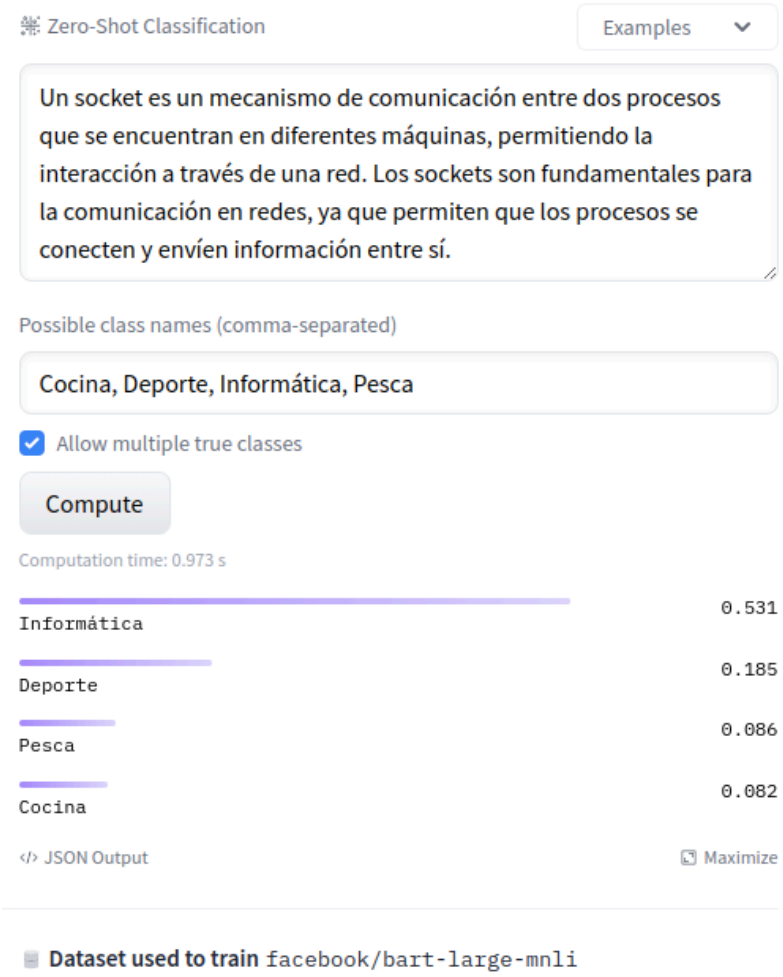
Cada botón de tarjeta de modelo le indica la tarea del modelo, cuándo se actualizó por última vez y cuántas descargas y "me gusta" tiene. Cuando hace clic en un botón de tarjeta de modelo, por ejemplo, el del modelo facebook/bart-large-mnli, la tarjeta del modelo se abrirá y mostrará toda la información del modelo:



Aunque una tarjeta de modelo puede mostrar prácticamente cualquier cosa, Hugging Face ha descrito la información que debe proporcionar una buena tarjeta de modelo. Esto incluye

información detallada sobre el modelo, sus usos y limitaciones, los parámetros de entrenamiento y los detalles del experimento, el conjunto de datos utilizado para entrenar el modelo y el rendimiento de evaluación del modelo.

Una tarjeta de modelo de alta calidad también incluye metadatos como la licencia del modelo, referencias a los datos de entrenamiento y enlaces a artículos de investigación que describen el modelo en detalle. En algunas tarjetas de modelo, también podrá modificar una instancia implementada del modelo a través de la API de inferencia. Puede ver un ejemplo de esto en la tarjeta de modelo facebook/bart-large-mnli:



Pasas un bloque de texto junto con los nombres de las clases en los que quieres categorizar el texto. Luego haces clic en Calcular y el modelo facebook/bart-large-mnli asigna una puntuación entre 0 y 1 a cada clase. Los números representan la probabilidad de que el modelo considere que el texto pertenece a la clase correspondiente. En este ejemplo, el modelo asigna puntuaciones alta a la clase Informática. Esto tiene sentido porque el texto de entrada describe un tema relacionado con la informática.

Para determinar si una tarjeta de modelo es adecuada para tu caso de uso, puedes revisar la información dentro de la tarjeta de modelo, incluidos los metadatos y las características de la API de inferencia. Estos son excelentes recursos para ayudarte a familiarizarte con el modelo y determinar su idoneidad. Y con esa introducción a Hugging Face y las tarjetas de modelo, estás listo para comenzar a ejecutar estos modelos en Transformers.

▼ La librería Transformers

La librería Transformers de Hugging Face le proporciona una API y herramientas que puede usar para descargar, ejecutar y entrenar modelos de IA de código abierto de última generación. Transformers admite la mayoría de los modelos disponibles en el centro de modelos de Hugging Face y abarca diversas tareas en el procesamiento del lenguaje natural, la visión artificial y el procesamiento de audio.

Debido a que está construido sobre PyTorch, TensorFlow y JAX, Transformers le brinda la flexibilidad de usar estos marcos para ejecutar y personalizar modelos en cualquier etapa. El uso de modelos de código abierto a través de Transformers tiene varias ventajas:

Reducción de costos: las empresas de IA propietarias como OpenAI, Cohere y Anthropic a menudo le cobran una tarifa simbólica por usar sus modelos a través de una API. Esto significa que paga por cada token que entra y sale del modelo, y sus costos de API pueden acumularse rápidamente. Al implementar su propia instancia de un modelo con Transformers, puede reducir significativamente sus costos porque solo paga por la infraestructura que aloja el modelo.

Seguridad de los datos: cuando crea aplicaciones que procesan datos confidenciales, es una buena idea mantener los datos dentro de su empresa en lugar de enviarlos a un tercero. Si bien los proveedores de IA de código cerrado suelen tener acuerdos de privacidad de datos, cada vez que los datos confidenciales salen de su ecosistema, corre el riesgo de que terminen en manos de la persona equivocada. Implementar un modelo con Transformers dentro de su empresa le brinda más control sobre la seguridad de los datos.

Ahorro de tiempo y recursos: debido a que los modelos de Transformers están entrenados previamente, no tiene que dedicar el tiempo y los recursos necesarios para entrenar un modelo de IA desde cero. Además, generalmente solo se necesitan unas pocas líneas de código para ejecutar un modelo con Transformers, lo que le ahorra el tiempo que lleva escribir el código del modelo desde cero.

En general, Transformers es un recurso fantástico que le permite ejecutar un conjunto de potentes modelos de IA de código abierto de manera eficiente. En la siguiente sección, obtendrá experiencia práctica con la biblioteca y verá lo sencillo que es ejecutar y personalizar modelos.

▼ Instalación de Transformers

Transformers está disponible en PyPI y puedes instalarlo con pip. Abre una terminal o un símbolo del sistema, crea un nuevo entorno virtual y luego ejecuta el siguiente comando:

```
(venv) $ python -m pip install transformers
```

Este comando instalará la última versión de Transformers desde PyPI en tu máquina. También aprovecharás PyTorch para interactuar con los modelos en un nivel inferior.

> Nota: La instalación de PyTorch puede llevar una cantidad considerable de tiempo. Normalmente, requiere la descarga de varios cientos de megabytes de dependencias, a menos que ya estén almacenadas en caché o incluidas con tu distribución de Python.

Puede instalar PyTorch con el siguiente comando:

```
(venv) $ python -m pip install torch
```

Para verificar que las instalaciones se hayan realizado correctamente, inicie un REPL de Python e importe transformers y Torch:

```
import transformers
import torch
```

Si las importaciones se ejecutan sin errores, entonces ha instalado correctamente las dependencias necesarias para este tutorial y está listo para comenzar con los pipelines (canalizaciones).

▼ Ejecución de pipelines

Los pipelines son la forma más sencilla de utilizar modelos listos para usar en Transformers. En particular, la función `pipeline()` le ofrece una abstracción de alto nivel sobre los modelos en el centro de modelos Hugging Face.

Para ver cómo funciona esto, suponga que desea utilizar un modelo de clasificación de sentimientos. Los modelos de clasificación de sentimientos toman texto como entrada y generan una puntuación que indica la probabilidad de que el texto tenga un sentimiento negativo, neutral o positivo. Un modelo de clasificación de sentimientos popular disponible en el centro es el modelo `cardiffnlp/twitter-roberta-base-sentiment-latest`.

Nota: Casi todos los clasificadores de aprendizaje automático generan puntuaciones que a menudo se denominan "probabilidades". Tenga en cuenta que "probabilidad" y "probabilidad" son términos matemáticos que tienen definiciones similares pero diferentes. Los resultados del clasificador no son probabilidades o probabilidades reales según estas definiciones.

Todo lo que necesita recordar es que cuanto más cerca esté una puntuación de 1, más seguro estará el modelo de que la entrada pertenece a la clase correspondiente. De manera similar, cuanto más cerca esté una puntuación de 0, más seguro estará el modelo de que la entrada no pertenece a la clase correspondiente.

Puede ejecutar este modelo con el siguiente código:

```
>>> from transformers import pipeline

>>> model_name = "cardiffnlp/twitter-roberta-base-sentiment-latest"
>>> sentiment_classifier = pipeline(model=model_name)

>>> text_input = "I'm really excited about using Hugging Face to run AI models!"
>>> sentiment_classifier(text_input)
[{'label': 'positive', 'score': 0.9870720505714417}]

>>> text_input = "I'm having a horrible day today."
>>> sentiment_classifier(text_input)
[{'label': 'negative', 'score': 0.9429882764816284}]

>>> text_input = "Most of the Earth is covered in water."
```

```
>>> sentiment_classifier(text_input)
[{'label': 'neutral', 'score': 0.7670556306838989}]
```

En este bloque, importa pipeline() y carga el modelo `cardiffnlp/twitter-roberta-base-sentiment-latest` especificando el parámetro de modelo en pipeline().

Cuando haces esto, pipeline() devuelve un objeto invocable, almacenado como `sentiment_classifier`, que puedes usar para clasificar texto. Una vez creado, `sentiment_classifier()` acepta texto como entrada y genera una etiqueta de sentimiento y una puntuación que indica la probabilidad de que el texto pertenezca a la etiqueta.

> Nota: Como verás en un momento, cada modelo que descargues puede requerir diferentes parámetros `pipeline()`. En este primer ejemplo, la única entrada requerida es el texto que quieres clasificar, pero otros modelos pueden requerir más entradas para hacer una predicción. Asegúrate de consultar la tarjeta del modelo si no estás seguro de cómo usar `pipeline()` para un modelo en particular.

Las puntuaciones del modelo van de 0 a 1. En el primer ejemplo, `sentiment_classifier` predice que el texto tiene un sentimiento positivo con alta confianza. En el segundo y tercer ejemplo, `sentiment_classifier` predice que los textos son negativos y neutrales, respectivamente.

Si desea clasificar varios textos en una llamada de función, puede pasar una lista a `sentiment_classifier`:

```
>>> text_inputs = [
...     "What a great time to be alive!",
...     "How are you doing today?",
...     "I'm in a horrible mood.",
... ]

>>> sentiment_classifier(text_inputs)
[
   {'label': 'positive', 'score': 0.98383939},
   {'label': 'neutral', 'score': 0.709688067},
   {'label': 'negative', 'score': 0.92381644}
]
```

Aquí, crea una lista de textos llamada `text_inputs` y la pasa a `sentiment_classifier()`. El modelo envuelto por `sentiment_classifier()` devuelve una etiqueta y una puntuación para cada línea de texto en el orden especificado por `text_inputs`. ¡Puede ver que el modelo ha hecho un buen trabajo al clasificar el sentimiento para cada línea de texto!

Si bien cada modelo en el centro de modelos tiene una interfaz ligeramente diferente, `pipeline()` es lo suficientemente flexible como para manejarlos todos. Por ejemplo, un paso más en complejidad con respecto a la clasificación de sentimientos es la clasificación de texto de *Zero-shot*. En lugar de clasificar el texto como positivo, neutral o negativo, los modelos de clasificación de texto de disparo cero pueden clasificar el texto en categorías arbitrarias.

A continuación, se muestra cómo puede crear una instancia de un clasificador de texto de disparo cero con `pipeline()`:

```
>>> model_name = "MoritzLaurer/deberta-v3-large-zeroshot-v2.0"
>>> zs_text_classifier = pipeline(model=model_name)

>>> candidate_labels = [
...     "Billing Issues",
...     "Technical Support",
...     "Account Information",
...     "General Inquiry",
... ]

>>> hypothesis_template = "This text is about {}"
```

En este ejemplo, primero se carga el modelo de clasificación de texto de disparo cero `MoritzLaurer/deberta-v3-large-zeroshot-v2.0` en un objeto llamado `zs_text_classifier`. Luego se definen `candidate_labels` y `hypothesis_template`, que son necesarios para que `zs_text_classifier` haga predicciones.

Los valores en `candidate_labels` indican al modelo en qué categorías se puede clasificar el texto, y `hypothesis_template` indica al modelo cómo comparar las etiquetas candidatas con el texto de entrada. En este caso, `hypothesis_template` indica al modelo que debe intentar averiguar cuál de las etiquetas candidatas es más probable que sea el texto de entrada.

Puede utilizar `zs_text_classifier` de esta manera:

```
customer_text = "My account was charged twice for a single order."
zs_text_classifier(
...     customer_text,
...     candidate_labels,
...     hypothesis_template=hypothesis_template,
...     multi_label=True
... )
```

```
{'sequence': 'My account was charged twice for a single order.',
'labels': ['Billing Issues',
   'General Inquiry',
   'Account Information',
   'Technical Support'],
'scores': [0.98844587, 0.01255007, 0.00804191, 0.00021988]}
```

Aquí, define `customer_text` y lo pasa a `zs_text_classifier` junto con `candidate_labels` y `hypothesis_template`.

Al establecer `multi_label` en True, permites que el modelo clasifique el texto en varias categorías en lugar de solo una. Esto significa que cada etiqueta puede recibir una puntuación entre 0 y 1 que es independiente de las otras etiquetas. Cuando `multi_label` es False, la suma de las puntuaciones del modelo es 1, lo que significa que el texto solo puede pertenecer a una etiqueta.

En este ejemplo, el modelo asignó una puntuación de aproximadamente 0,98 a Problemas de facturación, 0,0125 a Consulta general, 0,008 a Información de cuenta y 0,0002 a Soporte

técnico. A partir de esto, puedes ver que el modelo cree que `customer_text` probablemente se trate de Problemas de facturación, ¡y esto es cierto!

Para demostrar aún más el poder de las canalizaciones, usarás `pipeline()` para clasificar una imagen. La clasificación de imágenes es una subtarea de la visión artificial en la que un modelo predice la probabilidad de que una imagen pertenezca a una clase específica.

De manera similar al procesamiento del lenguaje natural, los clasificadores de imágenes en el centro de modelos se pueden entrenar previamente con un conjunto específico de etiquetas o se pueden entrenar para la clasificación de disparo cero.

Para utilizar los clasificadores de imágenes de Transformers, debe instalar la biblioteca de procesamiento de imágenes de Python, `Pillow`:

```
(venv) $ python -m pip install Pillow
```

Después de instalar Pillow, deberías poder crear una instancia del modelo de clasificación de imágenes predeterminado de la siguiente manera:

```
>>> image_classifier = pipeline(task="image-classification")
No model was supplied, defaulted to google/vit-base-patch16-224
and revision 5dca96d (https://huggingface.co/google/vit-base-patch16-224).
```

Observe aquí que no pasa el argumento del modelo a `pipeline()`. En su lugar, especifica **la tarea** como `image-classification` y `pipeline()` devuelve el modelo `google/vit-base-patch16-224` de forma predeterminada. Este modelo está entrenado previamente en un conjunto fijo de etiquetas, por lo que puede especificar las etiquetas como lo hace con la clasificación de disparo cero.

Ahora, suponga que desea utilizar `image_classifier` para clasificar la siguiente imagen de llamas, que puede descargar de los materiales de este tutorial:

https://realpython.com/cdn-cgi/image/width=1008,format=auto/https://files.realpython.com/media/llamas.d95420528314.png

Hay algunas formas de pasar imágenes a `image_classifier`, pero el método más sencillo es pasar la ruta de la imagen a la secuencia de comandos. Asegúrese de que la imagen *llamas.png* esté en el mismo directorio que su proceso de Python y ejecute lo siguiente:

```
>>> predictions = image_classifier(["llamas.png"])
>>> len(predictions[0])
5

>>> predictions[0][0]
{'label': 'llama',
 'score': 0.9991388320922852}

>>> predictions[0][1]
{'label': 'Arabian camel, dromedary, Camelus dromedarius',
 'score': 8.780974167166278e-05}

>>> predictions[0][2]
```

```
{'label': 'standard poodle',
 'score': 2.815701736835763e-05}
```

Aquí, pasas la ruta `llamas.png` a `image_classifier` y almacenas los resultados como predicciones. El modelo devuelve las cinco etiquetas más probables. Luego, observas la primera predicción de clase, `predictions[0][0]`, que es la clase a la que el modelo cree que es más probable que pertenezca la imagen. El modelo predice que la imagen debería etiquetarse como llama con una puntuación de aproximadamente `0,99`.

Las siguientes dos etiquetas más probables son camello árabe y caniche estándar, pero las puntuaciones para estas etiquetas son muy bajas. ¡Es bastante sorprendente la confianza que tiene el modelo para predecir llama en una imagen que nunca ha visto antes!

La conclusión más importante es lo sencillo que es usar modelos de forma inmediata con `pipeline()`. Todo lo que haces es pasar entradas sin procesar como texto o imágenes a las canalizaciones, junto con la cantidad mínima de entrada adicional que el modelo necesita para ejecutarse, como la plantilla de hipótesis o las etiquetas de los candidatos. La canalización se encarga del resto por ti.

Si bien las canalizaciones son excelentes para comenzar con los modelos, es posible que necesites más control sobre los detalles internos de un modelo. En la siguiente sección, aprenderá cómo dividir las canalizaciones en sus componentes individuales con clases automáticas.

▼ Mirando bajo el capó con clases automáticas

Como ha visto hasta ahora, las canalizaciones facilitan el uso de modelos listos para usar. Sin embargo, es posible que desee personalizar aún más los modelos a través de técnicas como el **ajuste fino**. El ajuste fino (fine tuning) es una técnica que adapta un modelo previamente entrenado a una tarea específica con datos potencialmente diferentes pero relacionados. Por ejemplo, puede tomar un clasificador de imágenes existente en Model Hub y entrenarlo aún más para clasificar imágenes que son propiedad de su empresa.

Para tareas de personalización como el ajuste fino, Transformers le permite acceder a los componentes de nivel inferior que forman las canalizaciones a través de **clases automáticas**. Esta sección no repasará el ajuste fino u otras personalizaciones específicamente, pero obtendrá una comprensión más profunda de cómo funcionan las canalizaciones bajo el capó al observar sus clases automáticas.

Suponga que desea un acceso y una comprensión más granulares de la canalización del clasificador de sentimientos `cardiffnlp/twitter-roberta-base-sentiment-latest` que vio en la sección anterior. El primer componente de esta canalización, y casi todas las canalizaciones de NLP (Natural Language Processing), es el **tokenizador**.

Los tokens pueden ser palabras, subpalabras o incluso caracteres, según el diseño del tokenizador. Un tokenizador es un componente que procesa el texto de entrada y lo convierte a un formato que el modelo puede entender. Para ello, divide el texto en tokens y asocia esos tokens con un ID.

Puede acceder a los tokenizadores mediante la clase AutoTokenizer. Para ver cómo funciona, observe este ejemplo:

```
from transformers import AutoTokenizer
model_name = "cardiffnlp/twitter-roberta-base-sentiment-latest"
```

```
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
input_text = "I really want to go to an island. Do you want to go?"
encoded_input = tokenizer(input_text)
encoded_input["input_ids"]
[0, 100, 269, 236, 7, 213, 7, 41, 2946, 4, 1832, 47, 236, 7, 213, 116, 2]
```

En este bloque, primero importas la clase `AutoTokenizer` de `transformers`. Luego, creas una instancia y almacenas el tokenizador para el modelo `cardiffnlp/twitter-roberta-base-sentiment-latest` usando el método de clase `.from_pretrained()`. Por último, pasas un `input_text` al tokenizador y miras los ID que asocia con cada token.

Cada entero en `input_ids` es el ID de un token dentro del vocabulario del tokenizador. Por ejemplo, ya puedes saber que el ID 7 corresponde al token “to” porque se repite varias veces. Esto puede parecer un poco críptico al principio, pero podemos entenderlo mejor usando `.convert_ids_to_tokens()` para convertir los ID nuevamente en tokens:

```
tokenizer.convert_ids_to_tokens(7)
# → 'Ġto'

>>> tokenizer.convert_ids_to_tokens(2946)
# → 'Ġisland'

>>> tokenizer.convert_ids_to_tokens(encoded_input["input_ids"])
# → ['<s>','I','Ġreally','Ġwant','Ġto','Ġgo','Ġto','Ġan','Ġisland','.','ĠDo', 'Ġyou', 'Ġwant', 'Ġto', 'Ġgo',
```

Con `.convert_ids_to_tokens()`, vemos que el ID 7 y el ID 2946 se convierten en tokens “to” e “island”, respectivamente. El prefijo Ġ es un símbolo especial que se utiliza para indicar el comienzo de una nueva palabra en contextos en los que se utilizan espacios en blanco como separador. Al pasar `encoded_input["input_ids"]` a `.convert_ids_to_tokens()`, se recupera la entrada de texto original con los tokens adicionales <s> y </s>, que indican el comienzo y el final del texto.

Puede ver cuántos tokens hay en el vocabulario del tokenizador observando el atributo `vocab_size`:

```
tokenizer.vocab_size
# → 50265
```

Este tokenizador en particular tiene 50 265 tokens. Si quisieras ajustar este modelo y hubiera nuevos tokens en tus datos de entrenamiento, tendrías que agregarlos al tokenizador con `.add_tokens()`:

```
new_tokens = ["whaleshark", "unicorn",]

tokenizer.convert_tokens_to_ids(new_tokens)
# → [3, 3]

tokenizer.convert_ids_to_tokens(3)
# → '<unk>'
```

```
tokenizer.add_tokens(new_tokens)
# → 2

tokenizer.convert_tokens_to_ids(new_tokens)
# → [50265, 50266]
```

Primero define una lista llamada `new_tokens`, que tiene dos tokens que no están en el vocabulario del tokenizador de forma predeterminada. Cuando llama a `.convert_tokens_to_ids()`, ambos tokens nuevos se asignan al ID 3. Corresponde a <unk>, que es el token predeterminado para el las palabras que no están en el vocabulario de entrada. Cuando pasas `new_tokens` a `.add_tokens()`, los tokens se agregan al vocabulario y se les asignan nuevos ID de 50265 y 50266.

También puede utilizar clases automáticas para acceder al objeto modelo. Para el modelo `cardiffnlp/twitter-roberta-base-sentiment-latest`, puede cargar el objeto del modelo directamente usando `AutoModelForSequenceClassification`:

```
import torch
from transformers import (AutoTokenizer, AutoModelForSequenceClassification)

model_name = "cardiffnlp/twitter-roberta-base-sentiment-latest"

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)
print(model)
```

```
RobertaForSequenceClassification(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(50265, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0-11): 12 x RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
```

```
        )
        (intermediate): RobertaIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
          (intermediate_act_fn): GELUActivation()
        )
        (output): RobertaOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
  )
  (classifier): RobertaClassificationHead(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
    (out_proj): Linear(in_features=768, out_features=3, bias=True)
  )
)
```

Aquí, agrega `AutoModelForSequenceClassification` a tus importaciones y crea una instancia del objeto de modelo para `cardiffnlp/twitter-roberta-base-sentiment-latest` usando `.from_pretrained()`. Cuando llamas a `model` en la consola, puedes ver la representación de cadena completa del modelo. El modelo de Roberta consta de una serie de capas a las que puedes acceder y modificar directamente.

Como ejemplo, observa la capa de embeddings (incrustaciones):

```
model.roberta.embeddings
```

```
RobertaEmbeddings(
  (word_embeddings): Embedding(50265, 768, padding_idx=1)
  (position_embeddings): Embedding(514, 768, padding_idx=1)
  (token_type_embeddings): Embedding(1, 768)
  (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (dropout): Dropout(p=0.1, inplace=False)
)
```

No es el objetivo de este tutorial analizar todas las complejidades del modelo de *Roberta*, pero preste mucha atención a la capa `word_embeddings`. Es posible que haya notado que la primera entrada a `Embedding()` en la capa `word_embeddings` es 50265, el tamaño exacto del vocabulario del tokenizador.

Esto se debe a que la primera capa de incrustación asigna cada token del vocabulario a un tensor de `PyTorch` de tamaño 768. En otras palabras, `Embedding(50265, 768)` asigna los 50265 tokens del vocabulario a un tensor de PyTorch con 768 elementos. Para comprender mejor cómo funciona esto, puede convertir el texto de entrada en incrustaciones directamente utilizando la capa de incrustaciones:

```
text = "I love using the Transformers library!"
encoded_input = tokenizer(text, return_tensors="pt")

embedding_tensor = model.roberta.embeddings(encoded_input["input_ids"])
embedding_tensor.shape
# → torch.Size([1, 9, 768])

embedding_tensor
```

```
tensor([[[ 0.0633, -0.0212,  0.0193,  ..., -0.0826, -0.0200, -0.0056],
         [ 0.1453,  0.3706, -0.0322,  ...,  0.0359, -0.0750,  0.0376],
         [ 0.2900, -0.0814,  0.0955,  ...,  0.3262, -0.0559,  0.0819],
         ...,
         [ 0.1059, -0.5638, -0.2397,  ..., -0.2077, -0.0784, -0.0951],
         [ 0.1675, -0.3334,  0.0130,  ..., -0.4127,  0.0121,  0.0215],
         [ 0.1316, -0.0281, -0.0168,  ...,  0.1175,  0.0908, -0.0614]]],
       grad_fn=<NativeLayerNormBackward0>)
```

En este bloque, defines el texto y conviertes cada token en su ID correspondiente usando `tokenizer()`. Luego pasas los ID de token a la capa de incrustaciones de Roberta y almacenas los resultados como `embedding_tensor`. Observa cómo el tamaño de embedding_tensor es [1, 9, 768]. Esto se debe a que pasaste una entrada de texto a la capa de incrustación que tenía nueve tokens, y cada token se convirtió en un tensor con 768 elementos.

Cuando miras la representación de cadena `embedding_tensor`, la primera fila es la incrustación para el token <s>, la segunda fila es para el token *I*, la tercera para el token *love*, y así sucesivamente. Si quisieras ajustar el modelo de Roberta con nuevos tokens, primero agregarías los nuevos tokens al tokenizer como lo hiciste anteriormente, y luego tendrías que actualizar y entrenar la capa de incrustaciones con un tensor de 768 elementos para cada nuevo token.

En el modelo completo, el tensor de incrustación pasa por varias capas donde se le cambia la forma, se lo manipula y, finalmente, se lo convierte en una puntuación prevista para cada clase de sentimiento.

Puedes unir estas clases automáticas para crear el flujo de trabajo completo:

```
>>> import torch
>>> from transformers import (
...     AutoTokenizer,
...     AutoModelForSequenceClassification,
...     AutoConfig
... )

>>> model_name = "cardiffnlp/twitter-roberta-base-sentiment-latest"

>>> config = AutoConfig.from_pretrained(model_name)
>>> tokenizer = AutoTokenizer.from_pretrained(model_name)
>>> model = AutoModelForSequenceClassification.from_pretrained(model_name)
```

```
>>> text = "I love using the Transformers library!"
>>> encoded_input = tokenizer(text, return_tensors="pt")

>>> with torch.no_grad():
...     output = model(**encoded_input)
...
>>> scores = output.logits[0]
>>> probabilities = torch.softmax(scores, dim=0)

>>> for i, probability in enumerate(probabilities):
...     label = config.id2label[i]
...     print(f"{i+1}) {label}: {probability}")
...
1) negative: 0.0026470276061445475
2) neutral: 0.010737836360931396
3) positive: 0.9866151213645935
```

Aquí, primero importamos `torch` junto con las clases automáticas que viste anteriormente. Además, importas `AutoConfig`, que tiene la configuración y los metadatos para el modelo. Luego, almacenas el nombre de la canalización en `model_name` y creas una instancia de los objetos de configuración, tokenizador y modelo.

A continuación, defines el texto y lo tokenizas con `tokenizer()`. Luego, pasas la entrada tokenizada, `encoded_input`, al objeto de modelo y almacenas los resultados como `output`. Utilizas el administrador de contextos de `torch.no_grad()` para acelerar la inferencia del modelo al deshabilitar los cálculos de gradiente.

Después de eso, conviertes la salida del modelo sin procesar en puntajes y luego transformas los puntajes para que sumen 1 usando `torch.softmax()`. Por último, recorres cada elemento en probabilidades y generas el valor junto con la etiqueta asociada, que proviene de `config.id2label`. Los resultados te indican que el modelo asigna una probabilidad predicha de aproximadamente 0,9866 a la clase positiva para el texto de entrada.

Puede verificar que este código da los mismos resultados que el pipeline `cardiffnlp/twitter-roberta-base-sentiment-latest` que utilizó en el ejemplo anterior:

Ahora comprende cómo funcionan las canalizaciones en profundidad y cómo puede acceder y manipular los componentes de la canalización con clases automáticas. Esto le brinda las herramientas para crear canalizaciones personalizadas a través de técnicas como el ajuste fino y una comprensión más profunda del modelo subyacente.

En la siguiente sección, cambiará de tema y aprenderá a mejorar el rendimiento de las canalizaciones aprovechando las GPU.

▼ El poder de las GPU

Casi todos los modelos enviados a Hugging Face son redes neuronales y, más específicamente, transformers. Estas redes neuronales comprenden múltiples capas con millones, miles de millones y, a veces, incluso billones de parámetros. Por ejemplo, el modelo `MoritzLaurer/deberta-v3-large-zeroshot-v2.0` que utilizó en la primera sección tiene 435 millones de parámetros, y se trata de un modelo de lenguaje relativamente pequeño.

El cálculo central de cualquier red neuronal es la multiplicación de matrices, y realizar la multiplicación de matrices sobre varios millones de parámetros puede ser costoso desde el

punto de vista computacional. Debido a esto, el entrenamiento y la inferencia para la mayoría de las redes neuronales grandes se realiza en unidades de procesamiento gráfico (GPU).

Las GPU son hardware especializado que puede acelerar significativamente el tiempo de entrenamiento e inferencia de las redes neuronales en comparación con la CPU. Esto se debe a que las GPU tienen miles de núcleos pequeños y eficientes diseñados para procesar múltiples tareas simultáneamente, mientras que las CPU tienen menos núcleos optimizados para el procesamiento serial secuencial. Esto hace que las GPU sean especialmente potentes para las operaciones matriciales y vectoriales requeridas en las redes neuronales.

Si bien todos los modelos disponibles en Transformers pueden ejecutarse en CPU, como los que viste en la sección anterior, es probable que la mayoría de ellos hayan sido entrenados y estén optimizados para ejecutarse en GPU. En la siguiente sección, verás lo fácil que es ejecutar pipelines en GPU con Transformers. Esto mejorará drásticamente el rendimiento de tus pipelines y te permitirá hacer predicciones a la velocidad del rayo.

▼ Configuración de un cuaderno de Google Colab

Es posible que no tengas acceso a una GPU en tu máquina local, por lo que en esta sección, usarás Google Colab, una interfaz de cuaderno de Python que ofrece Google para ejecutar pipelines en GPU de forma gratuita. Para comenzar, inicia sesión en Google Colab y crea un nuevo cuaderno. Una vez creado, tu cuaderno debería verse así:

Elegiremos T4 GPU

A continuación, deberá cargar los archivos `requirements.txt` y `Scraped_Car_Review_dodge.csv` de los materiales de este tutorial en su sesión de Notebook. Para ello, haga clic derecho en la pestaña de la carpeta y seleccione Cargar:

Una vez cargados, deberías ver los dos archivos en la pestaña de la carpeta:

```
!pip install -r /content/requirements.txt
```

Presiona Shift+Enter y tus requisitos deberían instalarse. Aunque Google Colab almacena en caché los paquetes populares para acelerar las instalaciones posteriores, es posible que debas esperar unos minutos hasta que finalice la instalación. Una vez que se complete, estarás listo para comenzar a ejecutar pipelines en la GPU.

▼ Ejecución de pipelines en GPU

Ahora que tienes una notebook en ejecución con acceso a una GPU, Transformers te facilita la ejecución de pipelines en la GPU. En esta sección, ejecutarás reseñas de autos desde el archivo Scraped_Car_Review_dodge.csv a través de un pipeline de clasificación de sentimientos tanto en la CPU como en la GPU, y verás cuánto más rápido se ejecuta el pipeline en la GPU. Si estás interesado, puedes leer más sobre el conjunto de datos de reseñas de autos en Kaggle.

Para comenzar, define la ruta a los datos e importa tus dependencias en una nueva celda:

```
DATA_PATH = "/content/Scraped_Car_Review_dodge.csv"

import time
from tqdm import tqdm
import polars as pl
```

```
import torch
from transformers import pipeline, TextClassificationPipeline,
```

No te preocupes si no estás familiarizado con algunas de estas dependencias: verás cómo se utilizan en un momento. A continuación, puedes usar la biblioteca Polars para leer las reseñas de los vehículos y almacenarlas en una lista:

```
reviews_list = pl.read_csv(DATA_PATH)["Review"].to_list()
len(reviews_list) # Out[3]: 8499
```

Aquí, lee la columna de reseñas de `Scraped_Car_Review_dodge.csv` y guárdala en `reviews_list`. Luego, observa la longitud de `reviews_list` y ve que hay 8499 reseñas. Esto es lo que dice la primera reseña:

```
In [4]: reviews_list[0]
Out[4]: " It's been a great delivery vehicle for my
cafe business good power, economy match easily taken
care of. Havent repaired anything or replaced anything
but tires and normal maintenance items. Upgraded tires
to Michelin LX series helped fuel economy. Would buy
another in a second"
```

Usarás el clasificador de opiniones `cardiffnlp/twitter-roberta-base-sentiment-latest` para predecir las opiniones de estas reseñas usando tanto una CPU como una GPU, y medirás el tiempo que tardan ambas. Para ayudarte con estos experimentos, define la siguiente función auxiliar:

```
def time_text_classifier(
    text_pipeline: TextClassificationPipeline,
    texts: list[str],
    batch_size: int = 1) -> None:
    """Time how long it takes a TextClassificationPipeline
       to run inference on a list of texts"""

    texts_generator = (t for t in texts)
    pipeline_iterable = tqdm(
        text_pipeline(
            texts_generator,
            batch_size=batch_size,
            truncation=True,
            max_length=500,
        ),
        total=len(texts),
    )

    for result in pipeline_iterable:
        pass
```

El objetivo de `time_text_classifier()` es evaluar cuánto tiempo tarda un `TextClassificationPipeline` en hacer predicciones sobre una lista de textos.

Primero convierte los textos en un generador llamado `text_generator` y pasa el generador al pipeline de clasificación de texto. Esto convierte el pipeline en un iterable que se puede recorrer en un bucle for para obtener predicciones. El batch_size determina cuántas predicciones hace el modelo en una pasada.

Envuelves el iterador del pipeline de clasificación de texto con `tqdm()` para ver el progreso de tu pipeline a medida que hace predicciones. Por último, iteras sobre `pipeline_iterable` y usas la declaración pass, ya que solo te interesa cuánto tiempo tarda en ejecutarse. Una vez que se realizan todas las predicciones, `tqdm()` muestra el tiempo de ejecución total.

A continuación, crearás una instancia de dos pipelines, una que se ejecuta en la CPU y la otra en la GPU:

```
model_name = "cardiffnlp/twitter-roberta-base-sentiment-latest"
sentiment_pipeline_cpu = pipeline(model=model_name, device=-1)
sentiment_pipeline_gpu = pipeline(model=model_name, device=0)
```

Transformers le permite especificar fácilmente en qué hardware se ejecuta su pipeline con el argumento `device`. Cuando pasa un entero al argumento device, le está indicando al pipeline en qué GPU debe ejecutarse. Cuando `device` es -1, el pipeline se ejecuta en la CPU y cualquier número de dispositivo no negativo le indica al pipeline qué GPU usar según su rango ordinal.

En este caso, es probable que solo tenga una GPU, por lo que configurar `device=0` le indica al pipeline que se ejecute en la primera y única GPU. Si tuviera una segunda GPU, podría configurar `device=1`, y así sucesivamente. Ahora está listo para cronometrar estos dos pipelines comenzando con la CPU:

```
time_text_classifier(sentiment_pipeline_cpu, reviews_list[0:1000])
# Out[7]: 100%|██████████| 1000/1000 [05:49<00:00,  2.86it/s]
```

Aquí, cronometras el tiempo que tarda `sentiment_pipeline_cpu` en hacer predicciones sobre las primeras 1000 reseñas. Los resultados muestran que tardó unos 5 minutos y 49 segundos. Esto significa que sentiment_pipeline_cpu hizo aproximadamente 2,86 predicciones por segundo.

Ahora puedes ejecutar el mismo experimento para `sentiment_pipeline_gpu`:

```
time_text_classifier(sentiment_pipeline_gpu, reviews_list[0:1000])
# Out[8]: 100%|██████████| 1000/1000 [00:16<00:00, 60.06it/s]
```

En las mismas 1000 revisiones, `sentiment_pipeline_gpu` tardó unos 16 segundos en hacer todas las predicciones. Eso es casi 61 predicciones por segundo y aproximadamente 21 veces más rápido que `sentiment_pipeline_cpu`. Tenga en cuenta que los tiempos de ejecución exactos variarán, pero puede ejecutar este experimento varias veces para medir el tiempo promedio.

Puede optimizar aún más el rendimiento del pipeline experimentando con `batch_size`, que es un parámetro que determina cuántas entradas procesa el modelo a la vez. Por ejemplo, si `batch_size` es 4, entonces el modelo hará predicciones de cuatro entradas simultáneamente. Vea el rendimiento de `cardiffnlp/twitter-roberta-base-sentiment-latest` en diferentes tamaños de lote:

```
In [9]: batch_sizes = [1, 2, 4, 8, 10, 12, 15, 20, 50, 100]
   ...: for batch_size in batch_sizes:
```

```
...:     print(f"Batch size: {batch_size}")
...:     time_text_classifier(
...:         sentiment_pipeline_gpu,
...:         reviews_list,
...:         batch_size=batch_size
...:     )
...:
Out[9]:
Batch size: 1
100%|██████████| 8499/8499 [01:50<00:00, 77.02it/s]
Batch size: 2
100%|██████████| 8499/8499 [01:33<00:00, 91.19it/s]
Batch size: 4
100%|██████████| 8499/8499 [01:39<00:00, 85.84it/s]
Batch size: 8
100%|██████████| 8499/8499 [01:47<00:00, 78.76it/s]
Batch size: 10
100%|██████████| 8499/8499 [01:51<00:00, 75.99it/s]
Batch size: 12
100%|██████████| 8499/8499 [01:56<00:00, 73.19it/s]
Batch size: 15
100%|██████████| 8499/8499 [02:01<00:00, 70.16it/s]
Batch size: 20
100%|██████████| 8499/8499 [02:04<00:00, 68.32it/s]
Batch size: 50
100%|██████████| 8499/8499 [02:37<00:00, 53.91it/s]
Batch size: 100
100%|██████████| 8499/8499 [03:08<00:00, 45.15it/s]
```

Aquí, itera sobre una lista de tamaños de lotes y ve cuánto tiempo le toma al pipeline ejecutar las 8499 revisiones en el tamaño de lote correspondiente. A partir de la salida de `tqdm`, puede ver que un tamaño de lote de 2 dio como resultado el mejor rendimiento con aproximadamente 91 predicciones por segundo.

En general, decidir un tamaño de lote óptimo para la inferencia requiere experimentación. Debe ejecutar experimentos como este varias veces en diferentes conjuntos de datos para ver qué tamaño de lote es mejor en promedio.

¡Ahora sabe cómo ejecutar y evaluar pipelines en GPU! Tenga en cuenta que, si bien la mayoría de los modelos disponibles en Transformers funcionan mejor en GPU, no siempre es posible hacerlo. En la práctica, las GPU pueden ser costosas y consumir muchos recursos, por lo que debe decidir si la ganancia de rendimiento es necesaria y vale la pena el costo para su aplicación. Experimente siempre y asegúrese de tener un conocimiento sólido del hardware que desea utilizar.

▼ Conclusión

La biblioteca Transformers de Hugging Face es una herramienta completa y fácil de usar que le permite ejecutar modelos de IA de código abierto en Python. Ha tenido una descripción general amplia de Hugging Face y la biblioteca Transformers, y ahora tiene el conocimiento y los recursos necesarios para comenzar a usar Transformers en sus propios proyectos.

Transformers está bien posicionado para adaptarse al panorama de IA en constante cambio, ya que admite una variedad de modalidades y tareas diferentes. ¿Cómo usará Transformers en su próximo proyecto de IA?