Introduction to Keras for engineers

Author: fchollet
Date created: 2023/07/10
Last modified: 2023/07/10
Description: First contact with Keras 3.

Introduction

Keras 3 is a deep learning framework works with TensorFlow, JAX, and PyTorch interchangeably. This notebook will walk you through key Keras 3 workflows.

Let's start by installing Keras 3:

```
!pip install keras==3.0.0 --upgrade --quiet
```

Setup

We're going to be using the JAX backend here -- but you can edit the string below to `"tensorflow"` or `"torch"` and hit "Restart runtime", and the whole notebook will run just the same! This entire guide is backend-agnostic.

```
import numpy as np
import os

os.environ["KERAS_BACKEND"] = "jax"

# Note that Keras should only be imported after the backend
# has been configured. The backend cannot be changed once the
# package is imported.
import keras
```

A first example: A MNIST convnet

Let's start with the Hello World of ML: training a convnet to classify MNIST digits.

Here's the data:

```
# Load the data and split it between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")
```

Here's our model.

Different model-building options that Keras offers include:

- The Sequential API (what we use below)
- The Functional API (most typical)
- Writing your own models yourself via subclassing (for advanced use cases)

```
# Model parameters
num_classes = 10
input_shape = (28, 28, 1)

model = keras.Sequential(
    [
        keras.layers.Input(shape=input_shape),
        keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        keras.layers.MaxPooling2D(pool_size=(2, 2)),
        keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
        keras.layers.GlobalAveragePooling2D(),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(num_classes, activation="softmax"),
    ]
)
```

Here's our model summary:

```
model.summary()
```

We use the `compile()` method to specify the optimizer, loss function, and the metrics to monitor. Note that with the JAX and TensorFlow backends, XLA compilation is turned on by default.

```
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name="acc"),
    ],
)
```

Let's train and evaluate the model. We'll set aside a validation split of 15% of the data during training to monitor generalization on unseen data.

```
batch_size = 128
epochs = 20

callbacks = [
    keras.callbacks.ModelCheckpoint(filepath="model_at_epoch_{epoch}.keras"),
    keras.callbacks.EarlyStopping(monitor="val_loss", patience=2),
]

model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.15,
    callbacks=callbacks,
)
score = model.evaluate(x_test, y_test, verbose=0)
```

During training, we were saving a model at the end of each epoch. You can also save the model in its latest state like this:

```
model.save("final_model.keras")
```

And reload it like this:

```
model = keras.saving.load_model("final_model.keras")
```

Next, you can query predictions of class probabilities with `predict()`:

```
predictions = model.predict(x_test)
```

That's it for the basics!

Writing cross-framework custom components

Keras enables you to write custom Layers, Models, Metrics, Losses, and Optimizers that work across TensorFlow, JAX, and PyTorch with the same codebase. Let's take a look at custom layers first.

The `keras.ops` namespace contains:

- An implementation of the NumPy API, e.g. `keras.ops.stack` or `keras.ops.matmul`.
- A set of neural network specific ops that are absent from NumPy, such as `keras.ops.conv` or `keras.ops.binary_crossentropy`.

Let's make a custom `Dense` layer that works with all backends:

```

class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, name=None):
        super().__init__(name=name)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.w = self.add_weight(
            shape=(input_dim, self.units),
            initializer=keras.initializers.GlorotNormal(),
            name="kernel",
            trainable=True,
        )

        self.b = self.add_weight(
            shape=(self.units,),
            initializer=keras.initializers.Zeros(),
            name="bias",
            trainable=True,
        )

    def call(self, inputs):
        # Use Keras ops to create backend-agnostic layers/metrics/etc.
        x = keras.ops.matmul(inputs, self.w) + self.b
        return self.activation(x)

```

Next, let's make a custom `Dropout` layer that relies on the `keras.random` namespace:

```python

class MyDropout(keras.layers.Layer):
    def __init__(self, rate, name=None):
        super().__init__(name=name)
        self.rate = rate
        # Use seed_generator for managing RNG state.
        # It is a state element and its seed variable is
        # tracked as part of `layer.variables`.
        self.seed_generator = keras.random.SeedGenerator(1337)

    def call(self, inputs):
        # Use `keras.random` for random ops.
        return keras.random.dropout(inputs, self.rate, seed=self.seed_generator)

```

Next, let's write a custom subclassed model that uses our two custom layers:

```python

class MyModel(keras.Model):
    def __init__(self, num_classes):
        super().__init__()
        self.conv_base = keras.Sequential(
            [
                keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
                keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
                keras.layers.MaxPooling2D(pool_size=(2, 2)),
                keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
                keras.layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),
                keras.layers.GlobalAveragePooling2D(),
            ]
        )
        self.dp = MyDropout(0.5)
        self.dense = MyDense(num_classes, activation="softmax")

    def call(self, x):
        x = self.conv_base(x)
        x = self.dp(x)
        return self.dense(x)

```

Let's compile it and fit it:

```
model = MyModel(num_classes=10)
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name="acc"),
    ],
)

model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=1,  # For speed
    validation_split=0.15,
)
```

Training models on arbitrary data sources

All Keras models can be trained and evaluated on a wide variety of data sources, independently of the backend you're using. This includes:

- NumPy arrays
- Pandas dataframes
- TensorFlow `tf.data.Dataset` objects
- PyTorch `DataLoader` objects
- Keras `PyDataset` objects

They all work whether you're using TensorFlow, JAX, or PyTorch as your Keras backend.

Let's try it out with PyTorch `DataLoaders` :

```
import torch

# Create a TensorDataset
train_torch_dataset = torch.utils.data.TensorDataset(
    torch.from_numpy(x_train), torch.from_numpy(y_train)
)
val_torch_dataset = torch.utils.data.TensorDataset(
    torch.from_numpy(x_test), torch.from_numpy(y_test)
)

# Create a DataLoader
train_dataloader = torch.utils.data.DataLoader(
    train_torch_dataset, batch_size=batch_size, shuffle=True
)
val_dataloader = torch.utils.data.DataLoader(
    val_torch_dataset, batch_size=batch_size, shuffle=False
)

model = MyModel(num_classes=10)
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name="acc"),
    ],
)
model.fit(train_dataloader, epochs=1, validation_data=val_dataloader)

```

Now let's try this out with `tf.data`:

```
import tensorflow as tf

train_dataset = (
    tf.data.Dataset.from_tensor_slices((x_train, y_train))
    .batch(batch_size)
    .prefetch(tf.data.AUTOTUNE)
)
test_dataset = (
    tf.data.Dataset.from_tensor_slices((x_test, y_test))
    .batch(batch_size)
    .prefetch(tf.data.AUTOTUNE)
)

model = MyModel(num_classes=10)
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name="acc"),
    ],
)
model.fit(train_dataset, epochs=1, validation_data=test_dataset)
```

Further reading

This concludes our short overview of the new multi-backend capabilities of Keras 3. Next, you can learn about:

How to customize what happens in `fit()`

Want to implement a non-standard training algorithm yourself but still want to benefit from the power and usability of `fit()`? It's easy to customize `fit()` to support arbitrary use cases:

- Customizing what happens in `fit()` with TensorFlow
- Customizing what happens in `fit()` with JAX
- Customizing what happens in `fit()` with PyTorch

How to write custom training loops

- Writing a training loop from scratch in TensorFlow

- [Writing a training loop from scratch in JAX]
- [Writing a training loop from scratch in PyTorch]