Customizing what happens in `fit()` with TensorFlow

Author: fchollet
Date created: 2020/04/15
Last modified: 2023/06/27
Description: Overriding the training step of the Model class with TensorFlow.

Introduction

When you're doing supervised learning, you can use `fit()` and everything works smoothly.

When you need to take control of every little detail, you can write your own training loop entirely from scratch.

But what if you need a custom training algorithm, but you still want to benefit from the convenient features of `fit()`, such as callbacks, built-in distribution support, or step fusing?

A core principle of Keras is **progressive disclosure of complexity**. You should always be able to get into lower-level workflows in a gradual way. You shouldn't fall off a cliff if the high-level functionality doesn't exactly match your use case. You should be able to gain more control over the small details while retaining a commensurate amount of high-level convenience.

When you need to customize what `fit()` does, you should **override the training step function of the `Model` class**. This is the function that is called by `fit()` for every batch of data. You will then be able to call `fit()` as usual -- and it will be running your own learning algorithm.

Note that this pattern does not prevent you from building models with the Functional API. You can do this whether you're building `Sequential` models, Functional API models, or subclassed models.

Let's see how that works.

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

Setup

```
import os

# This guide can only be run with the TF backend.
os.environ["KERAS_BACKEND"] = "tensorflow"

import tensorflow as tf
import keras
from keras import layers
import numpy as np
```

A first simple example

Let's start from a simple example:

- We create a new class that subclasses `keras.Model`.
- We just override the method `train_step(self, data)`.
- We return a dictionary mapping metric names (including the loss) to their current value.

The input argument `data` is what gets passed to fit as training data:

- If you pass NumPy arrays, by calling `fit(x, y, ...)`, then `data` will be the tuple `(x, y)`
- If you pass a `tf.data.Dataset`, by calling `fit(dataset, ...)`, then `data` will be what gets yielded by `dataset` at each batch.

In the body of the `train_step()` method, we implement a regular training update, similar to what you are already familiar with. Importantly, **we compute the loss via `self.compute_loss()`**, which wraps the loss(es) function(s) that were passed to `compile()`.

Similarly, we call `metric.update_state(y, y_pred)` on metrics from `self.metrics`, to update the state of the metrics that were passed in `compile()`, and we query results from `self.metrics` at the end to retrieve their current value.

```
class CustomModel(keras.Model):
    def train_step(self, data):
        # Unpack the data. Its structure depends on your model and
        # on what you pass to `fit()`.
        x, y = data

        with tf.GradientTape() as tape:
            y_pred = self(x, training=True)  # Forward pass
            # Compute the loss value
            # (the loss function is configured in `compile()`)
            loss = self.compute_loss(y=y, y_pred=y_pred)

        # Compute gradients
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)

        # Update weights
        self.optimizer.apply(gradients, trainable_vars)

        # Update metrics (includes the metric that tracks the loss)
        for metric in self.metrics:
            if metric.name == "loss":
                metric.update_state(loss)
            else:
                metric.update_state(y, y_pred)

        # Return a dict mapping metric names to current value
        return {m.name: m.result() for m in self.metrics}

```

Let's try this out:

```
# Construct and compile an instance of CustomModel
inputs = keras.Input(shape=(32,))
outputs = keras.layers.Dense(1)(inputs)
model = CustomModel(inputs, outputs)
model.compile(optimizer="adam", loss="mse", metrics=["mae"])

# Just use `fit` as usual
x = np.random.random((1000, 32))
y = np.random.random((1000, 1))
model.fit(x, y, epochs=3)
```

Going lower-level

Naturally, you could just skip passing a loss function in `compile()`, and instead do everything *manually* in `train_step`. Likewise for metrics.

Here's a lower-level example, that only uses `compile()` to configure the optimizer:

- We start by creating `Metric` instances to track our loss and a MAE score (in `__init__()`).
- We implement a custom `train_step()` that updates the state of these metrics (by calling `update_state()` on them), then query them (via `result()`) to return their current average value, to be displayed by the progress bar and to be pass to any callback.
- Note that we would need to call `reset_states()` on our metrics between each epoch! Otherwise calling `result()` would return an average since the start of training, whereas we usually work with per-epoch averages. Thankfully, the framework can do that for us: just list any metric you want to reset in the `metrics` property of the model. The model will call `reset_states()` on any object listed here at the beginning of each `fit()` epoch or at the beginning of a call to `evaluate()`.

```

class CustomModel(keras.Model):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.loss_tracker = keras.metrics.Mean(name="loss")
        self.mae_metric = keras.metrics.MeanAbsoluteError(name="mae")
        self.loss_fn = keras.losses.MeanSquaredError()

    def train_step(self, data):
        x, y = data

        with tf.GradientTape() as tape:
            y_pred = self(x, training=True)  # Forward pass
            # Compute our own loss
            loss = self.loss_fn(y, y_pred)

        # Compute gradients
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)

        # Update weights
        self.optimizer.apply(gradients, trainable_vars)

        # Compute our own metrics
        self.loss_tracker.update_state(loss)
        self.mae_metric.update_state(y, y_pred)
        return {
            "loss": self.loss_tracker.result(),
            "mae": self.mae_metric.result(),
        }

    @property
    def metrics(self):
        # We list our `Metric` objects here so that `reset_states()` can b
        # called automatically at the start of each epoch
        # or at the start of `evaluate()`.
        return [self.loss_tracker, self.mae_metric]


# Construct an instance of CustomModel
inputs = keras.Input(shape=(32,))
outputs = keras.layers.Dense(1)(inputs)
model = CustomModel(inputs, outputs)
```

```

# We don't passs a loss or metrics here.
model.compile(optimizer="adam")

# Just use `fit` as usual -- you can use callbacks, etc.
x = np.random.random((1000, 32))
y = np.random.random((1000, 1))
model.fit(x, y, epochs=5)
```

Supporting `sample_weight` & `class_weight`

You may have noticed that our first basic example didn't make any mention of sample weighting. If you want to support the `fit()` arguments `sample_weight` and `class_weight`, you'd simply do the following:

- Unpack `sample_weight` from the `data` argument
- Pass it to `compute_loss` & `update_state` (of course, you could also just apply it manually if you don't rely on `compile()` for losses & metrics)
- That's it.

```python

class CustomModel(keras.Model):
    def train_step(self, data):
        # Unpack the data. Its structure depends on your model and
        # on what you pass to `fit()`.
        if len(data) == 3:
            x, y, sample_weight = data
        else:
            sample_weight = None
            x, y = data

        with tf.GradientTape() as tape:
            y_pred = self(x, training=True)  # Forward pass
            # Compute the loss value.
            # The loss function is configured in `compile()`.
            loss = self.compute_loss(
                y=y,
                y_pred=y_pred,
                sample_weight=sample_weight,
            )

        # Compute gradients
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)

        # Update weights
        self.optimizer.apply(gradients, trainable_vars)

        # Update the metrics.
        # Metrics are configured in `compile()`.
        for metric in self.metrics:
            if metric.name == "loss":
                metric.update_state(loss)
            else:
                metric.update_state(y, y_pred, sample_weight=sample_weight)

        # Return a dict mapping metric names to current value.
        # Note that it will include the loss (tracked in self.metrics).
        return {m.name: m.result() for m in self.metrics}


# Construct and compile an instance of CustomModel
inputs = keras.Input(shape=(32,))
```

```
outputs = keras.layers.Dense(1)(inputs)
model = CustomModel(inputs, outputs)
model.compile(optimizer="adam", loss="mse", metrics=["mae"])

# You can now use sample_weight argument
x = np.random.random((1000, 32))
y = np.random.random((1000, 1))
sw = np.random.random((1000, 1))
model.fit(x, y, sample_weight=sw, epochs=3)
```

Providing your own evaluation step

What if you want to do the same for calls to `model.evaluate()`? Then you would override `test_step` in exactly the same way. Here's what it looks like:

```

class CustomModel(keras.Model):
    def test_step(self, data):
        # Unpack the data
        x, y = data
        # Compute predictions
        y_pred = self(x, training=False)
        # Updates the metrics tracking the loss
        loss = self.compute_loss(y=y, y_pred=y_pred)
        # Update the metrics.
        for metric in self.metrics:
            if metric.name == "loss":
                metric.update_state(loss)
            else:
                metric.update_state(y, y_pred)
        # Return a dict mapping metric names to current value.
        # Note that it will include the loss (tracked in self.metrics).
        return {m.name: m.result() for m in self.metrics}


# Construct an instance of CustomModel
inputs = keras.Input(shape=(32,))
outputs = keras.layers.Dense(1)(inputs)
model = CustomModel(inputs, outputs)
model.compile(loss="mse", metrics=["mae"])

# Evaluate with our custom test_step
x = np.random.random((1000, 32))
y = np.random.random((1000, 1))
model.evaluate(x, y)
```

Wrapping up: an end-to-end GAN example

Let's walk through an end-to-end example that leverages everything you just learned.

Let's consider:

- A generator network meant to generate 28x28x1 images.
- A discriminator network meant to classify 28x28x1 images into two classes ("fake" and "real").
- One optimizer for each.

- A loss function to train the discriminator.

```
# Create the discriminator
discriminator = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(64, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2D(128, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.GlobalMaxPooling2D(),
        layers.Dense(1),
    ],
    name="discriminator",
)

# Create the generator
latent_dim = 128
generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        # We want to generate 128 coefficients to reshape into a 7x7x128 map
        layers.Dense(7 * 7 * 128),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Reshape((7, 7, 128)),
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2D(1, (7, 7), padding="same", activation="sigmoid"),
    ],
    name="generator",
)
```

Here's a feature-complete GAN class, overriding `compile()` to use its own signature, and implementing the entire GAN algorithm in 17 lines in `train_step`:

```python

class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_loss_tracker = keras.metrics.Mean(name="d_loss")
        self.g_loss_tracker = keras.metrics.Mean(name="g_loss")
        self.seed_generator = keras.random.SeedGenerator(1337)

    @property
    def metrics(self):
        return [self.d_loss_tracker, self.g_loss_tracker]

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super().compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, real_images):
        if isinstance(real_images, tuple):
            real_images = real_images[0]
        # Sample random points in the latent space
        batch_size = tf.shape(real_images)[0]
        random_latent_vectors = keras.random.normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )

        # Decode them to fake images
        generated_images = self.generator(random_latent_vectors)

        # Combine them with real images
        combined_images = tf.concat([generated_images, real_images], axis=0)

        # Assemble labels discriminating real from fake images
        labels = tf.concat(
            [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], axis=0
        )
        # Add random noise to the labels - important trick!
        labels += 0.05 * keras.random.uniform(
            tf.shape(labels), seed=self.seed_generator
```

```
        )

        # Train the discriminator
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply(grads, self.discriminator.trainable_weights)

        # Sample random points in the latent space
        random_latent_vectors = keras.random.normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )

        # Assemble labels that say "all real images"
        misleading_labels = tf.zeros((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            predictions = self.discriminator(self.generator(random_latent_vectors)
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply(grads, self.generator.trainable_weights)

        # Update metrics and return their value.
        self.d_loss_tracker.update_state(d_loss)
        self.g_loss_tracker.update_state(g_loss)
        return {
            "d_loss": self.d_loss_tracker.result(),
            "g_loss": self.g_loss_tracker.result(),
        }
```

Let's test-drive it:

```
# Prepare the dataset. We use both the training & test MNIST digits.
batch_size = 64
(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
all_digits = np.concatenate([x_train, x_test])
all_digits = all_digits.astype("float32") / 255.0
all_digits = np.reshape(all_digits, (-1, 28, 28, 1))
dataset = tf.data.Dataset.from_tensor_slices(all_digits)
```

```
dataset = dataset.shuffle(buffer_size=1024).batch(batch_size)

gan = GAN(discriminator=discriminator, generator=generator, latent_dim=latent_dim)
gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0003),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0003),
    loss_fn=keras.losses.BinaryCrossentropy(from_logits=True),
)

# To limit the execution time, we only train on 100 batches. You can train on
# the entire dataset. You will need about 20 epochs to get nice results.
gan.fit(dataset.take(100), epochs=1)
```

The ideas behind deep learning are simple, so why should their implementation be painful?