Customizing what happens in `fit()` with JAX

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

View in Colab · GitHub source

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.

Setup

```
import os

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

import jax
import keras
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 implement a fully-stateless `train_step()` method to compute current metric values (including the loss) as well as updated values for the trainable variables, the optimizer variables, and the metric variables.

Note that you can also take into account the `sample_weight` argument by:

- Unpacking the data as `x, y, sample_weight = data`
- Passing `sample_weight` to `compute_loss()`
- Passing `sample_weight` alongside `y` and `y_pred` to metrics in `stateless_update_state()`

```
        self,
        trainable_variables,
        non_trainable_variables,
        x,
        y,
        training=False,
    ):
        y_pred, non_trainable_variables = self.stateless_call(
            trainable_variables,
            non_trainable_variables,
            x,
            training=training,
        )
        loss = self.compute_loss(x, y, y_pred)
        return loss, (y_pred, non_trainable_variables)

    def train_step(self, state, data):
        (
            trainable_variables,
            non_trainable_variables,
            optimizer_variables,
            metrics_variables,
        ) = state
        x, y = data

        # Get the gradient function.
        grad_fn = jax.value_and_grad(self.compute_loss_and_updates,
has_aux=True)

        # Compute the gradients.
        (loss, (y_pred, non_trainable_variables)), grads = grad_fn(
            trainable_variables,
            non_trainable_variables,
            x,
            y,
            training=True,
        )

        # Update trainable variables and optimizer variables.
        (
            trainable_variables,
            optimizer_variables,
        ) = self.optimizer.stateless_apply(
            optimizer_variables, grads, trainable_variables
        )

        # Update metrics.
        new_metrics_vars = []
        logs = {}
        for metric in self.metrics:
            this_metric_vars = metrics_variables[
                len(new_metrics_vars) : len(new_metrics_vars) +
len(metric.variables)
            ]
            if metric.name == "loss":
                this_metric_vars =
metric.stateless_update_state(this_metric_vars, loss)
            else:
                this_metric_vars = metric.stateless_update_state(
                    this_metric_vars, y, y_pred
                )
            logs[metric.name] = metric.stateless_result(this_metric_vars)
            new_metrics_vars += this_metric_vars

        # Return metric logs and updated state variables.
```

```
            optimizer_variables,
            new_metrics_vars,
        )
        return logs, state
```

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)
```

```
Epoch 1/3
 32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - mae: 1.0022 - loss: 
1.2464
Epoch 2/3
 32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 198us/step - mae: 0.5811 - loss: 
0.4912
Epoch 3/3
 32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 231us/step - mae: 0.4386 - loss: 
0.2905

<keras.src.callbacks.history.History at 0x14da599c0>
```

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:

```
        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 compute_loss_and_updates(
        self,
        trainable_variables,
        non_trainable_variables,
        x,
        y,
        training=False,
    ):
        y_pred, non_trainable_variables = self.stateless_call(
            trainable_variables,
            non_trainable_variables,
            x,
            training=training,
        )
        loss = self.loss_fn(y, y_pred)
        return loss, (y_pred, non_trainable_variables)

    def train_step(self, state, data):
        (
            trainable_variables,
            non_trainable_variables,
            optimizer_variables,
            metrics_variables,
        ) = state
        x, y = data

        # Get the gradient function.
        grad_fn = jax.value_and_grad(self.compute_loss_and_updates,
has_aux=True)

        # Compute the gradients.
        (loss, (y_pred, non_trainable_variables)), grads = grad_fn(
            trainable_variables,
            non_trainable_variables,
            x,
            y,
            training=True,
        )

        # Update trainable variables and optimizer variables.
        (
            trainable_variables,
            optimizer_variables,
        ) = self.optimizer.stateless_apply(
            optimizer_variables, grads, trainable_variables
        )

        # Update metrics.
        loss_tracker_vars = metrics_variables[:
len(self.loss_tracker.variables)]
        mae_metric_vars = metrics_variables[len(self.loss_tracker.variables)
:]

        loss_tracker_vars = self.loss_tracker.stateless_update_state(
            loss_tracker_vars, loss
        )
        mae_metric_vars = self.mae_metric.stateless_update_state(
            mae_metric_vars, y, y_pred
        )
        )
        logs[self.mae_metric.name] =
self.mae_metric.stateless_result(mae_metric_vars)

        new_metrics_vars = loss_tracker_vars + mae_metric_vars

        # Return metric logs and updated state variables.
        state = (
            trainable_variables,
            non_trainable_variables,
            optimizer_variables,
            new_metrics_vars,
        )
        return logs, state

    @property
    def metrics(self):
        # We list our `Metric` objects here so that `reset_states()` can be
        # 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 pass 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)
```

```
Epoch 1/5
 32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.6085 - mae:
0.6580
Epoch 2/5
 32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 215us/step - loss: 0.2630 - mae:
0.4141
Epoch 3/5
 32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 202us/step - loss: 0.2271 - mae:
0.3835
Epoch 4/5
 32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 192us/step - loss: 0.2093 - mae:
0.3714
Epoch 5/5
 32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 194us/step - loss: 0.2188 - mae:
0.3818

<keras.src.callbacks.history.History at 0x14de01420>
```

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:

```
        # Unpack the data.
        x, y = data
        (
            trainable_variables,
            non_trainable_variables,
            metrics_variables,
        ) = state

        # Compute predictions and loss.
        y_pred, non_trainable_variables = self.stateless_call(
            trainable_variables,
            non_trainable_variables,
            x,
            training=False,
        )
        loss = self.compute_loss(x, y, y_pred)

        # Update metrics.
        new_metrics_vars = []
        for metric in self.metrics:
            this_metric_vars = metrics_variables[
                len(new_metrics_vars) : len(new_metrics_vars) +
len(metric.variables)
            ]
            if metric.name == "loss":
                this_metric_vars =
metric.stateless_update_state(this_metric_vars, loss)
            else:
                this_metric_vars = metric.stateless_update_state(
                    this_metric_vars, y, y_pred
                )
            logs = metric.stateless_result(this_metric_vars)
            new_metrics_vars += this_metric_vars

        # Return metric logs and updated state variables.
        state = (
            trainable_variables,
            non_trainable_variables,
            new_metrics_vars,
        )
        return logs, state


# 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)
```

```
 32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 973us/step - mae: 0.7887 - loss:
0.8385

[0.8385222554206848, 0.7956181168556213]
```

That's it!