Writing a training loop from scratch in PyTorch

Author: fchollet
Date created: 2023/06/25
Last modified: 2023/06/25
Description: Writing low-level training & evaluation loops in PyTorch.

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

Setup

```
import os

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

import torch
import keras
import numpy as np
```

Introduction

Keras provides default training and evaluation loops, `fit()` and `evaluate()`. Their usage is covered in the guide Training & evaluation with the built-in methods.

If you want to customize the learning algorithm of your model while still leveraging the convenience of `fit()` (for instance, to train a GAN using `fit()`), you can subclass the `Model` class and implement your own `train_step()` method, which is called repeatedly during `fit()`.

Now, if you want very low-level control over training & evaluation, you should write your own training & evaluation loops from scratch. This is what this guide is about.

A first end-to-end example

To write a custom training loop, we need the following ingredients:

- A model to train, of course.
- An optimizer. You could either use a `keras.optimizers` optimizer, or a native PyTorch optimizer from `torch.optim`.
- A loss function. You could either use a `keras.losses` loss, or a native PyTorch loss from `torch.nn`.
- A dataset. You could use any format: a `tf.data.Dataset`, a PyTorch `DataLoader`, a Python generator, etc.

Let's line them up. We'll use torch-native objects in each case -- except, of course, for the Keras model.

First, let's get the model and the MNIST dataset:

```

# Let's consider a simple MNIST model
def get_model():
    inputs = keras.Input(shape=(784,), name="digits")
    x1 = keras.layers.Dense(64, activation="relu")(inputs)
    x2 = keras.layers.Dense(64, activation="relu")(x1)
    outputs = keras.layers.Dense(10, name="predictions")(x2)
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model


# Create load up the MNIST dataset and put it in a torch DataLoader
# Prepare the training dataset.
batch_size = 32
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = np.reshape(x_train, (-1, 784)).astype("float32")
x_test = np.reshape(x_test, (-1, 784)).astype("float32")
y_train = keras.utils.to_categorical(y_train)
y_test = keras.utils.to_categorical(y_test)

# Reserve 10,000 samples for validation.
x_val = x_train[-10000:]
y_val = y_train[-10000:]
x_train = x_train[:-10000]
y_train = y_train[:-10000]

# Create torch Datasets
train_dataset = torch.utils.data.TensorDataset(
    torch.from_numpy(x_train), torch.from_numpy(y_train)
)
val_dataset = torch.utils.data.TensorDataset(
    torch.from_numpy(x_val), torch.from_numpy(y_val)
)

# Create DataLoaders for the Datasets
train_dataloader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True
)
val_dataloader = torch.utils.data.DataLoader(
    val_dataset, batch_size=batch_size, shuffle=False
)
```

Next, here's our PyTorch optimizer and our PyTorch loss function:

```
# Instantiate a torch optimizer
model = get_model()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

# Instantiate a torch loss function
loss_fn = torch.nn.CrossEntropyLoss()
```

Let's train our model using mini-batch gradient with a custom training loop.

Calling `loss.backward()` on a loss tensor triggers backpropagation. Once that's done, your optimizer is magically aware of the gradients for each variable and can update its variables, which is done via `optimizer.step()`. Tensors, variables, optimizers are all interconnected to one another via hidden global state. Also, don't forget to call `model.zero_grad()` before `loss.backward()`, or you won't get the right gradients for your variables.

Here's our training loop, step by step:

- We open a `for` loop that iterates over epochs
- For each epoch, we open a `for` loop that iterates over the dataset, in batches
- For each batch, we call the model on the input data to retrive the predictions, then we use them to compute a loss value
- We call `loss.backward()` to
- Outside the scope, we retrieve the gradients of the weights of the model with regard to the loss
- Finally, we use the optimizer to update the weights of the model based on the gradients

```
epochs = 3
for epoch in range(epochs):
    for step, (inputs, targets) in enumerate(train_dataloader):
        # Forward pass
        logits = model(inputs)
        loss = loss_fn(logits, targets)

        # Backward pass
        model.zero_grad()
        loss.backward()

        # Optimizer variable updates
        optimizer.step()

        # Log every 100 batches.
        if step % 100 == 0:
            print(
                f"Training loss (for 1 batch) at step {step}: {loss.detach().numpy():.4f}"
            )
            print(f"Seen so far: {(step + 1) * batch_size} samples")
```

As an alternative, let's look at what the loop looks like when using a Keras optimizer and a Keras loss function.

Important differences:

- You retrieve the gradients for the variables via `v.value.grad`, called on each trainable variable.
- You update your variables via `optimizer.apply()`, which must be called in a `torch.no_grad()` scope.

Also, a big gotcha: while all NumPy/TensorFlow/JAX/Keras APIs as well as Python `unittest` APIs use the argument order convention `fn(y_true, y_pred)` (reference values first, predicted values second), PyTorch actually uses `fn(y_pred, y_true)` for its losses. So make sure to invert the order of `logits` and `targets`.

```
model = get_model()
optimizer = keras.optimizers.Adam(learning_rate=1e-3)
loss_fn = keras.losses.CategoricalCrossentropy(from_logits=True)

for epoch in range(epochs):
    print(f"\nStart of epoch {epoch}")
    for step, (inputs, targets) in enumerate(train_dataloader):
        # Forward pass
        logits = model(inputs)
        loss = loss_fn(targets, logits)

        # Backward pass
        model.zero_grad()
        trainable_weights = [v for v in model.trainable_weights]

        # Call torch.Tensor.backward() on the loss to compute gradients
        # for the weights.
        loss.backward()
        gradients = [v.value.grad for v in trainable_weights]

        # Update weights
        with torch.no_grad():
            optimizer.apply(gradients, trainable_weights)

        # Log every 100 batches.
        if step % 100 == 0:
            print(
                f"Training loss (for 1 batch) at step {step}: {loss.detach().numpy():.4f}"
            )
            print(f"Seen so far: {(step + 1) * batch_size} samples")
```

Low-level handling of metrics

Let's add metrics monitoring to this basic training loop.

You can readily reuse built-in Keras metrics (or custom ones you wrote) in such training loops written from scratch. Here's the flow:

- Instantiate the metric at the start of the loop
- Call `metric.update_state()` after each batch
- Call `metric.result()` when you need to display the current value of the metric

- Call `metric.reset_state()` when you need to clear the state of the metric (typically at the end of an epoch)

Let's use this knowledge to compute `CategoricalAccuracy` on training and validation data at the end of each epoch:

```
# Get a fresh model
model = get_model()

# Instantiate an optimizer to train the model.
optimizer = keras.optimizers.Adam(learning_rate=1e-3)
# Instantiate a loss function.
loss_fn = keras.losses.CategoricalCrossentropy(from_logits=True)

# Prepare the metrics.
train_acc_metric = keras.metrics.CategoricalAccuracy()
val_acc_metric = keras.metrics.CategoricalAccuracy()
```

Here's our training & evaluation loop:

```
for epoch in range(epochs):
    print(f"\nStart of epoch {epoch}")
    for step, (inputs, targets) in enumerate(train_dataloader):
        # Forward pass
        logits = model(inputs)
        loss = loss_fn(targets, logits)

        # Backward pass
        model.zero_grad()
        trainable_weights = [v for v in model.trainable_weights]

        # Call torch.Tensor.backward() on the loss to compute gradients
        # for the weights.
        loss.backward()
        gradients = [v.value.grad for v in trainable_weights]

        # Update weights
        with torch.no_grad():
            optimizer.apply(gradients, trainable_weights)

        # Update training metric.
        train_acc_metric.update_state(targets, logits)

        # Log every 100 batches.
        if step % 100 == 0:
            print(
                f"Training loss (for 1 batch) at step {step}: {loss.detach().numpy():.4f}
            )
            print(f"Seen so far: {(step + 1) * batch_size} samples")

    # Display metrics at the end of each epoch.
    train_acc = train_acc_metric.result()
    print(f"Training acc over epoch: {float(train_acc):.4f}")

    # Reset training metrics at the end of each epoch
    train_acc_metric.reset_state()

    # Run a validation loop at the end of each epoch.
    for x_batch_val, y_batch_val in val_dataloader:
        val_logits = model(x_batch_val, training=False)
        # Update val metrics
        val_acc_metric.update_state(y_batch_val, val_logits)
    val_acc = val_acc_metric.result()
```

```
    val_acc_metric.reset_state()
    print(f"Validation acc: {float(val_acc):.4f}")
```

Low-level handling of losses tracked by the model

Layers & models recursively track any losses created during the forward pass by layers that call `self.add_loss(value)`. The resulting list of scalar loss values are available via the property `model.losses` at the end of the forward pass.

If you want to be using these loss components, you should sum them and add them to the main loss in your training step.

Consider this layer, that creates an activity regularization loss:

```

class ActivityRegularizationLayer(keras.layers.Layer):
    def call(self, inputs):
        self.add_loss(1e-2 * torch.sum(inputs))
        return inputs

```

Let's build a really simple model that uses it:

```
inputs = keras.Input(shape=(784,), name="digits")
x = keras.layers.Dense(64, activation="relu")(inputs)
# Insert activity regularization as a layer
x = ActivityRegularizationLayer()(x)
x = keras.layers.Dense(64, activation="relu")(x)
outputs = keras.layers.Dense(10, name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
```

Here's what our training loop should look like now:

```
# Get a fresh model
model = get_model()

# Instantiate an optimizer to train the model.
optimizer = keras.optimizers.Adam(learning_rate=1e-3)
```

```
# Instantiate a loss function.
loss_fn = keras.losses.CategoricalCrossentropy(from_logits=True)

# Prepare the metrics.
train_acc_metric = keras.metrics.CategoricalAccuracy()
val_acc_metric = keras.metrics.CategoricalAccuracy()

for epoch in range(epochs):
    print(f"\nStart of epoch {epoch}")
    for step, (inputs, targets) in enumerate(train_dataloader):
        # Forward pass
        logits = model(inputs)
        loss = loss_fn(targets, logits)
        if model.losses:
            loss = loss + torch.sum(*model.losses)

        # Backward pass
        model.zero_grad()
        trainable_weights = [v for v in model.trainable_weights]

        # Call torch.Tensor.backward() on the loss to compute gradients
        # for the weights.
        loss.backward()
        gradients = [v.value.grad for v in trainable_weights]

        # Update weights
        with torch.no_grad():
            optimizer.apply(gradients, trainable_weights)

        # Update training metric.
        train_acc_metric.update_state(targets, logits)

        # Log every 100 batches.
        if step % 100 == 0:
            print(
                f"Training loss (for 1 batch) at step {step}: {loss.detach().numpy():.4f}"
            )
            print(f"Seen so far: {(step + 1) * batch_size} samples")

    # Display metrics at the end of each epoch.
    train_acc = train_acc_metric.result()
    print(f"Training acc over epoch: {float(train_acc):.4f}")

    # Reset training metrics at the end of each epoch
```

```
    train_acc_metric.reset_state()

    # Run a validation loop at the end of each epoch.
    for x_batch_val, y_batch_val in val_dataloader:
        val_logits = model(x_batch_val, training=False)
        # Update val metrics
        val_acc_metric.update_state(y_batch_val, val_logits)
    val_acc = val_acc_metric.result()
    val_acc_metric.reset_state()
    print(f"Validation acc: {float(val_acc):.4f}")
```

That's it!

```
for i in range(100):
  print()
```