Tensorflow vs. Pytorch

Saehwa Kim

Information and Communications Engineering
Hankuk University of Foreign Studies

Outline

- Name Space
- API
- Inference
- Training with compile() and fit() in TF
- Training with tf.GradientTape() in TF
- Dataset
- Evaluating

Name Space

Tensorflow	Pytorch
`import` **`tensorflow`** `as tf` `from tf.`**`keras`**` import `**`layers`** `from tf.keras import `**`optimizer`** `from tf.keras import `**`losses`** `costF = `**`losses`**`.SparseCategoricalCrossentropy()`	`import `**`torch`** `import torch.`**`nn`**` as nn` `import torch.`**`optim`**` as optim` `costF = `**`nn`**`.CrossEntropyLoss()`

from x import y	import x.y as y
Requires x/__init__.py to expose submodule y Good for **security**: submodules of x become **private**	Allows direct submodule access Good for **robustness**: submodules of x become **public**

API (1/2)

API Style		Tensorflow	Pytorch
Subclassing	Superclass	`tf.keras.Model`	`nn.Module`
	Function	`call()`	`forward()`
Sequential Model	Using Constructor	`tf.keras.Sequential([l1, l2])`	`nn.Sequential(l1, l2)`
	Adding Layers	`m = tf.keras.Sequential()` `m.add(l1); m.add(l2)`	Not available
Functional API	Defining Input Shape	`tf.keras.Input(input_shape)`	`nn.Linear(input_shape, …)`

API (2/2)

Tensorflow	Pytorch
`layers.Dense(output_size)`	`nn.Linear(input_size, output_size)`
`model.predict(input_data)`	`model.forward(input_data)`
`model.compile()` `model.fit()` `model.evaluate()` Overriding `tf.keras.Model`'s `train_step()` *`# called in fit()`*	Not available
Not available	`model.train()` `model.eval()` *`# set the model to training/evaluating mode`* `# (ex) drop-out, batch normalization`
`loss = costF(targets, predictions)`	`loss = costF(predictions, targets)`
`gradients = tape.gradient(` `loss, model.trainable_variables)`	`loss.backward()`
`optimizer.apply_gradients(` `zip(gradients,model.trainable_variables))`	`optimizer.step()`

Inference

```python
import tensorflow as tf
from tf.keras import layers

# Define a simple neural network model
class MyModel(tf.keras.Model):
    def __init__(self):
        super(MyModel, self).__init__()
        self.flatten = layers.Flatten()
        self.fc1 = layers.Dense(128,
                                activation='relu')
        self.fc2 = layers.Dense(10,
                                activation='softmax')

    def call(self, inputs):
        x = self.flatten(inputs)
        x = self.fc1(x)
        return self.fc2(x)


# Instantiate the model
model = MyModel()

# Define a sample input tensor
input_data = tf.random.normal((32, 28, 28))

# Perform a forward pass
output = model(input_data)
```

```python
import torch
import torch.nn as nn

# Define a simple neural network model
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 10)


    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        return self.fc2(x)

# Instantiate the model
model = MyModel()

# Define a sample input tensor
input_data = torch.randn((32, 28, 28))

# Perform a forward pass
output = model(input_data)
```

Training with compile() and fit() in TF

```python
import tensorflow as tf
from tf.keras import layers


# Define a simple neural network model
class MyModel(tf.keras.Model):
     # the same as inference
# Instantiate the model
model = MyModel()
# assume that data is prepared
inputs, targets = …




# Prepare optimizer, loss, and metrics
model.compile(optimizer="adam",
      loss="sparse_categorical_crossentropy",
      metrics=["accuracy"])

# Training loop (inside fit())
model.fit(inputs, targets, epochs=5,
               batch_size=128)
```

```python
import torch; import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torchmetrics
# Define a simple neural network model
class MyModel(nn.Module):
      # the same as inference
# Instantiate the model
model = MyModel()
# assume that data is prepared
inputs, targets = …
# Create a PyTorch dataset and dataloader
dataset = TensorDataset(inputs, targets)
dataloader = DataLoader(dataset, batch_size=128)

# Prepare optimizer, loss, and metrics
optimizer = optim.Adam(model.parameters())
cost = nn.CrossEntropyLoss()
met = torchmetrics.Accuracy()

# Training loop
for epoch in range(5):
    met.reset()
    for inputs_batch, targets_batch in dataloader:
        optimizer.zero_grad()
        predictions = model(inputs_batch)
        loss = cost(predictions, targets_batch)
        met(predictions, targets_batch)
        loss.backward()
        optimizer.step()
    print(epoch+1, loss.item(), met.compute())
```

Training with tf.GradientTape() in TF

```python
import tensorflow as tf
from tf.keras import layers, losses, optimizers
# Define a simple neural network model
class MyModel(tf.keras.Model):
   # the same as inference

# Instantiate the model, loss, and optimizer
model = MyModel()
costF = losses.SparseCategoricalCrossentropy()
optimizer = optimizers.Adam()

# assume that data is prepared
inputs, targets = …

# Training loop
for epoch in range(5):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = costF(targets, predictions)
    gradients = tape.gradient(loss,
                    model.trainable_variables)
    optimizer.apply_gradients(zip(gradients,
                    model.trainable_variables))

    print(f"{epoch + 1}, {loss.numpy()}")
```

```python
import torch; import torch.nn as nn
import torch.optim as optim
# Define a simple neural network model
class MyModel(nn.Module):
      # the same as inference

# Instantiate the model, loss, and optimizer
model = MyModel()
costF = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

# assume that data is prepared
inputs, targets = …

# Training loop
for epoch in range(5):
    optimizer.zero_grad()
    predictions = model(inputs)
    loss = costF(predictions, targets)
    loss.backward()

    optimizer.step()


    print(f"{epoch + 1},{loss.item()}")
```

Dataset

```python
import tensorflow as tf


# assume that data is prepared
inputs, targets = …

# create a TF dataset
dataset = tf.data.Dataset.from_tensor_slices(
                (inputs, targets))
batched_dataset = dataset.batch(128)


# take a batch from batched_dataset
for in_batch, t_batch in batched_dataset:
    print(in_batch, t_batch )
```

```python
import torch;
from torch.utils.data import DataLoader, \
                             TensorDataset

# assume that data is prepared
inputs, targets = …

# create a PyTorch dataset and dataloader
dataset = TensorDataset(inputs, targets)

batched_dataset = DataLoader(dataset,
                              batch_size=128)

# take a batch from batched_dataset
for in_batch, t_batch in batched_dataset:
    print(in_batch, t_batch )
```

Evaluating

```python
import tensorflow as tf

# Define a simple neural network model
class MyModel(tf.keras.Model):
     # the same as inference

# Instantiate the model
model = MyModel()

# assume that data is prepared
inputs, targets, test_inputs, test_targets = …

# assume that the model has been trained

# evaluating
l, m = model.evaluate(test_inputs, test_targets)
print(m)
```

```python
import torch;
import torchmetrics
# Define a simple neural network model
class MyModel(nn.Module):
      # the same as inference

# Instantiate the model
model = MyModel()

# assume that data is prepared
inputs, targets, test_inputs, test_targets = …

# assume that the model has been trained

# evaluating
metrics = torchmetrics.Accuracy()
model.eval()
test_predictions = model(test_inputs)
print(metrics(test_predictions, test_targets))
```