AP 03 TensorFlow & PyTorch

Lecture Notes - Prof. Tobias Schaffer

1 Overview of AI Frameworks: TensorFlow vs. PyTorch

TensorFlow and PyTorch are the two most widely-used deep learning frameworks today. Each has its own philosophy, design choices, and ecosystem.

TensorFlow

- Developed by Google Brain, designed for production and scalability.
- Strong ecosystem: TensorFlow Lite, TensorBoard, TensorFlow Serving, TFX.
- Often used in industry for deployment and mobile/embedded applications.

PyTorch

- Developed by Facebook AI Research (FAIR), designed for flexibility and research.
- Dynamic computation graph (define-by-run), easier for debugging.
- Strong adoption in academia and research.

2 Model Definition

TensorFlow

- Uses `tf.keras.Sequential` or `tf.keras.Model` subclasses.
- Layers are added sequentially.
- Input shape is specified in the first layer.

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(784,)),
    tf.keras.layers.Dense(10)
])
```

PyTorch

- Defines a class that inherits from `nn.Module`.
- The forward method defines the computation explicitly.
- Input reshaping must be handled manually (e.g. flattening images).

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 64)
        self.fc2 = nn.Linear(64, 10)

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

model = Net()
```

3 Training Loops

Steps in a Training Loop

In both TensorFlow and PyTorch, a typical training loop performs the following steps for each batch of data:

1. **Forward Pass:** Feed the input batch through the model to obtain predictions.
2. **Loss Computation:** Compare predictions with true labels using a loss function.
3. **Backward Pass:** Compute gradients of the loss with respect to model parameters.
4. **Optimizer Step:** Update model parameters using gradients and optimization algorithm (e.g., Adam, SGD).
5. **Metric Update:** (Optional) Track accuracy, precision, etc.

This loop is repeated over all batches in the dataset for each epoch.

Batch Size

Batch size defines the number of training samples processed before the model's parameters are updated.

- **Smaller batch sizes** (e.g., 32, 64) lead to more frequent updates with higher gradient noise. This noise acts like a regularizer and may help the model escape sharp local minima and generalize better to unseen data.
- **Larger batch sizes** (e.g., 256, 512, or higher) yield smoother gradients and faster training per epoch, but may converge to sharp minima that generalize worse. They also require more memory and may reduce model robustness if not tuned properly.

Key point: The training dataset is divided into smaller subsets (batches), and the model is updated after processing each one.

Generalization vs. Stability: The Batch Size Trade-off

- **Smaller batch sizes** (e.g., 32) introduce noisy gradient updates.
  - This acts as implicit regularization.
  - Prevent overfitting to the training set.

  - Encourage the model to find flatter minima, which are more robust to input variations.
  - Can help the model generalize better to unseen data.

- **Larger batch sizes** (e.g., 256+) provide more stable gradients.
  - Often train faster and converge quicker.
  - Risk finding sharp minima that generalize poorly unless properly tuned.
  - Requiring a smaller learning rate due to overshooting gradients

- **Gradient Accumulation** allows training with large effective batch sizes even on memory-limited devices.

TensorFlow

- Uses `model.fit()` for training.
- Abstracts away the training loop.
- Handles batching, shuffling, and optimization internally.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
```

PyTorch

- Requires manual implementation of training loop.
- Provides full control over training.
- Must handle data loading, forward pass, loss computation, backward pass, and optimization.

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(5):
    for x, y in dataloader:
        optimizer.zero_grad()
        outputs = model(x)
        loss = loss_fn(outputs, y)
        loss.backward()
        optimizer.step()
```

TensorFlow with `tf.GradientTape`

`tf.GradientTape` allows for writing custom training loops with fine-grained control over forward and backward passes. This is useful for research, debugging, or implementing non-standard training workflows.

- Enables manual loss computation and gradient updates.
- Offers flexibility similar to PyTorch's manual loop.

- Required for building custom training logic beyond `model.fit()`.

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()

for epoch in range(5):
    for x_batch, y_batch in dataset:
        with tf.GradientTape() as tape:
            logits = model(x_batch, training=True)
            loss = loss_fn(y_batch, logits)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

Compared to `model.fit()`, this approach:

- Gives full control over training, especially useful for implementing custom behaviors like gradient clipping, conditional updates, or debugging.
- Can be accelerated with `@tf.function` to compile it into a graph for better performance.

4 Inference

- **TensorFlow**: `model.predict()` handles inference.
- **PyTorch**: Requires `model.eval()` and `torch.no_grad()` context.

```
# PyTorch Inference
model.eval()
with torch.no_grad():
    pred = model(x_test)
```

5 Exporting & Converting Models

ONNX (Open Neural Network Exchange)

- Open standard for model interchange.
- Exportable from both TensorFlow and PyTorch.

```
# PyTorch to ONNX
torch.onnx.export(model, input_tensor, "model.onnx")
```

TensorFlow Lite

- Optimized for mobile and embedded systems.
- Conversion using `TFLiteConverter`.

```
# TensorFlow to TFLite
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
```

6 Summary and Key Takeaways

Production vs. Research Focus:

- TensorFlow is often chosen for production thanks to its mature deployment stack and support for various platforms.
- PyTorch is favored in research due to its dynamic nature and open training loop implementation.

Important Coding Differences:

- TensorFlow (with `tf.keras`) abstracts training logic for cleaner code.
- PyTorch requires explicit loop implementations, allowing more control but also more code overhead and potential errors.

Key Takeaways:

- Both frameworks are powerful, well-supported, and suitable for deep learning.
- The choice depends on project goals: ease of deployment vs. research flexibility.
- Learning both is advantageous