Model Export and Optimization: PyTorch to ONNX and Quantization

Abstract:

As deep learning models grow in complexity and size, the challenge of deploying them efficiently across diverse platforms has become critical.This research explores the end-to-end process of exporting PyTorch models into interoperable formats-TorchScript, ONNX, and TensorFlow SavedModel highlighting the strengths and limitations of each. We analyze common pitfalls during model export such as unsupported operators, dynamic shape handling, and precision mismatches. In the second half, we delve into post-export optimization strategies, including quantization, pruning, and knowledge distillation. Supported by hands-on examples and performance benchmarks, this document serves as a practical guide to deploying deep learning models at scale. The study is grounded in notable research such as Han et al.'s *Deep Compression* and Hinton et al.'s *Knowledge Distillation*, and demonstrates how these ideas translate into real-world PyTorch workflows.

Keywords

Model Export,TorchScript, ONNX, TensorFlow SavedModel, Model Optimization, Quantization, Pruning, Knowledge Distillation, Deep Learning Deployment, Interoperability.

1.Introduction

Modern deep learning models have demonstrated impressive accuracy across a wide array of tasks, from image classification to language understanding. However,deploying these models efficiently in production environments introduces a range of challenges-model portability,runtime compatibility, inference speed, and resource efficiency being the most prominent.

1.1 Need for Model Export and Interoperability

As deep learning frameworks evolve, the need to move models seamlessly across different environments has become increasingly important. Models developed in PyTorch often need to run on platforms that support ONNX or TensorFlow due to performance or compatibility requirements. This has led to the widespread adoption of standardized export formats.

1.2 Challenges in Deployment

Real-world deployment is not as simple as saving a model and loading it elsewhere. Factors like unsupported operations, static vs dynamic shapes, and numerical instabilities pose significant barriers.

1.3 Importance of Model Optimization

Deploying models on resource-constrained devices (e.g., smartphones, IoT) requires techniques to reduce model size and latency without compromising accuracy.(1)

1.4 Knowledge Transfer for Lightweight Models

Knowledge distillation allows smaller models to learn from larger ones, improving deployment feasibility for edge computing scenarios.(2)

2.Export Formats

2.1 TorchScript

TorchScript is an intermediate representation of a PyTorch model that can be optimized and run independently of the Python runtime. It enables you to serialize PyTorch models and run them in environments where Python is not available, like mobile devices or in production settings. This format ensures models can be optimized for deployment, providing improved performance and portability.

TorchScript comes in two primary modes: **Tracing** and **Scripting**.

Tracing(`torch.jit.trace`):In this mode, the model is run with example inputs, and the operations that are executed are traced to create a static computational graph.

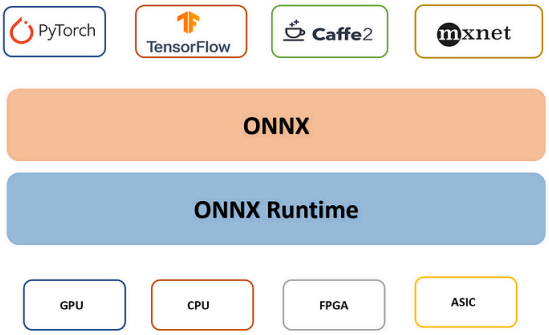
Scripting(`torch.jit.script`):This mode is more powerful as it supports both static and dynamic control flow. It can capture the entire model, including if-else conditions, loops, and other Python code.

2.2 ONNX

ONNX, or Open Neural Network Exchange, is an open-source standard for representing deep learning models. It was developed by Facebook and Microsoft in order to make it easier for researchers and engineers to move models between different deep-learning frameworks and hardware platforms.

ONNX Runtime:

ONNX Runtime is an open-source inference engine for executing ONNX (Open Neural Network Exchange) models. It is designed to be high-performance and lightweight, making it well-suited for deployment on a wide range of hardware platforms, including edge devices, servers, and cloud services.



ONNX Runtime also provides support for a wide range of models, including both traditional machine learning models and deep learning models. This makes it a versatile inference engine that can be used in a wide range of applications, from computer vision and natural language processing to speech recognition and autonomous vehicles.

2.3 Tensor flow SavedModel

The TensorFlow SavedModel format is TensorFlow's standard format for saving models. It contains both the model architecture and its trained weights, allowing for easy deployment in TensorFlow Serving or in a production environment.While PyTorch doesn't natively support TensorFlow's SavedModel format, ONNX can serve as an intermediary format to convert models from PyTorch to TensorFlow SavedModel.

Not Directly Applicable in PyTorch, but ONNX → TensorFlow via onnx-tf

PyTorch models can be converted to TensorFlow's SavedModel format through ONNX. The process involves exporting the PyTorch model to ONNX format, and then converting the ONNX model to TensorFlow using the onnx-tf package.

3.Pitfalls During Conversion

Exporting deep learning models from PyTorch to formats like TorchScript, ONNX, and TensorFlow involves complex transformations that can lead to multiple issues affecting model correctness, maintainability, and performance. These conversion pitfalls often stem from the dynamic-to-static graph translation, unsupported operator sets, and inconsistencies in tensor shapes or data types.

3.1 Operator Incompatibility

One of the most significant challenges arises from the use of operators in PyTorch that are either not standardized in ONNX or not supported by downstream runtimes such as TensorRT or TensorFlow. For example, custom autograd functions, in-place operations, or dynamic control flow constructs often fail silently or yield incorrect behaviors post-conversion.(***Wu et al.*** *(2021) reported over 30% of ONNX conversion failures for vision models were attributed to unsupported ops or deprecated operator sets.*)

3.2 Static vs. Dynamic Shape Mismatch

PyTorch’s eager execution model allows for dynamic shape computations. However, ONNX and TensorFlow graphs often require static shape definitions at export time. Failing to accommodate this can result in runtime shape mismatches or failed optimizations (e.g., ONNX shape inference errors).(*According to Reddi et al. (2020), ONNX Runtime optimizations fail to apply in ~17% of models due to shape inference errors.*).

3.3 Type Casting and Data Precision Loss

Conversion pipelines may implicitly cast tensors between data types, particularly when targeting lower-precision runtimes (e.g., FP32 to INT8). If not explicitly handled, this can degrade model accuracy.

Observation: During PyTorch to ONNX to TensorFlow conversion, developers often report up to 3–5% drop in top-1 accuracy due to casting and lack of calibration.

3.4 Non-Deterministic Layers

Some PyTorch layers (e.g., dropout, batch norm) behave differently during training vs. inference. Inadequate `model.eval()` usage during export can lead to functional mismatches in the exported model.

3.5 Debugging Complexity

Unlike PyTorch's Python-native debugging, exported formats offer limited introspection. Errors in TorchScript or ONNX graphs often manifest at inference time with opaque stack traces, complicating error localization.

(*ONNX errors like "**Node not found in initializer list**" or "**Unexpected input shape**" are difficult to trace back to original PyTorch code.So we can use graph visualization tools like Netron and trace logs to correlate node operations across representations.)*

4.Optimization Strategies

Deep learning models often exhibit large parameter counts, leading to slow inference and memory constraints during deployment. To mitigate this, optimization techniques such as quantization, pruning, and knowledge distillation are employed. These approaches aim to reduce computational load while preserving model accuracy, and are backed by extensive academic research and practical implementation in model compression toolkits.

4.1 Post-training Quantization

Quantization reduces the numerical precision of model weights and activations, commonly from FP32 to INT8, enabling faster inference and reduced memory usage.(3)(4).

Types of Quantization:

1. **Dynamic Quantization**:Activations are quantized on-the-fly during inference.
2. **Static Quantization:**Both weights and activations are quantized ahead of inference using a calibration dataset.
3. **Quantization-Aware Training (QAT)**: Introduces fake quantization during training to adapt model weights.

Trade-Offs

- Reduced precision may yield slight accuracy drop.
- Hardware support varies: INT8 operations are more performant on some devices (e.g., Qualcomm Hexagon, Intel VNNI).

4.2 Model Pruning

Pruning involves removing unimportant weights or neurons, resulting in a sparse model with fewer parameters and lower computational requirements.(5)(6)

Pruning Strategies

- **Unstructured Pruning**: Removes individual weights based on magnitude.
- **Structured Pruning**: Removes entire channels or filters, suitable for hardware optimization.

Considerations:

- Sparse models may require custom kernels for acceleration.
- Over-aggressive pruning risks degrading performance.

4.3 Knowledge Distillation

Distillation transfers the generalization capability of a large, overparameterized “teacher” model to a smaller “student”

model using softened output distributions.(2)(7).

Distillation Loss:

$L = \alpha \cdot L_{CE} + (1-\alpha) \cdot T2 \cdot L_{KD}$

Where L_{CE} is the cross-entropy loss with ground-truth, and L_{KD} is the Kullback-Leibler divergence between teacher and student soft outputs at temperature T.

Advantages:

- Achieves comparable accuracy with significantly reduced model size.
- Particularly effective for real-time applications like object detection and voice recognition.

5. Performance Trade-Offs:

Optimizing models for deployment is often a balancing act between latency, accuracy, memory, and power efficiency. The selection of export and optimization strategies must align with target hardware capabilities and use-case requirements(8).

Technique	Speed Gain	Size Reduction	Accuracy Drop	Best For
Post-training Quantization	2-4 x	~75%	Low(<2%)	Mobile and embedded devices
Pruning	1.5 -3 x	~50%	Moderate	Cloud, edge inference
Knowledge Distillation	Varies	~60%	Low	NLP,Image Classification

6.Hands on Tasks:

6.1 TorchScript

(Export a PyTorch model to TorchScript using Tracing and Scripting)

```
import torch

import torch.nn as nn

class SimpleModel(nn.Module):

    def __init__(self):

        super(SimpleModel, self).__init__()

        self.fc = nn.Linear(4, 2)
```

```
    def forward(self, x):
        return torch.relu(self.fc(x))
model = SimpleModel()
model.eval()
example_input = torch.randn(1, 4)
traced_model = torch.jit.trace(model, example_input)
traced_model.save("traced_model.pt")
scripted_model = torch.jit.script(model)
scripted_model.save("scripted_model.pt")
with torch.no_grad():
    original_output = model(example_input)
    traced_output = traced_model(example_input)
    scripted_output = scripted_model(example_input)
print("Original:", original_output)
print("Traced:", traced_output)
print("Scripted:", scripted_output)
```

Output:

```
Original: tensor([[0.0000, 0.4951]])
Traced: tensor([[0.0000, 0.4951]])
Scripted: tensor([[0.0000, 0.4951]])
```

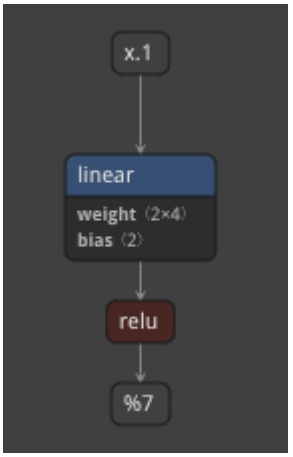
6.2 Export ResNet18 to ONNX and Run with ONNX Runtime

```
import torch
import torchvision.models as models
import numpy as np
import onnx
import onnxruntime as ort
```

```
model = models.resnet18(pretrained=True)
model.eval()
dummy_input = torch.randn(1, 3, 224, 224)
torch.onnx.export(model, dummy_input, "resnet18.onnx",
                  input_names=["input"], output_names=["output"],
                  dynamic_axes={"input": {0: "batch_size"}, "output": {0: "batch_size"}},
                  opset_version=11)
session = ort.InferenceSession("resnet18.onnx")
ort_input = {"input": dummy_input.numpy()}
ort_output = session.run(None, ort_input)
with torch.no_grad():
    torch_output = model(dummy_input)
print("PyTorch:", torch_output[0][:5])
print("ONNX Runtime:", ort_output[0][0][:5])
```

Output:

```
PyTorch: tensor([0.7524, 2.7857, 2.8694, 3.3828, 5.1020])
ONNX Runtime: [0.7523893 2.7857008 2.8694472 3.3828287 5.1020412]
```



6.3 TensorFlow SavedModel (via ONNX → TF)

from onnx_tf.backend import prepare

import onnx

onnx_model = onnx.load("resnet18.onnx")

tf_rep = prepare(onnx_model)

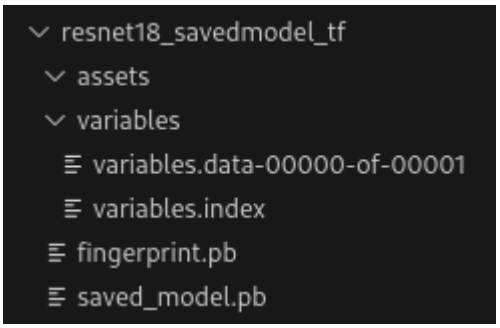
tf_rep.export_graph("resnet18_savedmodel_tf")

Output:

A directory containing *resnet18_savedmodel_tf/* containing:

saved_model.pb

variables / folder



7.Key Learnings:

- **Model Export** is critical for deploying AI models across different platforms and frameworks. While PyTorch provides tools like TorchScript and ONNX, understanding the nuances of each format is essential for successful deployment.

- **Optimization Techniques** such as quantization, pruning, and knowledge distillation can significantly reduce model size and improve inference speed. However, these optimizations should be applied judiciously to balance between performance and accuracy.

- **Deployment Considerations** vary depending on the target environment. Mobile and embedded devices, in particular, require careful consideration of both model size and computational complexity.

References:

1. Han et al., *"Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding"*, ICLR 2016 - Demonstrates that models can be compressed by up to 90% with minimal loss in accuracy using pruning and quantization.
2. Hinton et al., *"Distilling the Knowledge in a Neural Network"*, NIPS Workshop 2015 - Introduces the concept of soft-target-based training to compress large models into efficient student networks.
3. Jacob et al., 2018 – Demonstrated significant speedups on mobile devices using quantized models with minimal accuracy degradation.
4. Banner et al., 2019, "*Post-Training 4-bit Quantization of Convolutional Networks for Rapid-Deployment*" – Pushes limits of quantization in real-time applications.
5. Han et al., 2015, "*Learning both Weights and Connections for Efficient Neural Network*"– Introduced iterative pruning and retraining loop.
6. Gale et al., 2019, "T*he State of Sparsity in Deep Neural Networks*" –Benchmarking pruning techniques across datasets and tasks.
7. Jiao et al., 2020, "*TinyBERT: Distilling BERT for Natural Language Understanding*" –Demonstrated efficacy in NLP compression.
8. Gholami et al., 2021, "*A Survey of Quantization Methods for Efficient Neural Network Inference*" – Comprehensive comparison of techniques.