Model Deployment

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

From Training to Production

The Journey:

1. Train model on laptop/server
2. Evaluate performance
3. Package model
4. Deploy to production
5. Serve predictions
6. Monitor performance
7. Update and retrain

Key Concept: The "Wall of Confusion"

Developers write code, Ops deploy it. In MLOps, Data Scientists train models, but Engineers deploy them.

Deployment Options Overview

Strategy	Latency	Throughput	Use Case
REST API	Low (ms)	High	Real-time (Chatbots, Recommendations)
Batch	High (hrs)	Very High	Nightly Reports, Churn Prediction
Edge	Very Low	Low	IoT, Privacy-sensitive (FaceID)
Streaming	Low (ms)	High	Fraud Detection, Sensor Data

graph TD A[Client Request] --> B{Type?}; B -- Online --> C[REST API]; B -- Offline --> D[Batch Job]; B -- Event --> E[Stream Processor];

Model Quantization: Theory

Why? Models are huge.

- ResNet-50: ~98MB (Float32)
- Quantized: ~25MB (Int8) -> **4x smaller, 2-4x faster**

How it works: Map continuous float values to discrete integers.

$$Q(x) = \text{round}(x/S + Z)$$

- x: Input float
- S: Scale factor
- Z: Zero point
- $Q(x)$: Output integer

Trade-off: Precision vs. Size.

Usually, accuracy drop is < 1% for standard models.

Quantization Visualized

Float32 has a huge dynamic range. Int8 has only 256 values [-128, 127].

graph LR A[Float32 Range] -- Mapping --> B[Int8 Range]; A --> |-10.5| B1[-128]; A --> |0.0| B2[0]; A --> |10.5| B3[127]; style A fill:#e1f5fe style B fill:#fff9c4

PyTorch Code (Post-Training Static Quantization):

```
import torch

# 1. Define model
model = MyModel()
model.eval()

# 2. Fuse layers (Conv+BN+ReLU) for speed
model.fuse_model()

# 3. Prepare config (x86 or ARM)
model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
torch.quantization.prepare(model, inplace=True)
```

ONNX: The Universal Bridge

Problem: PyTorch models don't run in TensorFlow. Deployment hardware (NVIDIA, Intel) needs specific optimization.

Solution: ONNX (Open Neural Network Exchange).

- A common graph representation.
- Write in *any* framework -> Export to ONNX -> Run on *any* hardware.

graph LR A[PyTorch] --> D[ONNX Graph]; B[TensorFlow] --> D; C[Scikit-Learn] --> D; D --> E[ONNX Runtime]; E --> F[CPU]; E --> G[NVIDIA GPU]; E --> H[Mobile (ARM)];

Serving Architecture: Containerization

Why Docker?

"It works on my machine" is not a deployment strategy.

Containers package code + dependencies + OS libraries.

Comparing Virtual Machines vs Containers:

Virtual Machine

- Heavy (GBs)
- Guest OS per app
- Slow boot

Container

- Lightweight (MBs)

Scaling & Load Balancing

One server isn't enough.

Horizontal Scaling: Add more containers.
Load Balancer (Nginx): Distributes traffic.

graph LR User --> LB[Load Balancer]; LB --> S1[Model Replica 1]; LB --> S2[Model Replica 2]; LB --> S3[Model Replica 3];
Kubernetes (K8s):

- Orchestrates containers.
- Auto-scaling: "If CPU > 80%, add replica".
- Self-healing: "If replica crashes, restart it".

Deployment Strategies

1. **Blue/Green Deployment**:
   - Run two environments: Blue (Current), Green (New).
   - Switch traffic 100% to Green when ready.
   - **Pros**: Instant rollback. **Cons**: 2x cost.
2. **Canary Deployment**:
   - Send 10% traffic to V2, 90% to V1.
   - Monitor errors. Gradually increase to 100%.
   - **Pros**: Safer. **Cons**: Complex routing.
3. **A/B Testing**:
   - Split traffic to measure *business impact* (not just errors).

Model Drift: The Silent Killer

Code doesn't change, but **data** does.

1. **Data Drift**: Input distribution changes ($P(X)$).
   - *Example*: Training images were sunny, now users upload night photos.
2. **Concept Drift**: Relationship changes ($P(Y|X)$).
   - *Example*: "Spam" definition changes over time.

graph TD A[Training Data Dist] -- Time Passes --> B[Production Data Dist]; B --> C{Diff > Threshold?}; C -- Yes --> D[Alert & Retrain]; C -- No --> E[Keep Serving];
Detection:

- Statistical tests: Kolmogorov-Smirnov (KS) test.
- Tools: Evidently AI, Alibi Detect.

API Design with FastAPI

Pydantic ensures inputs match your schema.

```
from fastapi import FastAPI
from pydantic import BaseModel, conlist

app = FastAPI()

# Define constraints
class InputData(BaseModel):
    # List of exactly 10 floats
    features: conlist(float, min_items=10, max_items=10)

@app.post("/predict")
def predict(data: InputData):
    # No need to check len(data.features), FastAPI did it!
    prediction = model.predict([data.features])
    return {"class": int(prediction[0])}
```

Lab Overview: From Local to Cloud

Today's Lab:

1. **Serialize**: Save a scikit-learn model.
2. **API**: Wrap it in FastAPI.
3. **Containerize**: Write a Dockerfile.
4. **Optimize**: Convert to ONNX and benchmark speedup.
5. **Deploy**: (Optional) Push to Render/Heroku.

Questions?

Let's ship some code!