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

Challenges:

- Model size (GB)
- Inference speed

Deployment Options

1. **REST API**: HTTP endpoints (FastAPI, Flask)
2. **Batch Prediction**: Process large datasets offline
3. **Edge Deployment**: On-device (mobile, IoT)
4. **Streaming**: Real-time inference (Kafka, Flink)
5. **Serverless**: AWS Lambda, Google Cloud Functions

Choice depends on:

- Latency requirements
- Throughput
- Cost
- Infrastructure

Model Quantization

Problem: Models are large (ResNet-50: 98MB, GPT-3: 350GB)

Solution: Reduce model size with quantization

Quantization: Convert weights from 32-bit floats to lower precision

Types:

- INT8: 8-bit integers (4x smaller)
- INT4: 4-bit integers (8x smaller)
- Mixed precision: Critical layers in FP16, others in INT8

Trade-off: Size vs. accuracy

PyTorch Quantization

Post-Training Quantization:

```
import torch
from torchvision import models

# Load model
model = models.resnet50(pretrained=True)
model.eval()

# Quantize
model_quantized = torch.quantization.quantize_dynamic(
    model,
    {torch.nn.Linear},  # Layers to quantize
    dtype=torch.qint8
)

# Save
torch.save(model_quantized.state_dict(), 'model_quantized.pth')

# Compare sizes
```

ONNX: Open Neural Network Exchange

Standard format for ML models

Why ONNX?

- Framework interoperability (PyTorch → TensorFlow)
- Optimized for inference
- Hardware acceleration
- Smaller models

Convert to ONNX:

```
import torch

model = load_model()
dummy_input = torch.randn(1, 3, 224, 224)
```

ONNX Runtime

Fast inference with ONNX:

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(
    None,
    {'input': input_data}
)

print("Prediction:", outputs[0])
```

Model Serving with FastAPI

Create REST API for predictions:

```
from fastapi import FastAPI
import torch
import joblib

app = FastAPI()

# Load model at startup
model = None

@app.on_event("startup")
def load_model():
    global model
    model = joblib.load("models/model.pkl")

@app.post("/predict")
def predict(data: dict):
    features = data['features']
    prediction = model.predict([features])[0]
    probability = model.predict_proba([features])[0].max()

    return {
        "prediction": int(prediction),
        "confidence": float(probability)
    }
```

Batch Prediction

For large-scale offline inference:

```
import pandas as pd

def batch_predict(input_file, output_file, batch_size=1000):
    model = load_model()
    df = pd.read_csv(input_file)

    predictions = []

    for i in range(0, len(df), batch_size):
        batch = df[i:i+batch_size]
        features = batch[feature_columns].values
        preds = model.predict(features)
        predictions.extend(preds)

    df['prediction'] = predictions
    df.to_csv(output_file, index=False)
    print(f"Predictions saved to {output_file}")
```

Docker for Deployment

Containerize your model service:

```
FROM python:3.10-slim

WORKDIR /app

# Copy requirements and install
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy model and code
COPY models/ ./models/
COPY app.py .

# Expose port
EXPOSE 8000

# Run FastAPI
```

TorchServe

Production-ready model serving:

```
# Install
pip install torchserve torch-model-archiver

# Archive model
torch-model-archiver \
  --model-name resnet50 \
  --version 1.0 \
  --model-file model.py \
  --serialized-file model.pth \
  --handler image_classifier

# Start server
torchserve --start --model-store model_store --models resnet50=resnet50.mar

# Inference
curl -X POST http://localhost:8080/predictions/resnet50 -T image.jpg
```

TensorFlow Serving

For TensorFlow models:

```
# Save model in SavedModel format
model.save('saved_model/1/')

# Run with Docker
docker run -p 8501:8501 \
  -v $(pwd)/saved_model:/models/my_model \
  -e MODEL_NAME=my_model \
  tensorflow/serving

# Query
curl -X POST http://localhost:8501/v1/models/my_model:predict \
  -d '{"instances": [[1.0, 2.0, 3.0]]}'
```

Load Balancing

Handle high traffic:

```
# nginx.conf
upstream backend {
    server model-server-1:8000;
    server model-server-2:8000;
    server model-server-3:8000;
}

server {
    listen 80;

    location /predict {
        proxy_pass http://backend;
    }
}
```


Scale horizontally:

Caching Predictions

Speed up repeated requests:

```
from functools import lru_cache
import hashlib
import json

# In-memory cache
@lru_cache(maxsize=1000)
def predict_cached(features_json):
    features = json.loads(features_json)
    return model.predict([features])[0]

# Redis cache
import redis
r = redis.Redis()

def predict_with_redis(features):
    key = hashlib.md5(str(features).encode()).hexdigest()

    # Check cache
    cached = r.get(key)
    if cached:
        return json.loads(cached)

    # Predict and cache
```

Model Versioning

Manage multiple model versions:

```
from fastapi import FastAPI, Path

app = FastAPI()

models = {
    'v1': load_model('models/model_v1.pkl'),
    'v2': load_model('models/model_v2.pkl'),
}

@app.post("/predict/{version}")
def predict(version: str = Path( ... , regex="^(v1|v2)$"), data: dict):
    model = models[version]
    prediction = model.predict([data['features']])[0]
    return {"prediction": int(prediction), "version": version}
```

Gradual Rollout (Canary Deployment)

Test new model with small traffic:

```
import random

def route_to_model(user_id):
    # 10% to new model, 90% to old
    if hash(user_id) % 100 < 10:
        return models['v2']
    return models['v1']

@app.post("/predict")
def predict(user_id: str, data: dict):
    model = route_to_model(user_id)
    prediction = model.predict([data['features']])[0]
    return {"prediction": int(prediction)}
```

Model Monitoring

Track predictions in production:

```
import mlflow

@app.post("/predict")
def predict(data: dict):
    features = data['features']

    # Predict
    prediction = model.predict([features])[0]
    confidence = model.predict_proba([features])[0].max()

    # Log to MLflow
    with mlflow.start_run():
        mlflow.log_metric("confidence", confidence)
        mlflow.log_param("prediction", prediction)

    # Log to database for monitoring
    log_prediction(features, prediction, confidence)
```

Data Drift Detection

Alert when input distribution changes:

```
from scipy.stats import ks_2samp
import numpy as np

training_stats = load_training_statistics()

def check_drift(current_data):
    alerts = []

    for feature in current_data.columns:
        statistic, p_value = ks_2samp(
            current_data[feature],
            training_stats[feature]
        )

        if p_value < 0.05:
            alerts.append(f"Drift detected in {feature}: p={p_value:.4f}")

    return alerts
```

Model Compression Techniques

1. **Pruning**: Remove unimportant weights

```
import torch.nn.utils.prune as prune

# Prune 40% of weights
prune.l1_unstructured(model.fc, name='weight', amount=0.4)
```

2. **Knowledge Distillation**: Train small model from large

```
# Student learns from teacher
loss = student_loss + distillation_loss(student, teacher, temperature=3)
```

3. **Low-Rank Factorization**: Decompose weight matrices

Edge Deployment

Deploy to mobile/IoT devices:

1. TensorFlow Lite:

```
converter = tf.lite.TFLiteConverter.from_saved_model('saved_model')
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

with open('model.tflite', 'wb') as f:
    f.write(tflite_model)
```

2. PyTorch Mobile:

```
model.eval()
example = torch.rand(1, 3, 224, 224)
traced_script_module = torch.jit.trace(model, example)
traced_script_module.save("model_mobile.pt")
```

Serverless Deployment

AWS Lambda example:

```
import json
import boto3
import joblib

model = joblib.load('/tmp/model.pkl')

def lambda_handler(event, context):
    data = json.loads(event['body'])
    features = data['features']

    prediction = model.predict([features])[0]

    return {
        'statusCode': 200,
        'body': json.dumps({'prediction': int(prediction)})
    }
```

Kubernetes Deployment

Scale model serving:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: model-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: model
  template:
    metadata:
      labels:
        app: model
    spec:
      containers:
      - name: model
        image: my-model:latest
        ports:
        - containerPort: 8000
        resources:
          requests:
            memory: "2Gi"
            cpu: "1"
```

GPU Deployment

Leverage GPUs for inference:

```
FROM nvidia/cuda:11.7.0-runtime-ubuntu20.04

# Install Python and dependencies
RUN apt-get update && apt-get install -y python3-pip
COPY requirements.txt .
RUN pip3 install -r requirements.txt

COPY app.py models/ /app/
WORKDIR /app

CMD ["uvicorn", "app:app", "--host", "0.0.0.0"]
```

Run with GPU:

```
docker run --gpus all -p 8000:8000 model-gpu
```

Cost Optimization

Reduce deployment costs:

1. **Use smaller models**: Quantization, pruning
2. **Batch predictions**: Amortize overhead
3. **Auto-scaling**: Scale down during low traffic
4. **Spot instances**: 70% cheaper (AWS, GCP)
5. **Caching**: Avoid redundant predictions
6. **Model compilation**: TensorRT, OpenVINO

Example savings:

- Standard instance: $0.50/hour
- Spot instance: $0.15/hour
- Savings: 70%

Latency Optimization

Techniques:

1. **Model optimization**: ONNX, TensorRT
2. **Batch inference**: Process multiple at once
3. **Async processing**: Non-blocking
4. **Edge caching**: CDN for static results
5. **GPU acceleration**: 10-100x faster
6. **Model quantization**: Smaller, faster

Latency targets:

- Real-time: <100ms
- Interactive: <1s
- Batch: Minutes/hours OK

Security Considerations

1. Input validation:

```
from pydantic import BaseModel, validator

class PredictionRequest(BaseModel):
    features: list

    @validator('features')
    def validate_features(cls, v):
        if len(v) != 10:
            raise ValueError("Expected 10 features")
        if any(abs(x) > 1000 for x in v):
            raise ValueError("Feature values out of range")
        return v
```

2. Rate limiting:

Deployment Checklist

Before production:

- [ ] Model tested on validation set
- [ ] API endpoints tested
- [ ] Load testing completed
- [ ] Monitoring set up
- [ ] Logging configured
- [ ] Error handling implemented
- [ ] Security measures in place
- [ ] Documentation written
- [ ] Rollback plan prepared
- [ ] Cost estimated

What We've Learned

Optimization:

- Model quantization (INT8, ONNX)
- Compression techniques
- Performance tuning

Serving:

- REST APIs with FastAPI
- TorchServe, TensorFlow Serving
- Batch prediction

Deployment:

- Docker containers

Resources

Tools:

- TorchServe: https://pytorch.org/serve/
- TensorFlow Serving: https://www.tensorflow.org/tfx/guide/serving
- ONNX: https://onnx.ai/
- BentoML: https://www.bentoml.com/

Platforms:

- AWS SageMaker
- Google AI Platform
- Azure ML
- Hugging Face Inference API

Questions?

Lab: Deploy model as API, optimize and test