HSLU Lucerne University of Applied Sciences and Arts

Deployment and Serving

Marc Bravin
Andreas Marfurt

MLOps

Information Technology
08.11.2024

Overview

- Batch vs. Stream Processing
- Deploying on Cloud vs. Edge Devices
- Model serving
- APIs

What is Model Deployment?

Deployment means making your model **running and accessible** to end users in production.

🔬 **Light Production**

Generating plots from notebooks for business teams

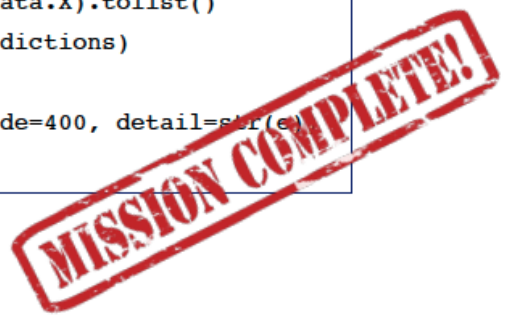
⚡ **Heavy Production**

Serving millions of users with millisecond latency and 99% uptime

The Easy Way

- Deployment can seem easy if complex challenges are ignored
- Simple approach: wrapping the model‘s predict function into an API
- Frameworks like FastAPI help expose the model as a REST endpoint
- Depdencies are packed into a container for portability
- The model and the container are pushed to cloud services like AWS or GCP
- This works well for small-scale applications but lacks production-level robustness

```
@app.post("/predict", response_model=PredictionOutput)
async def predict(data: PredictionInput):
    try:
        predictions = MODEL.predict(data.X).tolist()
        return PredictionOutput(y=predictions)
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
```



The Hard Part

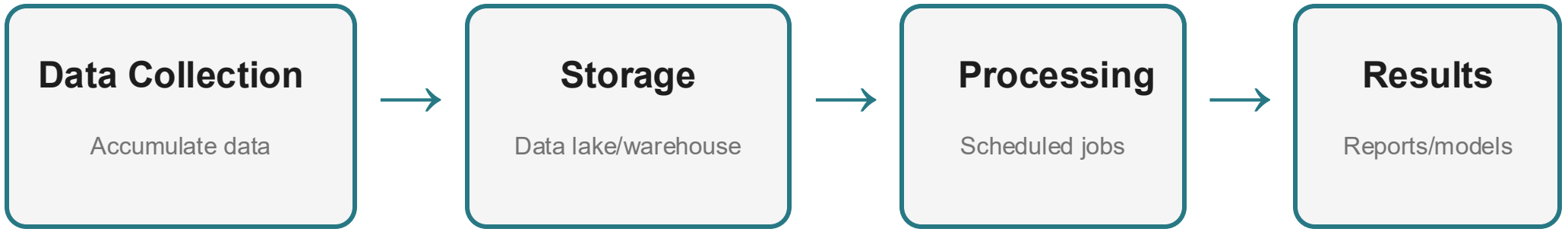
- Scaling models to serve millions of users with low latency
- Ensuring 99% or higher uptime
- Setting up monitoring and altering for immediate issue detection
- Diagnosing failures quickly to minimizing downtime
- Seamless deploying updates without service distruption
- Balancing performance, cost and maintainability in production

Batch Processing

Processing large volumes of data collected over time in scheduled intervals

- Data processed in fixed-size chunks or batches
- Often scheduled execution (hourly, daily, weekly)
- High throughput, optimized for volume
- Results available after processing completes

Batch Processing Architecture



Common Use-Cases

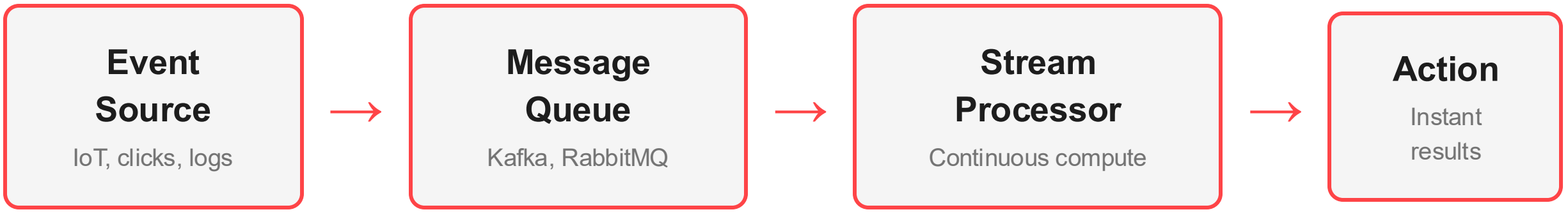
- Daily model retraining with accumulated data
- Monthly reporting and analytics dashboards
- Historical data analysis and trend detection
- ETL pipelines for data warehousing

Streaming Processing

Processing data continuously as it arrives in real-time or near real-time

- Data processed record-by-record or micro-batches
- Continuous, event-driven execution
- Low latency, optimized for speed
- Intermediate results and insights

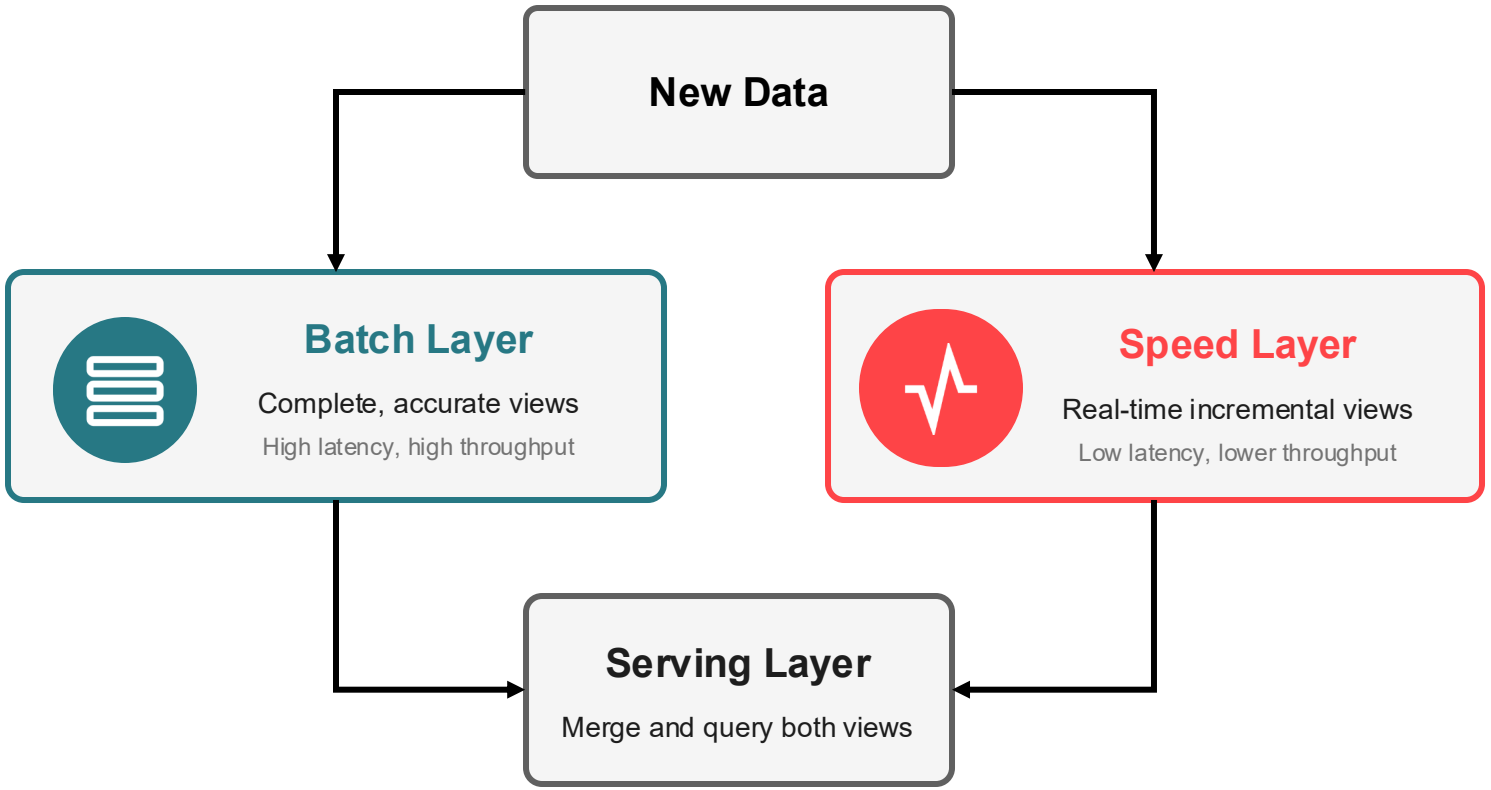
Streaming Processing Architecture



Common Use-Cases

- Real-time fraud detection
- Live recommendation systems
- IoT anomaly detection
- Real-time Monitoring

Lambda Architecture: Combination of Both



Batch Layer (Cold Path)
Stores the immutable master dataset and pre-computes accurate, comprehensive views from all historical data, ideal for training robust ML models.

Speed Layer (Hot Path) Processes live data streams in near real-time to provide low-latency results, enabling immediate model inference and real-time monitoring (e.g., for data drift).

Serving Layer
Merges results from both the batch and speed layers to provide a single, unified view, allowing applications to query both historical features and fresh, real-time predictions.

Real-World Example: TikTok

Recommend personalized videos to 1+ billion users with sub-second latency

Batch Processing (Offline)

Video Embedding Generation
- Extract visual features (scenes, objects)
- Audio analysis (music, speech)
- Text embeddings (captions, hashtags)

User Profile Updates
- Aggregate daily interactions
- Compute interest vectors
- Update user clusters

Model Training
- Train on billions of interactions
- A/B test new models
- Scheduled: nightly or weekly

Stream Processing (Online)

Real-time Recommendations
- Serve personalized feed instantly
- Use pre-computed embeddings

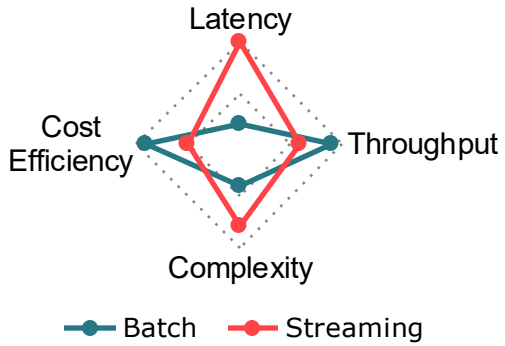
Interaction Processing
- Track likes, shares, watch time
- Update engagement signals
- Adjust feed in real-time

Trending Detection
- Identify viral content
- Boost trending videos

Technologies

- Batch processing
  - Hadoop
  - Spark
- Streaming
  - Queues: Kafka, RabbitMQ
  - Processing: Storm
- Combination
  - Lambda architecture
  - Flink
- All products are open-source

Batch vs. Streaming

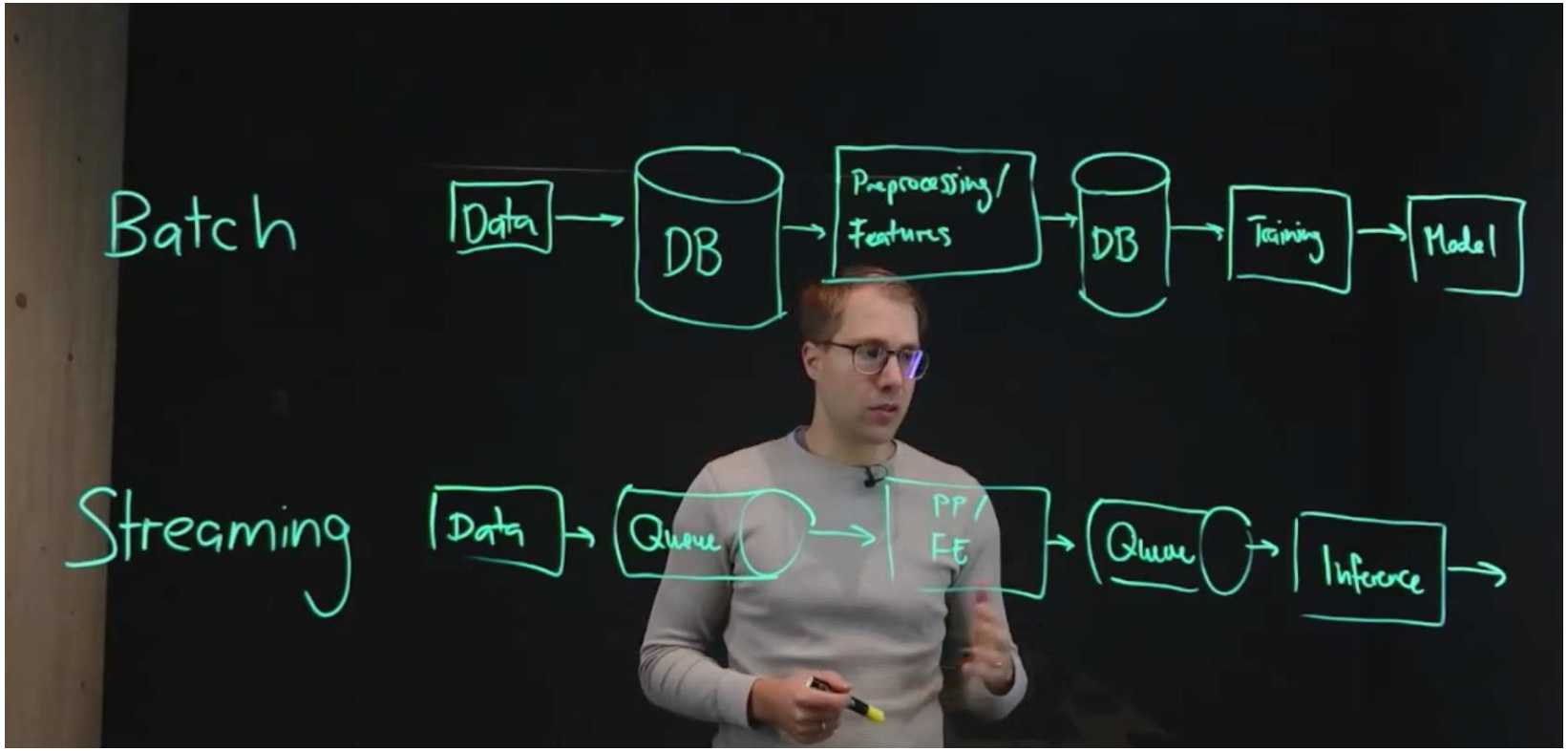


	Batch	**Streaming**
Execution frequency	Periodical (e.g. nightly)	When a request comes in
Serving	Asynchronous	Generally synchronous
Optimized for	High throughput	Low latency
Data	Bounded (fixed end time), finite	Unbounded, possibly infinite
Sorting	By any attribute (e.g. user ID)	Approximately sorted by request time
Used for	Computations over entire history, slow-changing attributes (e.g. interests)	Real-time predictions, decisions in the moment, reactions to new developments (e.g. news)
Example use cases	User interests, movie recommendations	Fraud detection, location-based ads
Example products	Hadoop, Spark	Kafka, RabbitMQ

A note about programming languages...

- Hadoop/Spark/Flink natively written in Java/Scala
- Wrappers/client libraries available in many languages, including Python
  - PySpark
  - PyFlink
- Use modular design and define clear APIs

Batch vs. Streaming Video on ILIAS

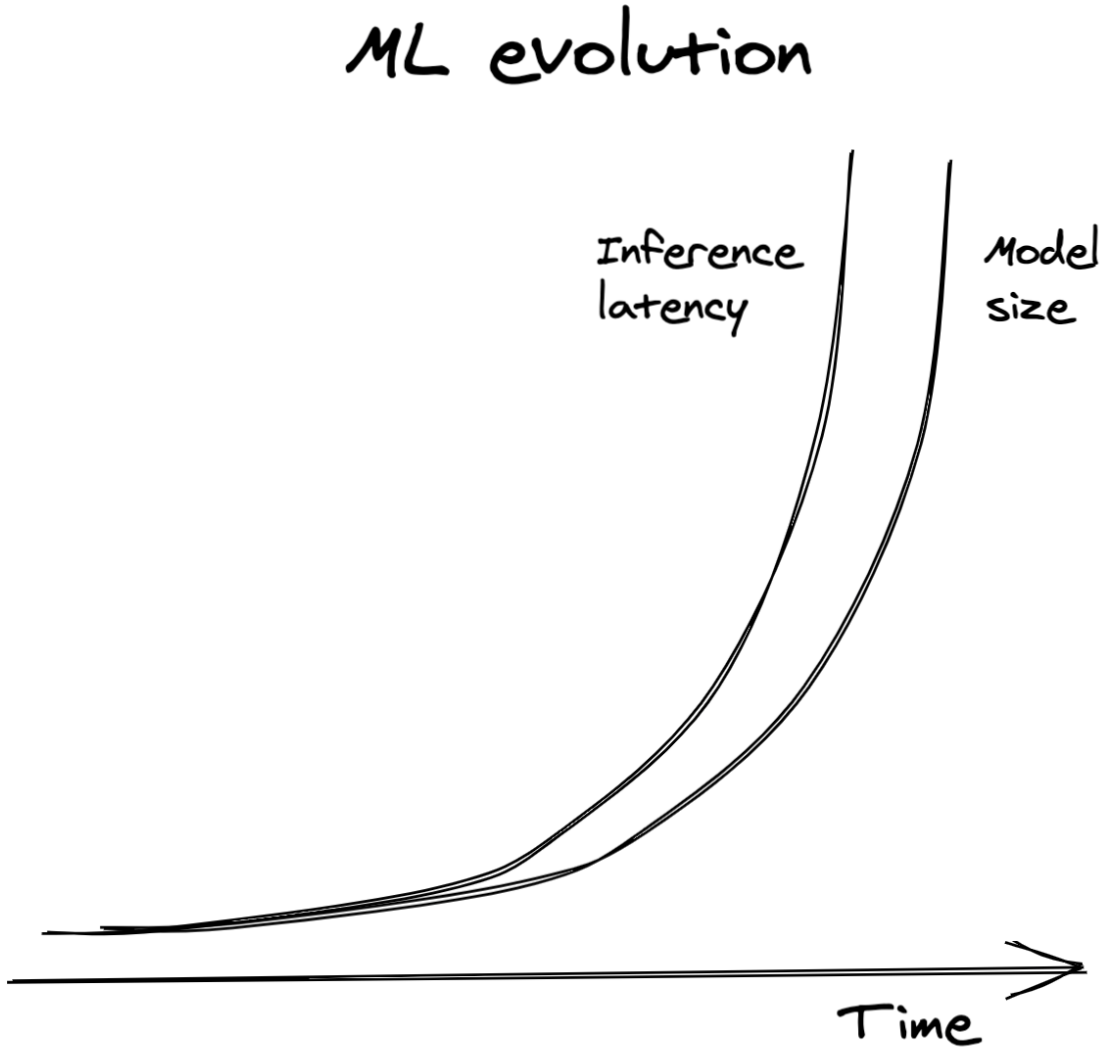


Deploying on Cloud vs. Edge Devices

Cloud vs. Edge Computation

	Cloud	**Edge**
Hardware	Servers in datacenters	Browsers, phones, tablets, laptops, smart watches, cars
Examples	ChatGPT, Google search, Alexa, translation for rare languages	Spelling correction, auto completion, translation of popular languages, fingerprint/face unlocking
Use cases	High resource requirement (large models: memory to load, disk space to store, energy consumption, updates)	Low latency requirement, simple tasks, no/unreliable internet connection, privacy concern, security, reduce server costs

ML Models: Bigger, Better, Slower



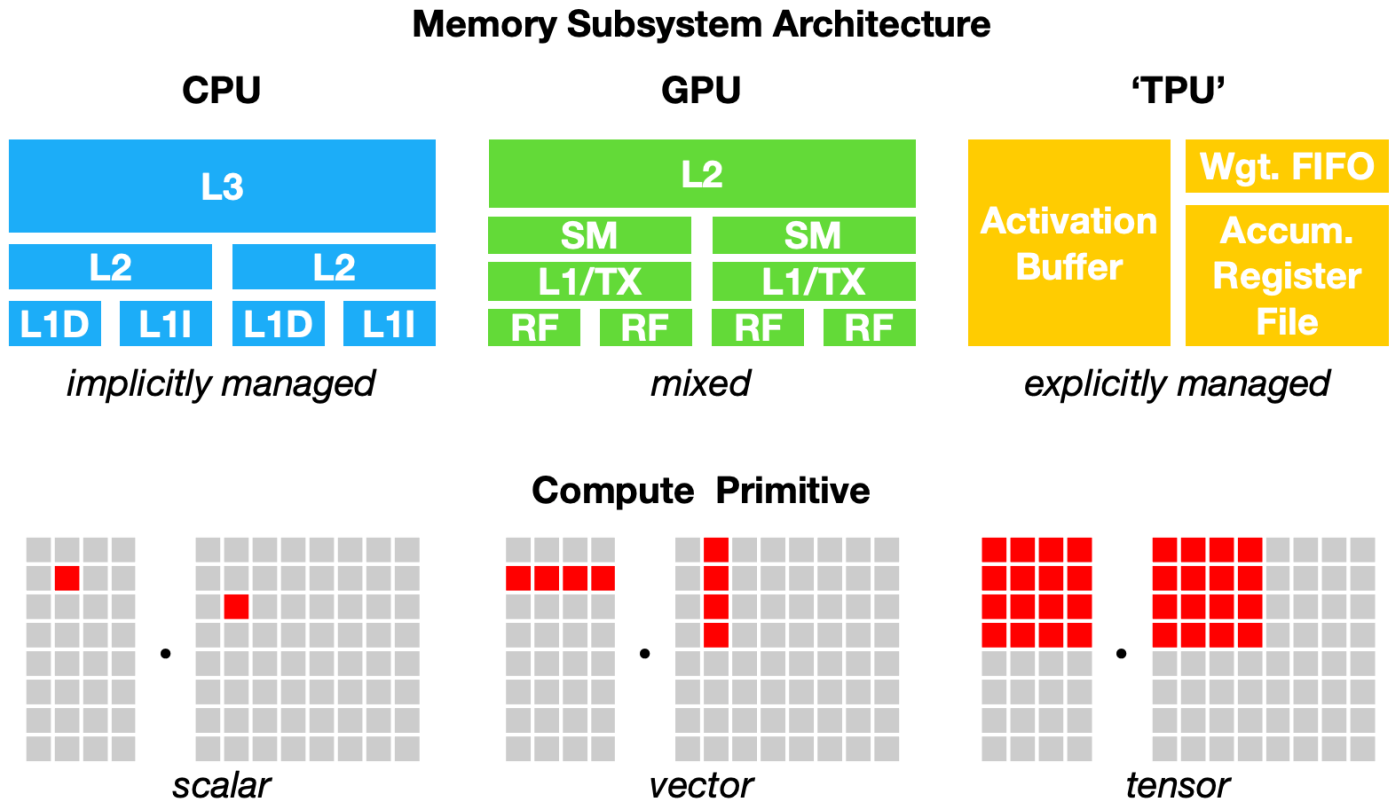
ML on Edge Devices: Solutions

- More powerful hardware
  - Use less energy
  - Stronger battery
  - Faster (more specific) hardware
- Smaller models (compression)
  - Quantization
  - Knowledge distillation
  - Pruning
- Faster models
  - Parallelizable operations (vectorization)
  - Software-hardware codesign (c cores → use n * c layers)
  - Compiled code for hardware

Efficiency techniques lecture

Specialized Hardware

- Different processors are optimized for different operations



Chen et al., 2018

Growing Complexity for Hardware Optimization

- CPUs, GPUs, TPUs, FPGAs, ASICs

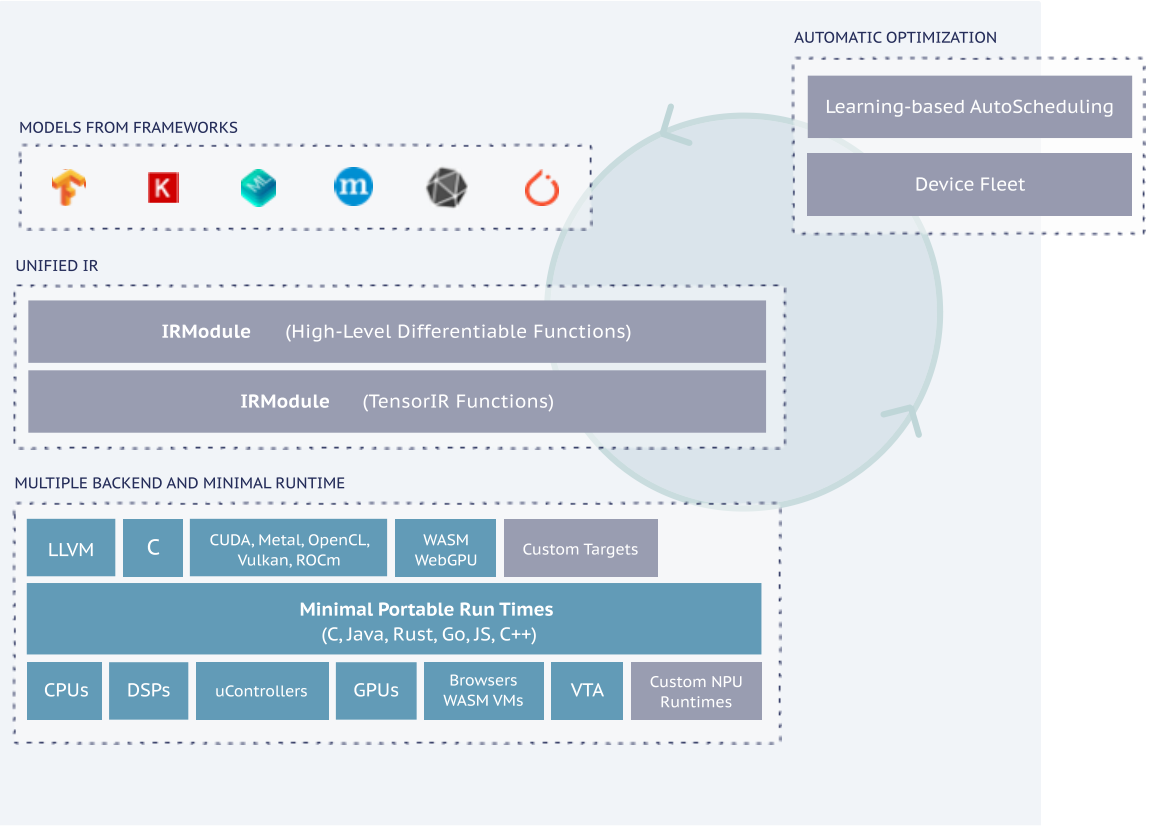
Hardware Optimization/Specialization

- Framework developers:
  - Offer support across a narrow range of server-class hardware
    - E.g. Hugging Face Optimum
- Hardware vendors:
  - Offer their own SDK/kernel libraries for a narrow range of frameworks (e.g. CUDA from Nvidia)

- Danger: Vendor/hardware lock-in
- Solution: Decouple frameworks and hardware
  - Frameworks → intermediate representations (IR; e.g. model computation graphs) → Hardware

Apache TVM (Tensor Virtual Machine)

- Automatic compilation to run on different hardware



Model Serving

Storing ML Models

- ONNX (Open Neural Network Exchange)
  - Open format for storing and exchanging ML models
    - Works with all big frameworks
  - Saves operations run during a forward pass
  - Enables interoperability between different deep learning frameworks
  - Examples for PyTorch (1, 2) and TensorFlow

```
import torch.onnx
import torchvision
sample_input = torch.randn(1, 3, 224, 224)
model = torchvision.models.alexnet(pretrained=True)
torch.onnx.export(model, sample_input, “alexnet.onnx”)
```

Serving in the Cloud

- Multitude of tools (blog post presenting some options)
  - BentoML, TensorFlow Serving, TorchServe, KServe (for Kubernetes), Triton Inference Server, Ray Serve
- Open-source
  - Some come with a paid cloud offering
  - Usually works with all big cloud providers
- Cloud providers
  - Azure ML, Azure Databricks, Google Cloud, Amazon SageMaker
  - Vendor lock-in
  - Usually more expensive
    - But less development and maintenance cost
  - Very easy to set up
  - Optimized for specific cloud provider's infrastructure

Serving in the Cloud: Vertex AI Example

Save model locally

```python
from transformers import AutoTokenizer, AutoModelForSequenceClassification

model_name = "distilbert-base-uncased-finetuned-sst-2-english"

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name)

model.save_pretrained("./model")
tokenizer.save_pretrained("./model")
```

Serving in the Cloud: Vertex AI Example

Wrapping model into HTTP-Endpoint

```python
def predict(self, instances: List[Union[str, Dict[str, Any]]]) -> List[Dict[str, Any]]:
    # Ensure model is loaded
    if not self.model_loaded:
        logger.warning("Model not loaded, attempting to load now...")
        self._load_model()

    if not self.model_loaded:
        raise RuntimeError("Model failed to load")

    logger.info(f"Processing {len(instances)} instances")
    results = []

    for idx, instance in enumerate(instances):
        try:
            # Tokenize input
            inputs = self.tokenizer(
                text,
                return_tensors="pt",
                truncation=True,
                padding=True,
                max_length=512
            )

            # Get predictions
            with self.torch.no_grad():
                outputs = self.model(**inputs)
                probabilities = self.torch.nn.functional.softmax(outputs.logits, dim=-1)
                predicted_class = self.torch.argmax(probabilities, dim=-1).item()
```

Serving in the Cloud: Vertex AI Example

Building custom docker image that contains all the required libraries (and optionally the trained model)

```
FROM python:3.10-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    curl \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy model files and predictor
COPY model/ /app/model/
COPY predictor.py /app/predictor.py

EXPOSE 8080

ENV AIP_HTTP_PORT=8080
ENV PYTHONUNBUFFERED=1

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=120s --retries=3 \
    CMD curl -f http://localhost:8080/health || exit 1

# Run the FastAPI app
CMD ["python", "predictor.py"]
```

Serving in the Cloud: Vertex AI Example

Uploading model to model registry

```
model = aiplatform.Model.upload(
    display_name="sentiment-analyzer",
    description="HuggingFace DistilBERT sentiment analysis model",
    serving_container_image_uri=IMAGE_URI,
    serving_container_predict_route="/predict",
    serving_container_health_route="/health",
)
```

Running model for batch prediction (only pay per use)

```
model = aiplatform.Model(model_name=MODEL_ID)

batch_prediction_job = model.batch_predict(
    job_display_name="sentiment-analysis-batch",
    gcs_source=[f"gs://{BUCKET_NAME}/batch-input/batch_input.jsonl"],
    gcs_destination_prefix=f"gs://{BUCKET_NAME}/batch-output/",
    machine_type="n1-standard-4",
    starting_replica_count=1,
    max_replica_count=1,
)
```

Serving in the Cloud: Vertex AI Example

Create endpoint for online prediction

```
model = aiplatform.Model(model_name=MODEL_ID)

endpoint = aiplatform.Endpoint.create(
    display_name="sentiment-analyzer-endpoint",
    project=PROJECT_ID,
    location=REGION,
)
```

Deploy model on endpoint (pay per hour)

```
endpoint.deploy(
    model=model,
    deployed_model_display_name="sentiment-analyzer-v1",
    machine_type="n1-standard-2",
    min_replica_count=1,
    max_replica_count=1,
    traffic_percentage=100,
)
```

Use model for online prediction

```
prediction = endpoint.predict(instances=[sentence])
```

Serving in the Browser

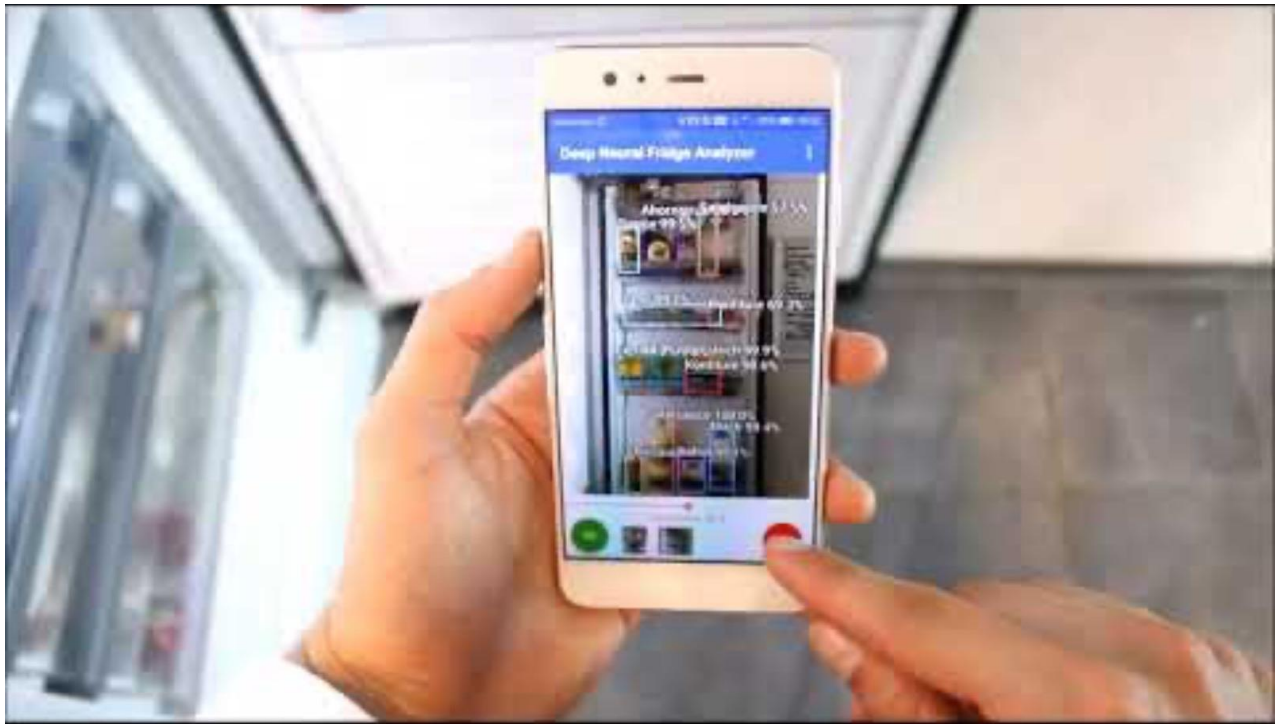
- The Promise: ”Write once, run anywhere”
- Compile model to JavaScript
- Run it in the browser of your users
  - TensorFlow.js
  - Brain.js (GitHub)
  - Uses GPUs when available
- Not widely used
- Several projects discontinued

Serving on the Phone

- Apple
  - APIs: Use Apple's models
  - Core ML: Convert your PyTorch/TensorFlow models to Core ML
- Google's ML Kit: On-device models for Android & iOS
- TensorFlow Lite: Deploy your models on Android & iOS
- Challenges
  - Updating a model means publishing a new version of the app
    - Takes time until all users have updated
  - Heterogeneous hardware on different phones

Serving on the Phone in 2018
Deep Neural Fridge Analyzer

- TensorFlow Mobile
- https://www.youtube.com/watch?v=Cd_plY6fDCY



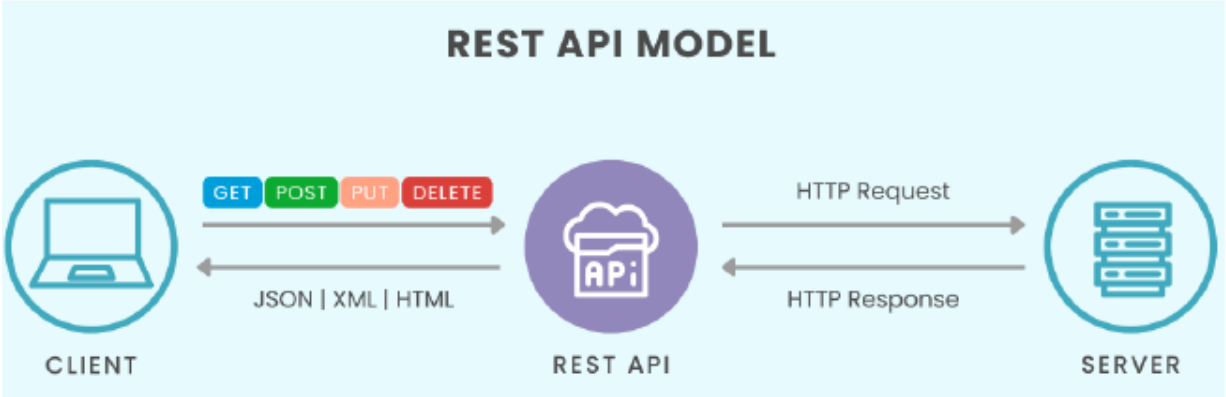
PyTorch on Edge Devices

- PyTorch Edge ecosystem to run PyTorch on edge devices (mobile phones, wearables, AR/VR/MR and embedded devices)
- ExecuTorch: On-device PyTorch inference for edge devices
  - For example: Llama-3.2 1B/3B have been quantized and optimized to run with ExecuTorch

APIs

What is REST?

- REST = Representational state transfer
- Architectural style focused on independent components and scalability
- RESTful applications adhere to REST principles
  - But term mostly used for RESTful APIs/REST APIs (HTTP interfaces)



REST Architectural Constraints

- Client-server architecture
  - Separation of concerns (UI vs. computation vs. data)
- Statelessness
  - No session information on the server (but sent with every request)
- Cacheability
  - Responses declare if they can be cached
- Layered system
  - Can’t tell if talking to proxy, load balancer, or server
- Uniform interface
  - Use URIs to identify resources
  - Messages include all relevant data

REST Requests

- Example request:
  1. The client makes a request to create, modify, or delete a resource on the server
  2. The request contains the resource endpoint and may also include additional parameters
  3. The server responds, returning the entire resource to the client once the operation is complete
  4. The response contains data in JSON format and status codes

REST Operations ("HTTP Request Verbs")

- GET: Read or retrieve item data
  - Example: `GET http://www.example.com/customers/12345/orders` (*Show me the orders of customer with ID 12345*)
  - Response: 200 (OK), 404 (Not Found)
- POST: Create new item
  - Response: 201 (Created), 404, 409 (Conflict) ← if already exists
- PUT: Update (entire) existing item
  - Response: 200, 204 (No Content) ← successful, 404
- PATCH: Modify part of an existing item
  - Response: 200, 204, 404
- DELETE: Remove an item
  - Response: 200, 404

Pros and Cons

- Pros
  - Interoperability with other applications
  - Flexibility of data format and programming language
  - Scalability
  - Security via access tokens
  - Extremely easy to use, plenty of tools available
- Cons
  - Some applications are hard to split into microservices
  - Large state hard to implement (or requests must ship a lot of data)
  - No failure handling if requests get lost/high latency (how long should a client wait?)

Converting Python to HTTP

- FastAPI
  - Build HTTP API with Python
  - Uses Python type hints
- Ray
  - Serving framework
  - Can be combined with FastAPI
    - Use FastAPI to parse/validate HTTP requests
    - Use Ray Serve to scale up deployment

```python
from typing import Union

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```

Ray Serve

- Can create HTTP as well
- Example GET request with Hugging Face model
- End-to-end example with a summarization model
  - Develop locally
  - Deploy to cloud

```python
import requests
from starlette.requests import Request
from typing import Dict

from transformers import pipeline

from ray import serve


# 1: Wrap the pretrained sentiment analysis model in a Serve deployment.
@serve.deployment(route_prefix="/")
class SentimentAnalysisDeployment:
    def __init__(self):
        self._model = pipeline("sentiment-analysis")

    def __call__(self, request: Request) -> Dict:
        return self._model(request.query_params["text"])[0]


# 2: Deploy the deployment.
serve.run(SentimentAnalysisDeployment.bind())

# 3: Query the deployment and print the result.
print(
    requests.get(
        "http://localhost:8000/", params={"text": "Ray Serve is great!"}
    ).json()
)
# {'label': 'POSITIVE', 'score': 0.9998476505279541}
```

gRPC

- Alternative to REST APIs
- RPC = remote procedure call
- Client directly calls functions on the server
  - Mimics local procedure call over HTTP
- Initially created by Google (hence the “g”)
- https://grpc.io/
- Useful for real-time applications (bidirectional streaming)

REST vs. gRPC

- Communication model
  - REST: Single request, single response
  - gRPC: All combinations of single/multiple requests with single/multiple responses, allows bidirectional streaming and integrated authentication
- Callable operations
  - REST: Everything with an HTTP endpoint (one of the HTTP request verbs)
  - gRPC: Any method on the server (much tighter coupling with server's implementation)
- Data exchange format
  - REST: JSON
  - gRPC: Protocol Buffers/JSON
- Trade-off
  - REST has better decoupling, is easier to implement and maintain
  - gRPC is faster and useful for data streaming

https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/