Оптимизация моделей и конвертация в ONNX

Лекция №3: От исследовательского прототипа к промышленному артефакту

От исследования к эксплуатации

Обученная модель — это не готовый продукт, а **исследовательский результат**. Переход к эксплуатации требует превращения модели в воспроизводимый артефакт.

Исследование

Гибкость, эксперименты, быстрые итерации

Эксплуатация

Детерминированность, воспроизводимость, стабильность

Модель в продакшене должна обрабатывать тысячи запросов в секунду с гарантированной точностью.

Артефакт модели

Артефакт — это формализованное представление модели, включающее не только веса, но и полный контекст исполнения:

$$ModelArtifact = \{W, f(x;\theta), \text{preprocess}, \text{postprocess}, \text{deps}, \text{config}\}$$

Веса W

Матрицы весов и смещений

Граф f(x; θ)

Вычислительный граф модели

Препроцессинг

Подготовка входных данных

Зависимости

Версии библиотек и драйверов

Зачем нужна оптимизация

После обучения модель избыточна: высокая точность параметров, неоптимальные структуры данных, избыточные вычисления.

1

Размер модели

Уменьшить для быстрой загрузки

2

Время инференса

Сократить для высокой пропускной способности

3

Энергопотребление

Снизить для встроенных устройств

Функция компромисса между критериями:

$$J = \alpha \cdot \text{Accuracy} - \beta \cdot \text{Latency} - \gamma \cdot \text{Memory}$$

Проблема воспроизводимости

Модель зависит от множества факторов: версии Python, порядка операций в NumPy, состояния CUDA. Эти различия разрушают воспроизводимость в продакшене.

> Критерий эквивалентности исполнения:
>
> $$f_{\text{prod}}(x;\theta) = f_{\text{train}}(x;\theta), \quad \forall x \in D_{\text{val}}$$

Решение: фиксация среды через контейнеры (Docker) и дескрипторы зависимостей (requirements.txt, environment.yaml).

Этапы подготовки к эксплуатации

1

Фиксация графа

Превращение динамической модели в статическое представление

2

Оптимизация

Сокращение параметров, снижение точности (FP16, INT8)

3

Конвертация в ONNX

Экспорт в переносимый формат

4

Проверка эквивалентности

Тестирование числового совпадения

5

Интеграция в пайплайн

Упаковка в CI/CD систему

От гибкости к детерминированности

PyTorch

Создан для исследований

Пространство гипотез

Вопрос: "А что если?"

ONNX

Создан для эксплуатации

Пространство контрактов

Вопрос: "Будет ли работать?"

Модель становится артефактом только тогда, когда она перестаёт зависеть от исследователя и становится частью инфраструктуры.

Динамический граф PyTorch

PyTorch создаёт вычислительный граф «на лету» при каждом выполнении. Граф не фиксируется статически — он формируется интерпретатором Python во время исполнения.

```
import torch
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x * 2
z = y.mean()
z.backward()
```

Autograd Graph создаётся и уничтожается при каждом вызове. Это обеспечивает гибкость, но делает граф недетерминированным.

Императивный vs Декларативный граф

Императивный (PyTorch)

Вычисления определяются во время исполнения. Можно использовать if-else, циклы, любые конструкции Python.

Декларативный (ONNX)

Вычисления описаны заранее и могут быть скомпилированы вне контекста Python.

Эксплуатация требует декларативного графа для воспроизводимости и оптимизации.

TorchScript: статическая форма

TorchScript — промежуточное представление, превращающее динамический граф в статическое описание, независимое от Python.

Требование эквивалентности:

$$\forall x \in D: \quad f_{\text{TorchScript}}(x;\theta) = f_{\text{PyTorch}}(x;\theta)$$

Трейсинг

Запись операций при конкретном прогоне

Скриптинг

Анализ кода и компиляция в IR

Трейсинг (Tracing)

Трейсинг записывает последовательность операций при конкретном входе:

```
import torch

class SimpleModel(torch.nn.Module):
    def forward(self, x):
        return x * 2 + 3

model = SimpleModel()
traced = torch.jit.trace(model, torch.ones(1))
traced.save("model_traced.pt")
```

Ограничение: Трейсинг не подходит для моделей с динамическими ветвлениями (if-else). Он зафиксирует только одну ветвь.

Скриптинг (Scripting)

Скриптинг анализирует код на уровне Python и компилирует его, сохраняя логику ветвлений и циклов:

```
class ThresholdModel(torch.nn.Module):
    def forward(self, x):
        if x.sum() > 0:
            return x * 2
        else:
            return x - 2

scripted = torch.jit.script(ThresholdModel())
scripted.save("model_scripted.pt")
```

Скриптинг строит абстрактное синтаксическое дерево и преобразует его в статический граф, сохраняя корректность при любом входе.

Трейсинг vs Скриптинг

Трейсинг

- Точно воспроизводит операции
- Теряет логические ветви
- Идеален для CNN
- Простой в использовании

Скриптинг

- Сохраняет логику ветвлений
- Может быть несовместим с произвольным Python-кодом
- Необходим для RNN, трансформеров
- Требует больше внимания

На практике часто комбинируют оба подхода для разных частей модели.

JIT-компиляция и оптимизации

TorchScript JIT анализирует граф и применяет оптимизации:

Dead Code Elimination

Устранение неиспользуемого кода

Constant Folding

Свёртка констант

Operation Fusion

Слияние операций

Модель сериализуется в бинарный формат .pt и может быть загружена в C++ через libtorch без Python.

Проверка эквивалентности

Обязательный шаг: тестирование совпадения поведения оригинальной и TorchScript-модели.

```
model = MyModel().eval()
scripted = torch.jit.script(model)
x = torch.randn(8, 3, 224, 224)

with torch.no_grad():
    y1 = model(x)
    y2 = scripted(x)

diff = (y1 - y2).abs().max()
print("Max difference:", diff.item())
```

$$|f_{\text{orig}}(x) - f_{\text{scripted}}(x)|_2 < \epsilon$$

Типичный порог: ε = 10-5

TorchScript как первый шаг

TorchScript — промежуточный слой между исследовательской моделью и экспортируемым артефактом.

Исследовательская модель

Динамический Python-код

TorchScript

Статический граф, независимый от Python

ONNX

Универсальный формат для любых платформ

TorchScript используется как в экосистеме PyTorch (через libtorch), так и как источник для конвертации в ONNX.

Узкие места инференса

Производительность ограничивается несколькими факторами:

CPU–GPU передача

Копирование данных через шину PCIe тратит время на синхронизацию

Плохо параллелизуемые операции

Softmax, sigmoid, reshape создают точки синхронизации

Memory Bandwidth

Скорость доступа к памяти ограничивает производительность

Цель оптимизации: максимизировать FLOPs/s при минимальных обращениях к памяти.

Оптимизация batch size

Размер батча определяет баланс между латентностью и пропускной способностью:

$$\text{Throughput}(n) = \frac{n}{t_p(n)}$$

$$\text{Latency}(n) = \frac{t_p(n)}{n}$$

Batch size = 1: минимальная задержка, неэффективное использование GPU

Большой batch: высокая эффективность, но увеличенная задержка

Оптимальное значение n* определяется эмпирически через профилирование.

Смешанная точность (Mixed Precision)

Использование форматов с меньшей разрядностью ускоряет вычисления и снижает потребление памяти.

FP32

1 бит знака, 8 бит экспоненты, 23 бита мантиссы

FP16

1 бит знака, 5 бит экспоненты, 10 бит мантиссы

BF16

Та же экспонента что FP32, укороченная мантисса

FP16 даёт 2× ускорение и 2× экономию памяти, но может вызвать потерю численной стабильности.

Automatic Mixed Precision (AMP)

PyTorch автоматически выбирает, какие операции выполнять в FP16, а какие в FP32:

```
import torch

model = MyModel().eval().to('cuda')
x = torch.randn(16, 3, 224, 224, device='cuda')

with torch.autocast(device_type='cuda', dtype=torch.float16):
    y = model(x)
```

AMP сохраняет FP32 для нормализации и экспоненциальных функций, используя FP16 для матричных операций.

Эффект: ускорение в 1.5–2.5× без потери точности.

Ошибка округления в FP16

При переходе к FP16 возникает ошибка округления:

$$x' = x(1 + \delta), \quad |\delta| \leq \epsilon$$

Машинная точность:

- FP32: ε ≈ 1.19 × 10-7
- FP16: ε ≈ 4.88 × 10-4

При правильной реализации ошибки остаются в пределах допуска, но могут суммироваться в длинных вычислительных цепочках.

Квантизация (Quantization)

Перевод весов и активаций в целочисленный формат (INT8) для уменьшения размера и ускорения.

Масштаб и смещение:

$$s = \frac{x_{\max} - x_{\min}}{2^b - 1}, \quad z = \text{round}\left(\frac{-x_{\min}}{s}\right)$$

Квантизованное значение:

$$x_q = \text{round}\left(\frac{x}{s}\right) + z$$

Деквантизация:

$$x \approx s(x_q - z)$$

Типы квантизации

1

Dynamic Quantization

Веса в INT8, активации в FP32. Масштабы пересчитываются во время инференса.

2

Static Quantization

Веса и активации квантизованы. Требует калибровки на данных.

3

Quantization Aware Training

Квантизация имитируется во время обучения для адаптации модели.

Пример динамической квантизации

```
import torch
from torch import nn
from torch.quantization import quantize_dynamic

model_fp32 = nn.Linear(512, 512)
model_int8 = quantize_dynamic(
    model_fp32,
    {nn.Linear},
    dtype=torch.qint8
)
```

Результат: размер модели уменьшается в ~4 раза, инференс на CPU ускоряется в 2–3 раза. Потеря точности обычно не превышает 1–2%.

Анализ ошибок квантизации

Ошибка квантизации — разница между исходным и восстановленным значениями:

$$E = \frac{1}{N} \sum_i (x_i - \hat{x}_i)^2$$

Дисперсия ошибки при равномерной квантизации:

$$\sigma_E^2 \approx \frac{s^2}{12} = \frac{(x_{\max} - x_{\min})^2}{12(2^b - 1)^2}$$

При переходе с FP32 на INT8 ошибка возрастает в ~256 раз, но абсолютные значения остаются малыми при ограниченном диапазоне весов.

Прореживание весов (Pruning)

Обнуление малозначимых весов для уменьшения количества активных параметров:

$$\min_{\theta'} L(\theta', D) \quad \text{при условии} \quad |\theta'|_0 \leq k$$

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

prune.l1_unstructured(
    model.fc,
    name='weight',
    amount=0.3
)
```

Обнуляет 30% весов с наименьшей абсолютной величиной. Эффективно только при поддержке sparse-операций в библиотеках (TensorRT, ONNX Runtime).

Дистилляция знаний

Обучение компактной модели (student) имитировать поведение большой модели (teacher):

$$L = (1 - \lambda) L_{\text{hard}} + \lambda T^2 \cdot \text{KL}(p_T \| q_T)$$

1

Teacher

Большая модель с высокой точностью

2

Дистилляция

Передача «мягких» предсказаний

3

Student

Компактная модель (в 3–5 раз меньше)

Проверка после оптимизации

После любой оптимизации необходимо проверить корректность модели:

$$\Delta = \frac{|f_{\text{opt}}(x) - f_{\text{orig}}(x)|_2}{|f_{\text{orig}}(x)|_2}$$

Критерии эквивалентности:

- $\Delta < 10\text{-}3$
- Метрики качества снижаются не более чем на 1%

В промышленной практике создаются регрессионные тесты: каждая новая версия сравнивается с эталонной на фиксированном наборе входов.

Философия ONNX

ONNX (Open Neural Network Exchange) — открытый формат представления нейронных сетей, обеспечивающий совместимость между фреймворками.

> Описать вычисления модели не в терминах кода фреймворка, а в терминах базовых операторов линейной алгебры

ONNX играет роль промежуточного представления (IR), отделяя описание вычислений от их реализации.

Структура ONNX-графа

Graph

Контейнер, описывающий структуру вычислений

Node

Отдельная операция (Conv, ReLU, MatMul)

Tensor

Данные (веса, смещения) или промежуточные результаты

Attribute

Параметры операторов (размер ядра, шаг)

Opset

Версия спецификации ONNX

$$f_{\text{ONNX}}(x) = o_k \circ o_{k-1} \circ \ldots \circ o_1(x)$$

Версия Opset

Opset определяет набор поддерживаемых операторов и их сигнатуры. Несовпадение версий приводит к несовместимости.

Opset 11

Новые параметры ConvTranspose

Opset 13

Изменено поведение ReduceMean

Opset 15+

Расширенная поддержка трансформеров

Номер opset фиксируется в конфигурации проекта и становится частью метаданных артефакта.

Экспорт модели в ONNX

```
import torch
import torch.onnx

model = MyModel().eval()
dummy_input = torch.randn(1, 3, 224, 224)

torch.onnx.export(
 model,
 dummy_input,
 "model.onnx",
 export_params=True,
 opset_version=13,
 do_constant_folding=True,
 input_names=["input"],
 output_names=["output"],
 dynamic_axes={"input": {0: "batch"},
 "output": {0: "batch"}}
)
```

Параметры экспорта

export_params

Сохранить веса модели в файл

do_constant_folding

Вычислить константные операции заранее

dynamic_axes

Оси, которые могут изменяться (например, batch size)

opset_version

Версия спецификации ONNX

Типичные ошибки при экспорте

1 **Неподдерживаемые операторы**

Не все операции PyTorch имеют эквивалент в ONNX. Требуется реализация замены.

2 **Динамические размеры**

Циклы и условия, зависящие от входных данных, фиксируют только конкретный путь.

3 **Несовместимость opset**

Новая версия может изменить сигнатуру операторов.

4 **Численная точность**

Разные реализации операторов дают разные результаты.

Проверка эквивалентности ONNX

```
import onnxruntime as ort
import numpy as np

ort_session = ort.InferenceSession("model.onnx")

x = torch.randn(1, 3, 224, 224)
torch_out = model(x)

ort_out = ort_session.run(
    None,
    {"input": x.numpy()}
)

np.testing.assert_allclose(
    torch_out.numpy(),
    ort_out[0],
    rtol=1e-03,
    atol=1e-05
)
```

$$\Delta = \frac{|f_{\text{torch}}(x) - f_{\text{onnx}}(x)|_2}{|f_{\text{torch}}(x)|_2} < \epsilon$$

Инженерные правила экспорта

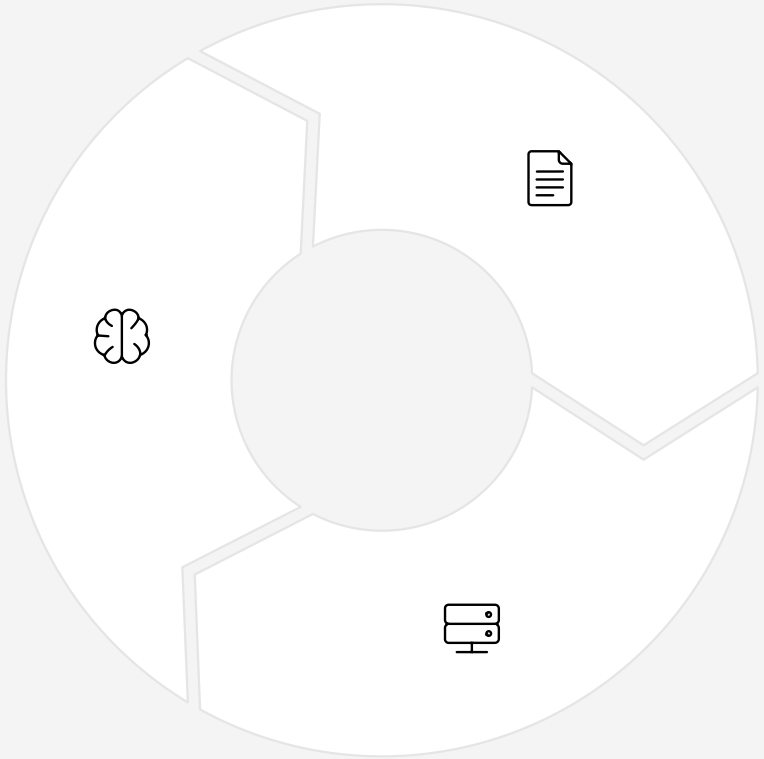
- Фиксируйте версию opset как часть конфигурации артефакта
- Проверяйте зависимость от внешнего кода (NumPy, кастомные функции)
- Учитывайте формат входных данных (NCHW vs NHWC)
- Сохраняйте препроцессинг отдельно от модели
- Проверяйте типы тензоров (FP32, FP16, INT8)

ONNX как контракт

ONNX фиксирует вычислительный граф в декларативном виде, обеспечивая воспроизводимость на любом оборудовании.

Обучение

Исследовательская фаза



ONNX

Единый контракт

Инференс

Эксплуатационная фаза

Функция f(x; θ), определённая при обучении, воспроизводится точно при инференсе.

ONNX как промежуточный формат

ONNX — не конечная форма, а источник для оптимизации под конкретное оборудование:

1	2	3
ONNX	Оптимизация	Исполнение
Универсальный граф	TensorRT, OpenVINO, CoreML	Нативный код для устройства

ONNX — аналог байт-кода, ONNX Runtime — виртуальная машина для нейронных сетей.

ONNX Runtime

Открытый движок инференса от Microsoft для исполнения ONNX-моделей с высокой производительностью на различных платформах.

1. **Загрузка и валидация**
   Проверка типов и совместимости операторов
2. **Построение плана**
   Распределение узлов по исполнительным устройствам
3. **Оптимизация графа**
   Применение трансформаций
4. **Исполнение**
   Вычисление с профилированием

Уровни оптимизации графа

1

Basic

Устранение дубликатов, constant folding

Extended

Слияние операторов, устранение неиспользуемых выходов

3

All

Аппаратно-специфические преобразования, CUDA-ядра

$$G_{\text{opt}} = \Phi(G_{\text{orig}})$$

Преобразования сохраняют функциональную эквивалентность.

Профилирование производительности

```
import onnxruntime as ort
import numpy as np
import time

session = ort.InferenceSession(
 "model.onnx",
 providers=["CUDAExecutionProvider"]
)
x = np.random.randn(1, 3, 224, 224).astype(np.float32)

n_runs = 100
start = time.time()
for _ in range(n_runs):
 _ = session.run(None, {"input": x})
end = time.time()

latency = (end - start) / n_runs * 1000
throughput = n_runs / (end - start)
```

Ключевые метрики

2.7

Latency (мс)

Время выполнения одной итерации

$$L = \frac{T_{\text{end}} - T_{\text{start}}}{n_{\text{runs}}}$$

370

Throughput (инф/с)

Количество итераций в единицу времени

$$T = \frac{n_{\text{runs}}}{T_{\text{end}} - T_{\text{start}}}$$

Эти значения фиксируются до оптимизации для последующего сравнения.

Проверка точности после оптимизации

Любая оптимизация должна сохранять точность. Метрики:

Абсолютная ошибка

$$E_{\text{abs}} = \frac{1}{N} \sum_{i} |y_i - \hat{y}_i|$$

Относительная ошибка

$$E_{\text{rel}} = \frac{|y - \hat{y}|_2}{|y|_2}$$

Критерий эквивалентности: Erel < 10-3

При использовании FP16/INT8 вводится метрика совпадения метрик качества (например, разница в Top-1 Accuracy не более 1%).

Методы оптимизации в ONNX Runtime

Fuse Operations

Объединение последовательных операций (Conv → BatchNorm → ReLU) в одну

Quantization

Пост-тренинговая квантизация (PTQ) и QAT

Graph Transformation API

Пользовательские оптимизации графа

Hardware Backends

TensorRT, OpenVINO, CoreML, DirectML

Фьюзинг операций

Объединение нескольких операций в одну для уменьшения накладных расходов:

> Conv → BatchNorm → ReLU → FusedConvRelu

Математически:

$$f(x) = \max(0, W * x + b)$$

Всё вычисляется за один проход без промежуточных записей в память, что снижает количество обращений к памяти и увеличивает пропускную способность GPU.

Квантизация в ONNX Runtime

```
from onnxruntime.quantization import (
    quantize_dynamic,
    QuantType
)

quant_model = quantize_dynamic(
    "model.onnx",
    "model_int8.onnx",
    weight_type=QuantType.QInt8
)
```

Заменяет веса на INT8 и добавляет масштабные коэффициенты. Потеря точности минимальна, выигрыш в производительности: 2–4× на CPU.

Hardware-specific backends

TensorRT

CUDA-оптимизированные кернелы для NVIDIA GPU

OpenVINO

Оптимизация под Intel CPU и ускорители Movidius

CoreML

Для iOS-устройств Apple

DirectML

Для Windows-платформ

Эти бекенды превращают ONNX-граф в нативный исполняемый код.

Профилирование графа

Анализ горячих участков — узлов, занимающих большую часть времени:

```
session.start_profiling("profile.json")
session.run(None, {"input": x})
session.end_profiling()
```

JSON содержит список операторов, их время исполнения и количество вызовов. Визуализация в Chrome Tracing.

$$T_{\text{total}} = \sum_i T_i$$

Цель: минимизировать сумму при сохранении функциональной эквивалентности.

Матрица совместимости

Модель должна стабильно работать на разных устройствах и с разными версиями драйверов:

Параметр	Значение
ONNX Runtime	1.18.0
CUDA/cuDNN	12.1 / 8.9
Opset Version	13
GPU Driver	535.104.05
Платформа	Linux x86_64

Любые изменения требуют регрессионного тестирования.

Triton Inference Server

Сервер инференса от NVIDIA для масштабных производственных систем. Поддерживает ONNX, TensorRT, PyTorch и другие форматы.

Конфигурация модели (config.pbtxt):

```
name: "model"
platform: "onnxruntime_onnx"
max_batch_size: 16
input [
 {
 name: "input"
 data_type: TYPE_FP32
 dims: [3, 224, 224]
 }
]
output [
 {
 name: "output"
 data_type: TYPE_FP32
 dims: [1000]
 }
]
```

Цикл проверки оптимизации

Baseline

Измерить исходные метрики

Optimize

Применить оптимизации

Profile

Собрать метрики производительности

Validate

Проверить эквивалентность

Regression

Зафиксировать в CI

Многоцелевая оптимизация

Каждая оптимизация решает задачу минимизации:

$$\min_{\phi \in \mathcal{T}} [\alpha \cdot \text{Latency}(\phi) + \beta \cdot \text{Memory}(\phi) - \gamma \cdot \text{Accuracy}(\phi)]$$

где ϕ — комбинация оптимизаций, 𝒯 — множество допустимых преобразований.

Оптимизация успешна, если точность падает не более чем на $\delta = 1\%$, а скорость увеличивается хотя бы в k раз (для CV: $k = 2–5\times$, для NLP: $k = 1.3–2\times$).

Промышленный пайплайн

Модель превращается в элемент промышленного конвейера — замкнутого цикла CI/CD для машинного обучения:

1. **Train**
   Обучение
2. **Optimize**
   Оптимизация
3. **Export**
   Конверсия
4. **Validate**
   Валидация
5. **Package**
   Упаковка
6. **Deploy**
   Развертывание

Артефакт модели в CI/CD

Артефакт — управляемый программный объект, подобный бинарному пакету:

$$A = (f, W, \theta, C, V, M)$$

f — граф

Вычислительный граф

W — веса

Параметры модели

θ — гиперпараметры

Параметры оптимизации

C — конфигурация

Окружение

V — версия

Opset

M — метрики

Точность и производительность

Управление версиями артефактов

Системы управления моделями: MLflow, Weights & Biases, DVC, Model Registry.

Метаданные артефакта:

- Имя и версия (model_v3.1.onnx)
- Hash-коммит
- Номер opset
- Дата оптимизации
- Параметры квантизации
- Метрики точности и производительности

Артефакты хранятся в репозиториях (Artifactory, S3, Google Artifact Registry) с checksum и подписью.

Автоматизация тестов

1 **Unit-тест**

Проверка входов, выходов, форм тензоров, типов данных

2 **Regression-тест**

Сравнение с эталонной версией на фиксированных примерах

3 **Performance-тест**

Измерение latency и throughput

4 **Compatibility-тест**

Запуск с разными версиями Runtime

Провал любой стадии блокирует развертывание.

Документирование окружения

Дескриптор модели фиксирует всё для воспроизводимости:

```
model_name: resnet50_opt
version: 3.1
opset: 13
framework: pytorch
onnx_runtime_version: 1.18.0
precision: fp16
quantization: dynamic
device: gpu
latency_ms: 2.7
accuracy_drop: 0.3%
hash: 8f32c9d1
```

Контейнеризация (Docker) фиксирует ОС, CUDA, cuDNN, Python и все библиотеки.

$$\text{Environment} = \{\text{OS}, \text{Python}, \text{Deps}, \text{CUDA}, \text{Drivers}\}$$

Мониторинг в эксплуатации

После развертывания модель контролируется на уровне метрик сервиса:

Время отклика

Latency для каждого запроса

Использование GPU

Загрузка вычислительных ресурсов

Ошибки обработки

Частота и типы ошибок

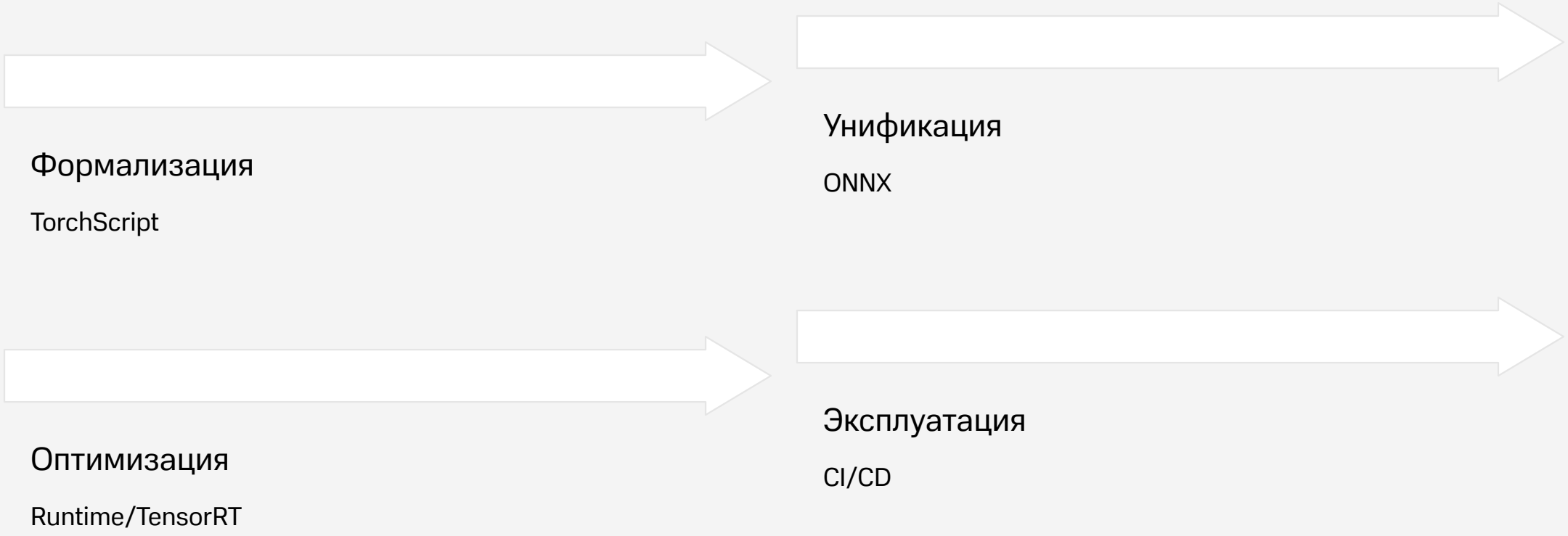
Дрейф модели

Расхождение между реальными и обучающими данными

Системы: Prometheus, Grafana, Evidently AI

Системное инженерное проектирование

Путь модели от Jupyter до продакшна — это системное инженерное проектирование:



Каждый шаг делает поведение модели детерминированным, измеримым и контролируемым.

Заключение

Мы прошли весь путь от исследовательского прототипа до промышленного артефакта.

> Уравнение воспроизводимости:
>
> $$f_{\text{train}}(x;\theta) = f_{\text{prod}}(x;\theta), \quad \forall x \in D_{\text{val}}$$

Все технологии — TorchScript, ONNX, Runtime, Triton, Docker — инструменты для обеспечения этого равенства.

Оптимизация и конвертация — не только технические процедуры, но и философия промышленного ML: качество модели определяется не только accuracy, но и воспроизводимостью, скоростью, стабильностью и управляемостью.