# СКВОЗНОЙ БЕНЧМАРКИНГ ML-СТЕКА: ОТ PYTHON-БЭКЕНДА ДО JS-ФРОНТЕНДА В ЗАДАЧЕ КЛАССИФИКАЦИИ ИЗОБРАЖЕНИЙ

*Бурчук Д.А., Ясев А.И., студенты*

*Белорусский государственный университет информатики и радиоэлектроники*
*г. Минск, Республика Беларусь*

*Жвакина А.В. – канд. техн. наук, доцент*

**Аннотация**. В работе рассмотрен сквозной бенчмаркинг производительности стека технологий для задачи классификации изображений, охватывающий этапы от инференса моделей на бэкенде до отображения результатов на веб-клиенте. Анализируются подходы к развертыванию как собственных моделей на фреймворках PyTorch и TensorFlow, так и готовых решений с использованием специализированных рантаймов. Рассматриваются следующие методы оптимизации серверной логики Python-бэкенда: асинхронность, батчинг, кэширование. В качестве методов оптимизации клиентской части на JavaScript рассматриваются работа с Canvas API, веб-воркерами, управление событиями. Цель работы – выявление узких мест и формулировка практических рекомендаций для построения высокопроизводительных ML-веб-приложений.

При выборе фреймворка для инференса кастомных нейросетевых архитектур ключевым фактором становится не только удобство разработки, но и эффективность исполнения. Исследование, сравнивающее PyTorch, TensorFlow и прочие фреймворки, демонстрирует, что нативные оптимизации и компиляция графа вычислений могут давать значительный прирост [1]. TensorFlow, с его статическим графом по умолчанию, исторически имел преимущество в продакшн-среде, однако PyTorch с режимом torch.jit.trace или torch.compile сокращает этот разрыв. Критически важным является использование аппаратных ускорителей, таких как CUDA и MPS, и высокоуровневых библиотек, таких как cuDNN либо OneDNN, что на порядки ускоряет операции линейной алгебры. Бенчмарки показывают, что разница в скорости инференса между фреймворками для одной и той же модели может достигать 15-30% в зависимости от типа операций и размера батча.

Использование готовых, предобученных моделей из репозиториев, среди которых самые популярные на данный момент – Hugging Face и TIMM, или облачных ML-сервисов заменяет задачу тренировки собственных моделей на задачу эффективного обслуживания готовых. Форматы ONNX и TensorRT кратно упрощают процесс кроссплатформенного развертывания, позволяя проводить аппаратно-зависимые оптимизации. ONNX Runtime предоставляет провайдеры для выполнения на CPU, GPU и специализированных процессорах, демонстрируя в ряде задач превосходство над нативными фреймворками за счет агрессивного фьюзинга операций и использования квантизации [2]. Для моделей из TensorFlow Serving эффективность упирается в настройку батчинга на уровне сервера и использование протокольной буферизации для сериализации вместо JSON.

Оптимизация серверной логики на Python-бэкенде с использованием современных фреймворков, таких как FastAPI и Django, требует решения проблем GIL и ввода-вывода. Выделение инференса в отдельный микросервис или использование многопроцессной модели позволяет изолировать блокирующие вычисления. Применение асинхронного кода эффективно для задач ввода-вывода, но не для исполняемых на центральном процессоре операций инференса. Реализация пула запросов для накопления и пакетной обработки входящих изображений от разных клиентов значительно повышает утилизацию GPU. Кэширование результатов предобработки изображений и часто запрашиваемых классификаций в Redis/Memcached снижает нагрузку на модель. В качестве дополнительного средства оптимизации также используется профилирование. Использование инструментов профилирования cProfile и Py-Spy позволяет выявить критические секции, такие как затратная декодировка изображений или излишнее копирование тензоров.

На клиентской стороне (в частности при использовании JavaScript) основная нагрузка связана с загрузкой и предобработкой изображений. Вынесение ресурсоемких задач, таких как предварительная нормализация и изменение размерности изображения, в Web Worker предотвращает блокировку интерфейса. Использование WebGL через библиотеки наподобие TensorFlow.js для выполнения части препроцессинга или даже легких моделей непосредственно в браузере разгружает сервер. Оптимизация обработки событий через троттлинг и дебаунсинг для таких действий, как загрузка файлов или изменение параметров классификации, снижает количество бесполезных запросов. Ленивая загрузка тяжелых JS-модулей и использование современных форматов изображений (таких как WebP) уменьшают время первой загрузки приложения.

Сквозной анализ показывает, что максимальная производительность достигается только при согласованной оптимизации всех компонентов стека. Низкая задержка обеспечивается эффективным инференсом на сервере и минимальным временем передачи данных, что требует выбора оптимального формата сериализации (Protobuf, MessagePack, Base64 JSON). Высокая пропускная способность достигается за счет батчинга на сервере и эффективного управления очередями на клиенте.

Бенчмаркинг должен проводиться на репрезентативных данных и с учетом реальных сценариев использования, а не только на изолированных метриках скорости инференса модели.

***Список использованных источников:***

1. *A Comparative Survey of PyTorch vs TensorFlow for Deep Learning: Usability, Performance, and Deployment Trade-offs [Electronic resource]. – Mode of access: https://arxiv.org/html/2508.04035v1. – Date of access: 26.12.2025.*

2. Selective Quantization Tuning for ONNX Models *[Electronic resource]. – Mode of access: https://arxiv.org/html/2507.12196v1. – Date of access: 29.12.2025.*