1. Установка зависимостей

В первой клетке документа мы установили зависимости в виртуальное окружение. Для этого был написан файл requirements.txt следующего содержания:

```
# ============================================
# PyTorch + ROCm (AMD GPU support)
# ============================================
torch==2.4.1+rocm6.1
torchvision==0.19.1+rocm6.1
torchaudio==2.4.1+rocm6.1
--extra-index-url https://download.pytorch.org/whl/rocm6.1

# ============================================
# Core ML / Data Science Libraries
# ============================================
numpy
pandas
scipy
scikit-learn
matplotlib
seaborn
opencv-python
Pillow
tqdm

# ============================================
# Jupyter environment
# ============================================
jupyterlab
notebook

# ============================================
# Deep Learning Tools / Model Deployment
# ============================================
onnx
onnxruntime-gpu
albumentations
```

2. GPU-отчёт

Далее, была выведена модель CUDA-устройства и количество VRAM.

```
 Device Name           | Total Memory (GB)
-----------------------|------------------
 AMD Radeon RX 6600    | 7.98
```

Данная лабораторная работа выполнялась на AMD Radeon RX 6600 с 8 гигабайтами VRAM. Для имитации CUDA использовался бекенд ROCm.

3. Альбументо-аугментации

В качестве базы была взята готовая модель EfficientNet-B3. Для улучшения качества модели была использована библиотека albumentations. Итоговая точность дообученной модели составила 92.42%. График представлен на рис. 1.

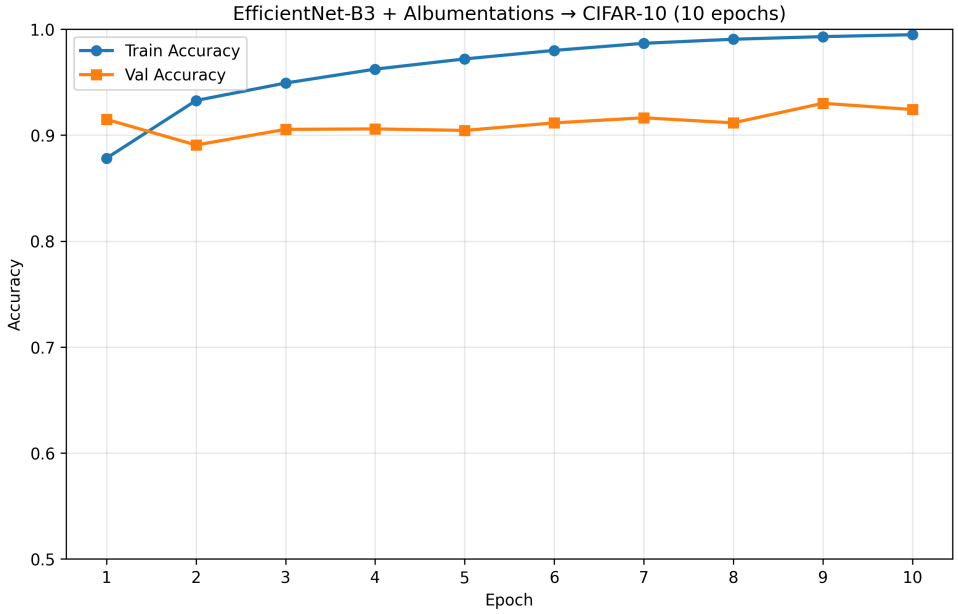

Рис. 1: График точности дообученной EfficientNet-B3

4. A/B EfficientNet

Далее тот же процесс был проделан на EfficientNet-B0. Итоговая точность составила 92.85%. График представлен на рис. 2.

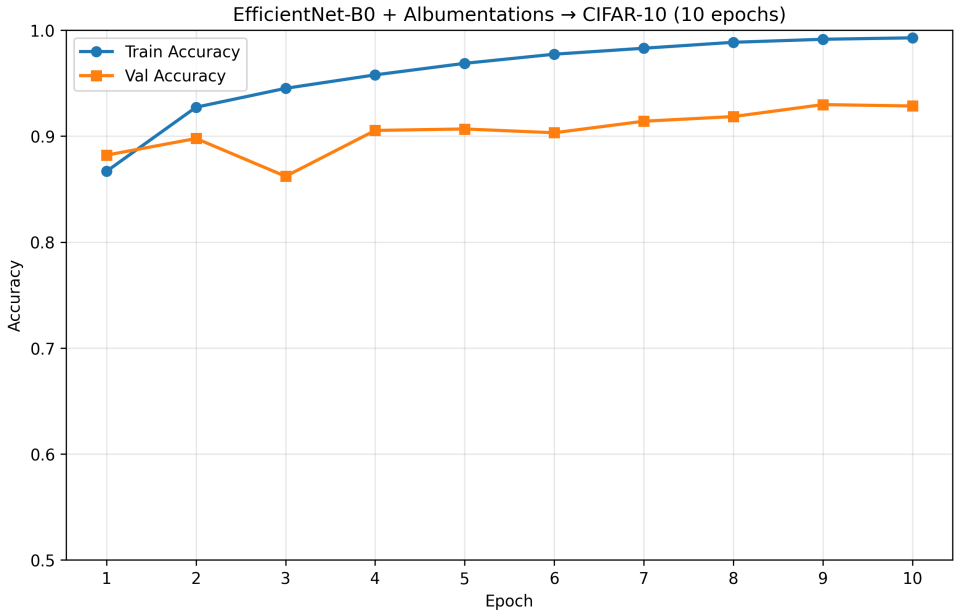

Рис. 2: График точности дообученной EfficientNet-B0

5. Early Stopping

Была добавлена клетка с ранней остановкой обучения в случае, если валидационная точность перестаёт улучшаться.

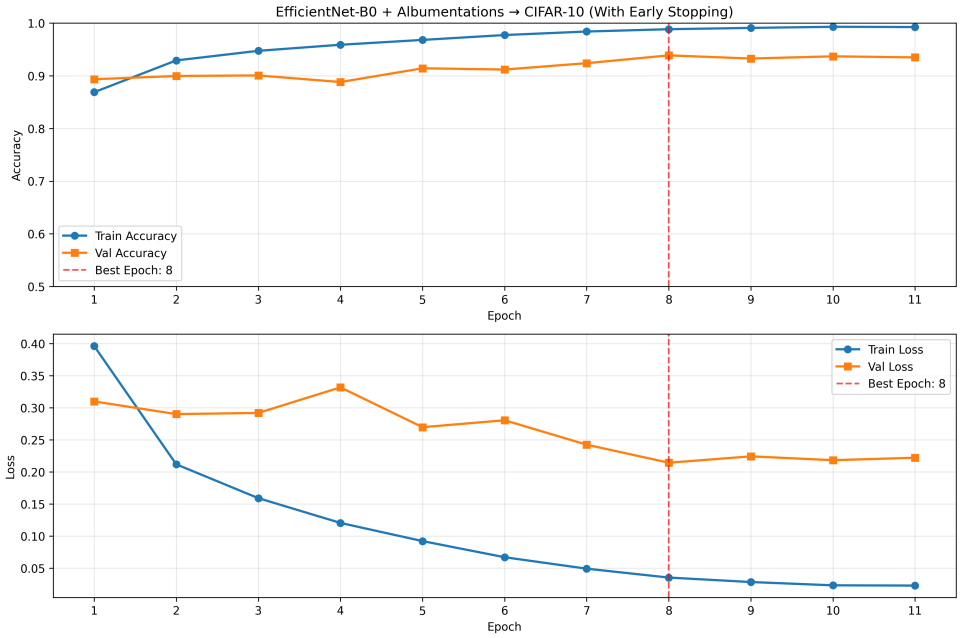

Рис. 3: График точности дообученной EfficientNet-B0

6. Latency

Была измерена пропускная задержка при обработке отдельно взятого изображения (1000 прогонов). Результат представлен на рис. 3. После начального прогрева задержка колеблется в районе 8.79 для B3 и в районе 5.54 для B0.

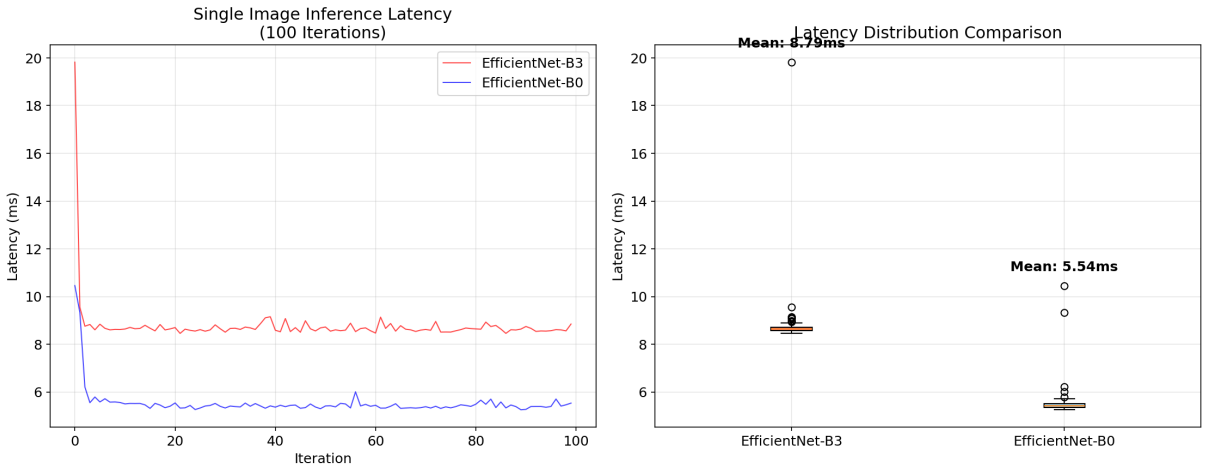

Рис. 4: График задержки на 1000 прогонах изображения

7. TensorRT

Попробовали использовать TensorRT в качестве бекенда. Непонятно зачем мы занялись такой глупостью на AMD. Результат ожидаемый.

```
======================================================================
🎯 FINAL THROUGHPUT COMPARISON
======================================================================
Backend              Throughput (img/s) Relative Speed
-------------------- ------------------ ---------------
PyTorch              507.01             1.00x
TensorRT             134.71             0.27x

📊 Performance Summary:
   • Speedup: 0.27x
   • Absolute gain: -372.30 img/s
   • Efficiency improvement: -73.4%
   ❗ PyTorch performs better

💾 Results saved to: throughput_results.txt

==================================================
🔧 SYSTEM INFORMATION
==================================================
PyTorch version: 2.4.1+rocm6.1
ONNX Runtime version: 1.23.2
CUDA available: True
GPU: AMD Radeon RX 6600
TensorRT Available: True
ONNX Runtime Provider Used: TensorRT
```

Рис. 5: Выхлоп TensorRT vs. PyTorch

8. Seaborn-barplot

Сравнили скорость инференса PyTorch, ONNX и ONNX с квантизацией int8. Результат представлен на следующем изображении.

9. INT8-точность

Сравнили точность float32 и int8. Результат плачевный: int8 пытается везде увидеть автомобиль. Точность соответствующая.

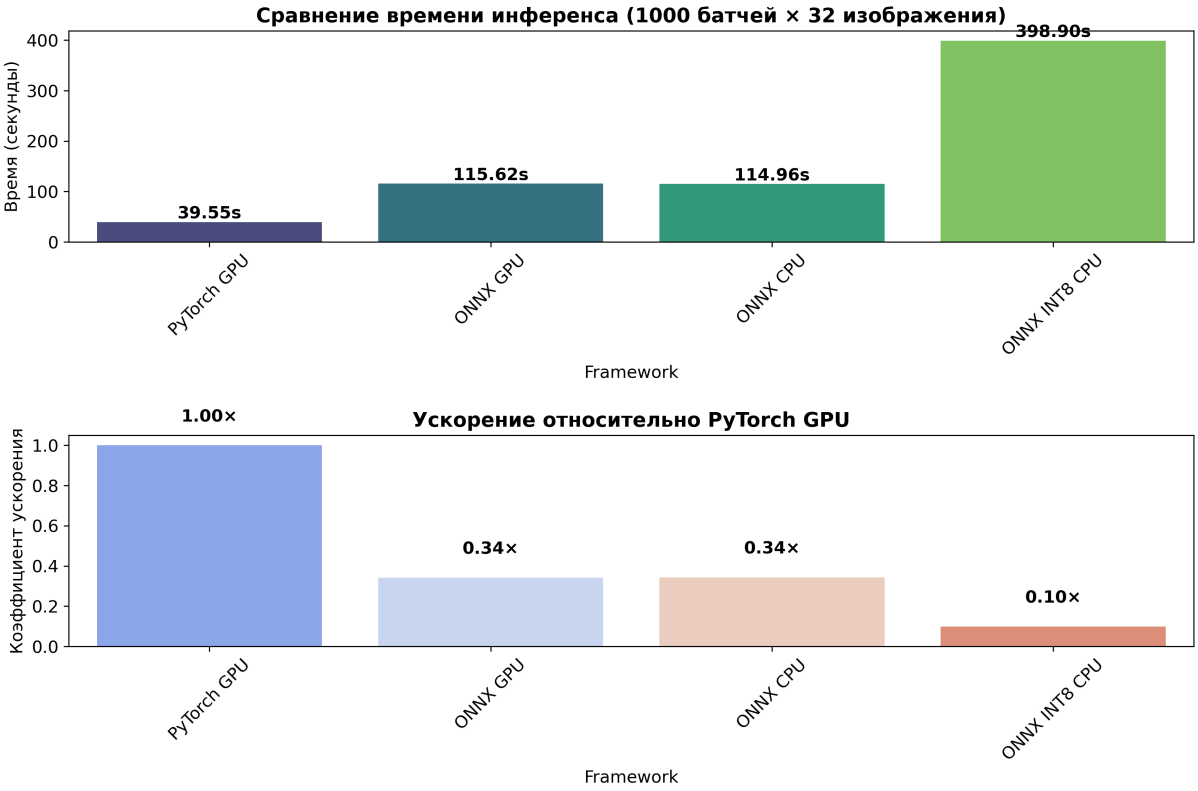

Рис. 6: Графики PyTorch vs. ONNX vs. ONNX INT8

Рис. 7: Сравнение квантов