Семинар 3. Оптимизация моделей и конвертация в ONNX

Ниже приведены **пять практических заданий**, направленных на закрепление ключевых инженерных приёмов, обсуждённых в лекции. Рекомендуется выполнять их последовательно — от простых до интеграционных, фиксируя результаты каждого шага и делая краткие наблюдения о компромиссе между скоростью и точностью.

Задание 1. TorchScript: фиксация вычислительного графа

Цель: понять разницу между динамическим и статическим графом в PyTorch и научиться получать воспроизводимую модель.

1. Создайте небольшую нейросеть (например, простую CNN для MNIST или CIFAR10, либо полносвязную сеть).
2. Выполните два варианта преобразования модели:
   - через `torch.jit.trace(model, example_input)`
   - через `torch.jit.script(model)`
3. Сохраните обе версии (`model_traced.pt`, `model_scripted.pt`).
4. Проверьте эквивалентность результатов на нескольких входах, вычислив норму разности: $$ \Delta = \frac{|f_{\text{orig}}(x) - f_{\text{jit}}(x)|*2}{|f*{\text{orig}}(x)|_2}. $$
5. Зафиксируйте наблюдения: как отличается поведение модели при условных ветвлениях?

Задание 2. Оптимизация вычислений: смешанная точность и квантизация

Цель: исследовать влияние формата чисел и методов оптимизации на скорость и точность инференса.

1. Возьмите модель из Задания 1.
2. Используя `torch.autocast`, выполните инференс в FP16 и сравните время выполнения с FP32.
3. Примените динамическое квантование:

```
from torch.quantization import quantize_dynamic
model_int8 = quantize_dynamic(model, {torch.nn.Linear}, dtype=torch.qint8)
```

4. Измерьте время инференса и разницу в предсказаниях между FP32 и INT8 версиями.
5. Сравните размер файлов (`.pt`) и сделайте вывод, какова цена ускорения в процентах точности.

Задание 3. Экспорт в ONNX и проверка эквивалентности

Цель: научиться экспортировать оптимизированную PyTorch-модель в ONNX и убедиться в корректности конверсии.

1. Экспортируйте одну из версий модели (FP32 или INT8) в формат ONNX:

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

2. Загрузите её с помощью `onnxruntime.InferenceSession` и выполните инференс.
3. Сравните результаты с PyTorch, вычислив среднюю абсолютную разницу: $$ E = \frac{1}{N}\sum_i |y_i - \hat{y}_i|. $$
4. Проверьте, что расхождение не превышает 10^{-3}.
5. Добавьте отчёт о размере `.onnx` файла и версии opset.

Задание 4. Оптимизация и профилирование ONNX-модели

Цель: освоить методы оптимизации графа и оценить их влияние на производительность.

1. Установите `onnxruntime` и `onnxoptimizer`.
2. Примените оптимизацию графа:

```
from onnx import optimizer
passes = ["eliminate_deadend", "fuse_add_bias_into_conv",
"fuse_bn_into_conv"]
model_optimized = optimizer.optimize(onnx.load("model.onnx"), passes)
onnx.save(model_optimized, "model_opt.onnx")
```

3. Создайте `InferenceSession` для исходной и оптимизированной модели.
4. Измерьте **latency** и **throughput** по 100 прогонам (используйте `time.time()`).
5. Сравните производительность и сохраните результаты в таблицу:

Модель	Размер (MB)	Latency (ms)	Throughput (req/s)	Δ Accuracy
model.onnx	...	...	...	...
model_opt.onnx	...	...	...	...

Задание 5. Подготовка артефакта для CI/CD

Цель: оформить модель как эксплуатационный артефакт, пригодный для интеграции в пайплайн.

1. Создайте папку `artifact_v1/`, поместив в неё:
   - `model_opt.onnx`
   - файл `metadata.yaml` с ключевыми параметрами:

```yaml
model_name: simple_cnn
version: 1.0
opset: 13
precision: int8
optimization: fuse_bn_into_conv
runtime: onnxruntime 1.18.0
latency_ms: <ваше значение>
accuracy_drop: <ваше значение>
```

2. Добавьте скрипт `validate.py`, который проверяет эквивалентность с эталонной моделью.
3. Зафиксируйте всё в Git (или DVC) с комментарием: *"Optimized ONNX artifact, verified equivalence ≤ 1e-3"*.
4. Подумайте, как этот артефакт можно использовать в автоматическом пайплайне CI/CD — какие проверки должны выполняться при каждом обновлении?
