Основы PyTorch для разработки и обучения моделей

Лекция 2

PyTorch как основа связки разработки и эксплуатации

История

2016 год — Facebook AI Research

Динамические графы vs статические TensorFlow

Преимущества

Условные конструкции и циклы

Простота отладки и интеграции

Современность

Стандарт де-факто

GPT-3, Hugging Face, Microsoft, Meta

Динамическая модель вычислений

1

Статические графы

Описание → компиляция → исполнение

2

Динамические графы

Граф строится в процессе выполнения

3

Гибкость

if, for, while в описании модели

Важно для эксплуатации: граф создается заново при каждом проходе — нужна стабильность для экспорта в ONNX/TorchScript

Тензоры: основа данных в PyTorch

Математическое определение тензора ранга d:

$$T \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$$

01

Многомерный массив

d=1 — вектор, d=2 — матрица

02

GPU поддержка

Хранение в памяти GPU для ускорения

03

Автодифференцирование

requires_grad=True для обучения

Создание и операции с тензорами

```
import torch

# Тензор из списка
tensor = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)

# Случайный тензор
random_tensor = torch.randn(3, 3)

# Арифметика
a = torch.tensor([1.0, 2.0, 3.0])
b = torch.tensor([4.0, 5.0, 6.0])
c = a + b # tensor([5., 7., 9.])
d = a * b # tensor([4., 10., 18.])

# Перемещение между устройствами
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
tensor = tensor.to(device)
```

Autograd: автоматическое дифференцирование

Система автоматического вычисления градиентов для обучения нейронных сетей

```
x = torch.tensor(2.0, requires_grad=True)
y = x**2 + 3*x + 1
y.backward()
print(x.grad)  # tensor(7.)
```

Функция:

$$y = x^2 + 3x + 1$$

Производная:

$$\frac{dy}{dx} = 2x + 3$$

При x=2 градиент равен 7

nn.Module: объектная структура моделей

```
import torch.nn as nn
import torch.nn.functional as F

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Инженерный контракт: nn.Module — ядро модели для сохранения и переноса между средами

Оптимизаторы и управление обучением

```
import torch.optim as optim

model = SimpleNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Управление весами

Обновление параметров на основе градиентов

Воспроизводимость

Фиксация параметров для промышленности

Данные как контракт между обучением и эксплуатацией

Формальное описание проблемы:

$$\mathcal{D}_{\text{train}} = \{(x_i, y_i)\}_{i=1}^{N} \sim P_{\text{train}}(X, Y)$$

$$\{x_t\}_{t \geq 1} \sim P_{\text{serve}}(X)$$

Контракт данных

Формально зафиксированные ожидания о структуре, типах, диапазонах

Инварианты

Проверка на каждом этапе пайплайна

Dataset и DataLoader: универсальный интерфейс

01

Map-style Dataset

__len__ и __getitem__ для локальных файлов

02

IterableDataset

Потоковая семантика для бесконечных данных

03

DataLoader

Батчирование, многопоточность, GPU-оптимизация

Пример TabularDataset с контрактами

```
from typing import Dict, Any, Tuple
import torch
from torch.utils.data import Dataset, DataLoader

class TabularDataset(Dataset):
 def __init__(self, rows: list[Dict[str, Any]],
 schema: Dict[str, str], target: str):
 self.rows = rows
 self.schema = schema # {"age":"float", "amount":"float"}
 self.target = target

 def __getitem__(self, idx: int) -> Tuple[torch.Tensor, torch.Tensor]:
 r = self.rows[idx]
 # Проверяем наличие всех полей
missing = [k for k in self.schema if k not in r]
if missing:
raise ValueError(f"missing features {missing}")

# Приводим типы строго по контракту
feats = []
for k, t in self.schema.items():
v = r[k]
if t == "float":
feats.append(float(v))
elif t == "int":
feats.append(int(v))

x = torch.tensor(feats, dtype=torch.float32)
y = torch.tensor(r[self.target], dtype=torch.long)
return x, y
```

Предобработка и инварианты

Train Pipeline

Стохастические аугментации

Документированные параметры

Serve Pipeline

Детерминированность инференса

Точно те же преобразования

```
class Normalize(nn.Module):
 def __init__(self, mean, std):
 super().__init__()
 m = torch.as_tensor(mean, dtype=torch.float32).view(1, -1, 1, 1)
 s = torch.as_tensor(std, dtype=torch.float32).view(1, -1, 1, 1)
 self.register_buffer("mean", m)
 self.register_buffer("std", s)

 def forward(self, x: torch.Tensor) -> torch.Tensor:
 return (x - self.mean) / self.std
```

Схемы и валидация данных

```
import pandera as pa
from pandera import Column, DataFrameSchema, Check

schema = DataFrameSchema({
    "age": Column(pa.Float, Check.in_range(0, 120), nullable=False),
    "amount": Column(pa.Float, Check.ge(0.0), nullable=False),
    "country": Column(pa.String, Check.isin(["NL","DE","RU"]), nullable=False),
    "label": Column(pa.Int, Check.isin([0,1]), nullable=False),
})

# Применяем схему ко всем наборам данных
df_train = schema.validate(df_train)
df_val = schema.validate(df_val)
df_test = schema.validate(df_test)
```

Схема переводит споры из устных в строгие правила

Воспроизводимость данных

1 **Сырой источник**

Хэш партиций, snapshot времени

2 **Код подготовки**

Версионирование скриптов фичей

3 **Чистый датасет**

Артефакт с хэшем

4 **Препроцессинг**

Сохранение параметров нормализаций

Типы дрейфа данных

Covariate Shift

Изменилось $P(X)$, но $P(Y|X)$ прежнее

Реакция: аугментации, стабилизация препроцессинга

Label Shift

Изменилось $P(Y)$, но $P(X|Y)$ прежнее

Реакция: рекалибровка вероятностей

Concept Drift

Изменилось $P(Y|X)$: сама зависимость стала другой

Реакция: переобучение модели

Измерение дрейфа: PSI и KL-дивергенция

Population Stability Index (PSI):

$$\mathrm{PSI} = \sum (p_i - q_i) \ln \frac{p_i}{q_i}$$

KL-дивергенция:

$$D_{KL}(P|Q) = \sum p_i \log \frac{p_i}{q_i}$$

```
def psi(reference: np.ndarray, current: np.ndarray, bins: int = 10, eps: float = 1e-6) -> float:
 q = np.linspace(0, 100, bins+1)
 r_bins = np.percentile(reference, q)
 r_hist, _ = np.histogram(reference, bins=r_bins)
 c_hist, _ = np.histogram(current, bins=r_bins)
 p = (r_hist / max(1, r_hist.sum())) + eps
 q = (c_hist / max(1, c_hist.sum())) + eps
 return float(np.sum((p - q) * np.log(p / q)))
```

Производственные нюансы форматов

Computer Vision

Один декодер, нормализация к sRGB, обработка EXIF

Аудио

Фиксация частоты дискретизации, окна STFT

Текст

Нормализация Unicode (NFC), политика пробелов

Табличные

Колоночные форматы (Parquet), явные типы

Безопасность и производительность данных

Безопасность

PII обезличивание

Защита от отравления данных

Контроль источников

Производительность

pin_memory=True

non_blocking=True

Континуальная память

```
def seed_worker(_):
 seed = torch.initial_seed() % 2**32
 random.seed(seed); np.random.seed(seed)

g = torch.Generator().manual_seed(1337)
loader = DataLoader(dataset, batch_size=256, num_workers=8,
 pin_memory=True, worker_init_fn=seed_worker, generator=g)
```

От проектирования модели к промышленному артефакту

1

Исследование

Описание архитектуры в классе

2

Разработка

Версионирование и сериализация

3

Эксплуатация

Тестирование, оптимизация, мониторинг

Модель в промышленности — это артефакт экосистемы, а не просто код

Архитектура: от учебного к управляемому

Учебный пример

```
class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(10, 1)

    def forward(self, x):
        return torch.sigmoid(self.linear(x))
```

Промышленный подход

```
class RobustNet(nn.Module):
 def __init__(self, in_features, out_features):
 super().__init__()
 self.linear = nn.Linear(in_features, out_features)

 def forward(self, x):
 if x.dim() != 2 or x.size(1) != self.linear.in_features:
 raise ValueError(f"Unexpected input shape: {x.shape}")
 out = self.linear(x)
 return torch.sigmoid(out)
```

Функции потерь: от формул к бизнес-метрикам

Математическая формулировка оптимизации:

$$\hat{\theta} = \arg\min_{\theta} \mathbb{E}_{(x,y)\sim P_{\text{train}}}[\mathcal{L}(f_\theta(x), y)]$$

Стандартные функции

MSE для регрессии

Кросс-энтропия для классификации

Бизнес-приоритеты

Взвешенные функции потерь

Учет стоимости ошибок

```
# Взвешенная кросс-энтропия для несбалансированных классов
loss_fn = nn.CrossEntropyLoss(weight=torch.tensor([1.0, 5.0]))
```

Оптимизация и воспроизводимость

Формула обновления параметров:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(f_\theta(x), y)$$

```
import torch, random, numpy as np

# Фиксация всех генераторов для воспроизводимости
torch.manual_seed(42)
random.seed(42)
np.random.seed(42)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

Без фиксации seed повторное обучение может дать другую модель

Тренировочный цикл: от минимализма к управляемости

Учебный цикл

```
for xb, yb in loader:
    optimizer.zero_grad()
    preds = model(xb)
    loss = loss_fn(preds, yb)
    loss.backward()
    optimizer.step()
```

Промышленный цикл

```
import mlflow

with mlflow.start_run():
 for epoch in range(epochs):
 for xb, yb in loader:
 optimizer.zero_grad()
 preds = model(xb)
 loss = loss_fn(preds, yb)
 loss.backward()
 optimizer.step()

 mlflow.log_metric("loss", loss.item(), step=epoch)
 mlflow.pytorch.log_model(model, "model")
```

Валидация и защита от переобучения

Математическое определение переобучения:

$$\mathbb{E}_{(x,y)\sim P_{\text{train}}}[\ell(f_\theta(x), y)] \ll \mathbb{E}_{(x,y)\sim P_{\text{test}}}[\ell(f_\theta(x), y)]$$

```
best_val = float("inf")
patience, trigger = 3, 0

for epoch in range(epochs):
 train(...)
 val_loss = validate(...)

 if val_loss < best_val:
 best_val = val_loss
 trigger = 0
 else:
 trigger += 1
 if trigger >= patience:
 print("Early stopping")
 break
```

Метрики качества: от Accuracy к бизнес-значимости

Антифрод

Критичен Recall — пропущенный мошенник дорого стоит

Рекомендации

NDCG, MRR — важна релевантность первых N результатов

Медицина

Баланс Precision и Recall — ложные срабатывания опасны

Главный принцип: метрика — мост между моделью и бизнесом

Интеграция в MLOps

01

Гиперпараметры

Фиксация в трекере экспериментов

02

Артефакты

Сохранение и версионирование

03

Воспроизводимость

Фиксация seed и окружения

04

Оценка

Бизнес-метрики и мониторинг

Зачем нужна отладка в промышленной эксплуатации

Исследования

Печать форм тензоров

Визуализация примеров

Промышленность

Систематическая практика

Инфраструктура мониторинга

Автоматические оповещения

Отладка превращается из случайного процесса в воспроизводимую процедуру

Визуализация и проверка тензоров

```
x = torch.randn(32, 3, 224, 224)  # батч картинок
print(x.shape)  # ожидаем [batch, channels, height, width]
```

Исследования

Простая проверка размерностей

Эксплуатация

Встроенные проверки в forward

Осмысленные исключения

Предотвращение тихих сбоев

Профилирование производительности

```
import torch.profiler as profiler

model = nn.Sequential(
    nn.Conv2d(3, 16, 3),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(16*222*222, 10)
)

x = torch.randn(8, 3, 224, 224)

with profiler.profile(record_shapes=True) as prof:
    with profiler.record_function("model_inference"):
        model(x)

print(prof.key_averages().table(sort_by="cpu_time_total"))
```

Помогает локализовать узкие места в препроцессинге и передаче данных

Регистрация метрик и логирование

```
from prometheus_client import Counter, Histogram

requests_total = Counter("inference_requests_total", "Total inference requests")
latency = Histogram("inference_latency_seconds", "Latency of inference")

def predict(x):
    requests_total.inc()
    with latency.time():
        return model(x)
```

1

Сбор метрик

Латентность, ошибки, распределения

2

Дашборды

Prometheus + Grafana

3

Наблюдаемость

Превращение "черного ящика" в управляемую систему

Обнаружение дрейфа в эксплуатации

```
def monitor_drift(ref, cur, bins=10):
    # ref и cur — numpy-массивы значений признака
    hist_ref, bins = np.histogram(ref, bins=bins)
    hist_cur, _ = np.histogram(cur, bins=bins)
    p = hist_ref / np.sum(hist_ref)
    q = hist_cur / np.sum(hist_cur)
    psi = np.sum((p - q) * np.log((p + 1e-6)/(q + 1e-6)))
    return psi
```

Регулярные проверки каждые 10 минут → триггер для алертов → инженерная реакция

Отладка через explainability

Важность признаков

Анализ влияния входных переменных

Grad-CAM

Подсветка областей изображения

Объяснения

Критично в финансах и медицине

Методы интерпретации помогают убедиться, что модель "смотрит" на правильные области

Отладка как непрерывный процесс

Проверка входов

Согласованность данных

Профилирование

Устранение узких мест

Мониторинг

Логирование метрик

Интерпретация

Объяснение поведения

Дрейф данных

Предупреждения инженеров

Отладка — главный механизм надежности ML-систем в эксплуатации