Шпаргалка по PyTorch и TensorFlow / Torch

Cheatsheet (XeLaTeX)

Краткий справочник по основным операциям

April 2, 2025

Contents

Зачем нужны DL Фреймворки?

PyTorch и TensorFlow — это мощные инструменты, которые сильно упрощают создание и обучение глубоких нейронных сетей (Deep Learning, DL). Они берут на себя сложную математику и оптимизацию, позволяя тебе сосредоточиться на архитектуре модели и данных.

- **Аналогия:** Представь, что строишь сложный механизм. Вместо того чтобы вытачивать каждую шестеренку вручную, ты используешь готовые стандартные блоки (слои, функции активации) и инструменты (оптимизаторы, расчет градиентов), которые предоставляют фреймворки.
- **Цель этой шпаргалки:** Понять самые базовые "строительные блоки" и процесс "сборки" (обучения) в PyTorch и TensorFlow. Не бойся синтаксиса, концепции очень похожи!

1 Тензор: Основа Всего

Что такое Тензор?

Тензор (Tensor) — это фундаментальная структура данных в DL фреймворках. По сути, это многомерный массив чисел.

- 0-мерный тензор: скаляр (просто число).
- 1-мерный тензор: вектор (массив).
- 2-мерный тензор: матрица (таблица).
- 3-мерный тензор: куб чисел (например, цветное изображение RGB).
- и так далее...

Аналогия: Думай о тензоре как о супер-продвинутом NumPy массиве, который умеет работать на GPU и "помнит" вычисления для градиентов. В тензорах хранятся входные данные, веса модели, выходы слоев и т.д. *Микро-уточнение для TensorFlow:* В TF основной неизменяемый тип — `tf.Tensor` (результат операций, создается через `tf.constant`, `tf.zeros` и т.д.). Для изменяемых параметров модели, которые нужно отслеживать для градиентов, используется `tf.Variable`.

Создание Тензоров

```python
# === PyTorch ===
import torch

# Создание из списка Python
t_list_pt = torch.tensor([[1., 2.], [3., 4.]])

# Создание тензора из нулей
t_zeros_pt = torch.zeros((2, 3)) # Форма (2 строки, 3 столбца)

# Создание тензора из случайных чисел
t_rand_pt = torch.rand((2, 2))

# === TensorFlow ===
import tensorflow as tf

# Создание из списка Python (создает tf.Tensor)
t_list_tf = tf.constant([[1., 2.], [3., 4.]])

# Создание тензора из нулей (создает tf.Tensor)
t_zeros_tf = tf.zeros((2, 3)) # Форма (2 строки, 3 столбца)

# Создание тензора из случайных чисел (создает tf.Tensor)
t_rand_tf = tf.random.uniform((2, 2))
# tf.Variable обычно создается отдельно для весов модели
```

2 Автодифференцирование: Как Сеть Учится

Зачем это нужно?

Обучение нейросети — это подбор таких весов, чтобы ошибка (loss) была минимальной. Для этого используется **градиентный спуск**. Нам нужно знать, как небольшое изменение каждого веса влияет на итоговую ошибку. Это и есть **градиент**. Вычислять его вручную для сложных сетей нереально. **Автоматическое дифференцирование (Autograd)** — это механизм, который автоматически вычисляет градиенты функции потерь по всем параметрам модели. **Аналогия:** Представь "волшебного бухгалтера", который следит за каждой математической операцией в сети. Когда ты получаешь итоговую ошибку, ты можешь спросить у него: "Эй, как изменение вот этого конкретного веса в самом начале повлияло на эту ошибку?". И он мгновенно даст тебе ответ (градиент).

Autograd/GradientTape на практике

```python
# === PyTorch (Autograd) ===
import torch

# Тензор, для которого нужно считать градиенты
x_pt = torch.tensor(2.0, requires_grad=True)
w_pt = torch.tensor(3.0, requires_grad=True)
b_pt = torch.tensor(1.0, requires_grad=True)

# Операции
y_pt = w_pt * x_pt + b_pt # y = 3 * 2 + 1 = 7

# Вычисляем градиенты d(y)/d(w), d(y)/d(x), d(y)/d(b)
y_pt.backward()

# Градиенты сохраняются в атрибуте .grad
print(f"PyTorch dy/dw: {w_pt.grad}") # Ожидаем x = 2
print(f"PyTorch dy/dx: {x_pt.grad}") # Ожидаем w = 3
print(f"PyTorch dy/db: {b_pt.grad}") # Ожидаем 1

# === TensorFlow (GradientTape) ===
import tensorflow as tf

# Используем tf.Variable, чтобы TF отслеживал их
x_tf = tf.Variable(2.0)
w_tf = tf.Variable(3.0)
b_tf = tf.Variable(1.0)

# Операции должны быть внутри контекста GradientTape
with tf.GradientTape() as tape:
    y_tf = w_tf * x_tf + b_tf # y = 3 * 2 + 1 = 7

# Вычисляем градиенты
# tape.gradient(целевая_переменная, список_переменных_по_которым_считаем)
gradients = tape.gradient(y_tf, {'w': w_tf, 'x': x_tf, 'b': b_tf})

print(f"TensorFlow dy/dw: {gradients['w']}") # Ожидаем x = 2
print(f"TensorFlow dy/dx: {gradients['x']}") # Ожидаем w = 3
print(f"TensorFlow dy/db: {gradients['b']}") # Ожидаем 1
```

Важно!

В PyTorch нужно явно указывать `requires_grad=True` для тензоров, по которым будут считаться градиенты (обычно это параметры модели). В TensorFlow для `tf.Variable` отслеживание включено по умолчанию, а операции нужно помещать внутрь `tf.GradientTape()`.

3 Модель: Рецепт Преобразования Данных

Что такое модель?

Модель (Model) — это, по сути, функция (часто очень сложная), которая преобразует входные данные в выходные (предсказание). В DL она обычно состоит из последовательности **слоев (Layers)**. Каждый слой выполняет определенное преобразование над данными, используя свои **параметры (веса)**. **Аналогия:** Модель — это как кулинарный рецепт. Входные данные — ингредиенты. Слои — шаги рецепта (смешать, запечь, нарезать). Веса слоя — это параметры шага (сколько муки, температура духовки). Выход модели — готовое блюдо (предсказание).

Пример простой линейной модели

```python
# === PyTorch (nn.Module) ===
import torch
import torch.nn as nn

class SimpleLinearPT(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SimpleLinearPT, self).__init__()
        # Определяем слои в конструкторе
        self.linear = nn.Linear(input_dim, output_dim)

    # Метод forward определяет прямой проход
    def forward(self, x):
        # Определяем порядок вычислений (прямой проход)
        out = self.linear(x)
```

```
        return out

# Создаем модель: входная размерность 5, выходная 1
model_pt = SimpleLinearPT(input_dim=5, output_dim=1)
print("PyTorch Model:", model_pt)

# === TensorFlow (tf.keras.Model / tf.keras.Sequential) ===
import tensorflow as tf
from tensorflow.keras.layers import Dense
from tensorflow.keras import Model, Sequential

# Простой способ через Sequential (для линейной последовательности слоев)
model_tf_seq = Sequential([
    Dense(units=1, input_shape=(5,)) # units - выходная размерность
])
print("\nTensorFlow Sequential Model:")
model_tf_seq.build(input_shape=(None, 5)) # Явно строим модель (None - размер батча)
model_tf_seq.summary()

# Более гибкий способ через наследование Model
class SimpleLinearTF(Model):
    def __init__(self, output_dim):
        super(SimpleLinearTF, self).__init__()
        # Определяем слои
        self.dense = Dense(units=output_dim)

    # Метод call определяет прямой проход (аналог forward в PyTorch)
    def call(self, x):
        # Определяем прямой проход
        return self.dense(x)

# Создаем модель: выходная размерность 1
model_tf_custom = SimpleLinearTF(output_dim=1)
# Чтобы увидеть summary, нужно "вызвать" модель на данных нужной формы
_ = model_tf_custom(tf.zeros((1, 5))) # Прогоняем "пустой" тензор для построения
print("\nTensorFlow Custom Model:")
model_tf_custom.summary()
```

4 Цикл Обучения: Шаг за Шагом к Результату

Как происходит обучение?

Обучение — это итеративный процесс, где модель настраивает свои веса, чтобы минимизировать ошибку на обучающих данных. Каждый проход по данным называется **эпохой**. Внутри эпохи данные обычно делятся на **батчи (batches)**. **Аналогия:** Студент готовится к экзамену (обучение). У него есть учебник (данные). Он читает главу за главой (эпохи). Каждую главу он прорабатывает по частям (батчи). Для каждой части: 1. Пытается ответить на вопросы (делает предсказание - **forward pass**). 2. Сравнивает свои ответы с правильными (считает ошибку - **loss**). 3. Анализирует, где ошибся и почему (считает градиенты - **backward pass**). 4. Корректирует свое понимание материала (обновляет веса - **optimizer step**). 5. Перед следующей частью "очищает" голову от предыдущих размышлений (обнуляет градиенты - **zero grad**).

Концептуальный цикл обучения (Псевдокод)

```
# --- Общие компоненты ---
# model = ... (ваша модель PyTorch или TensorFlow)
# criterion = ... (функция потерь, напр., nn.MSELoss() или tf.keras.losses.MeanSquaredError())
# optimizer = ... (оптимизатор, напр., torch.optim.SGD() или tf.keras.optimizers.SGD())
# train_loader = ... (загрузчик данных, который выдает батчи X, y)
# num_epochs = ... (количество эпох)

# === PyTorch ===
# for epoch in range(num_epochs):
#     for i, (inputs, labels) in enumerate(train_loader):
#         # 1. Forward pass: получить предсказания
#         outputs = model(inputs)
#
#         # 2. Calculate loss: сравнить с реальными метками
#         loss = criterion(outputs, labels)
#
#         # 3. Backward pass: вычислить градиенты
#         optimizer.zero_grad() # !!! ОБНУЛИТЬ градиенты перед backward() !!!
#         loss.backward()
#
#         # 4. Optimizer step: обновить веса модели
#         optimizer.step()
#
#     # (Опционально) Логирование потерь за эпоху...

# === TensorFlow (с использованием model.fit - проще для старта) ===
# model.compile(optimizer=optimizer, loss=criterion, metrics=['accuracy']) # Опционально метрики
#
# history = model.fit(train_loader, # Или просто X_train, y_train
#                     epochs=num_epochs,
#                     #
validation_data=validation_loader # Опционально
#                    )
# print("Обучение завершено!")

# === TensorFlow (ручной цикл - для понимания) ===
# for epoch in range(num_epochs):
#     for step, (x_batch_train, y_batch_train) in enumerate(train_loader):
#         with tf.GradientTape() as tape:
#             # 1. Forward pass
#             logits = model(x_batch_train, training=True) # Указать training=True для некоторых слоев
#             # 2. Calculate loss
#             loss_value = criterion(y_batch_train, logits)
#
#         # 3. Backward pass (вычисление градиентов)
#         grads = tape.gradient(loss_value, model.trainable_weights)
#
#         # 4. Optimizer step (применение градиентов)
#         optimizer.apply_gradients(zip(grads, model.trainable_weights))
#
#     # (Опционально) Логирование потерь за эпоху...
```

Ключевые моменты цикла:

* **Forward Pass:** Прогон данных через модель для получения предсказания. * **Loss Calculation:** Вычисление функции потерь (насколько предсказание отличается от истины). * **Backward Pass:** Вычисление градиентов функции потерь по параметрам модели (используя `autograd` / `GradientTape`). * **Optimizer Step:** Обновление параметров модели с использованием вычисленных градиентов и выбранного алгоритма оптимизации (SGD, Adam и т.д.). * **Zero Gradients (PyTorch):** **Критически важно** обнулять градиенты перед каждым `backward()` вызовом в PyTorch (`optimizer.zero_grad()`), иначе градиенты будут накапливаться от предыдущих батчей. В TensorFlow при использовании `GradientTape` это происходит автоматически для каждого вызова `tape.gradient`.