6. Как работает обучение: построчный разбор с пояснением всех терминов

```
optimizer.zero_grad()        # Обнуляем градиенты, чтобы не накапливались
```

👉 **Градиенты** — это производные функции ошибки по весам модели. Они указывают, в каком направлении нужно изменить веса, чтобы уменьшить ошибку.

Каждый шаг обучения должен начинаться с **обнуления градиентов**, иначе они будут накапливаться от прошлых итераций и мешать корректному обучению.

```
out = model(X)               # Прямой проход — получаем предсказания модели
(логиты)
```

👉**Прямой проход** (forward pass) — это процесс, в котором входные данные проходят через слои модели, и на выходе мы получаем **логиты**.

😥 **Что такое логиты?** Это значения, которые ещё не прошли через softmax. Они не являются вероятностями, но могут быть преобразованы в вероятности. CrossEntropyLoss применяет softmax автоматически.

```
loss = criterion(out, y)     # Считаем ошибку модели (функция потерь)
```

👉 **Функция потерь** (loss function) измеряет, насколько предсказание модели отклоняется от правильного ответа.

Мы используем:

- `nn.CrossEntropyLoss()` — это совмещённая функция, которая применяет softmax к логитам и затем вычисляет логарифмическую ошибку. Это стандарт для задач классификации.

```
loss.backward()              # Обратное распространение ошибки (backpropagation)
```

👉 **Обратное распространение ошибки** — это алгоритм, который автоматически вычисляет градиенты всех параметров (весов) модели на основе текущей ошибки `loss`.

```
optimizer.step()             # Обновляем веса модели с помощью градиентов
```

👉Это ключевой момент: **веса модели корректируются** на небольшую величину (шаг градиента), чтобы улучшить точность предсказаний в будущем.

👈🏽 Используется оптимизатор `Adam` — адаптивный алгоритм, который работает лучше SGD в большинстве задач.

```
predicted = torch.argmax(out, dim=1)  # Выбор наиболее вероятного класса
correct += (predicted == y).sum().item()  # Подсчёт правильных предсказаний
```

👉Здесь происходит **подсчёт точности** модели на текущей эпохе:

- `torch.argmax` выбирает номер класса с наибольшим логитом
- Сравниваем с правильной меткой `y` и считаем точные попадания

```
loss_history.append(avg_loss)
accuracy_history.append(accuracy)
```

👉Эти строки сохраняют значение ошибки и точности за каждую эпоху, чтобы потом можно было построить графики.

```
print(full_log)
with open("train_log.txt", "w", encoding="utf-8") as f:
    f.write(full_log)
```

👉Ведётся лог всех эпох: это важно для отладки и анализа — можно посмотреть, как изменилась ошибка и точность со временем.

---

7. Расшифровка цепочек типа `self._apply_activation(self.bn2(self.fc2(x)))`

Разберём эту строку пошагово:

```
x = self._apply_activation(self.bn2(self.fc2(x)))
```

1. `self.fc2(x)` — **линейный слой (Linear)**:
2. Производится **матричное умножение** входного вектора на матрицу весов слоя и добавляется смещение (bias):
    - `x @ W + b`
3. Это позволяет преобразовать вход в другое линейное пространство.
4. `self.bn2(...)` — **Batch Normalization**:

5. Нормализует выход по мини-пакету (батчу): вычитает среднее, делит на стандартное отклонение, а потом обучает масштаб и сдвиг.

6. Делает градиенты более стабильными, ускоряет сходимость.

7. `self._apply_activation(...)` — применяется **нелинейность (ReLU, Sigmoid, Tanh)**:

8. Это позволяет модели учить более сложные зависимости.

9. Без активаций вся модель была бы просто одним большим линейным преобразованием.

Таким образом, одна такая строка — это **полноценный слой нейросети**:

> Линейная трансформация → Нормализация → Нелинейность

---

8. Как работает `self`, `self.fcX`, `self.bnX`, `self.dropoutX`

- `self` — это ссылка на экземпляр класса. Она позволяет сохранять и использовать параметры внутри класса.
- `self.fc1 = nn.Linear(...)` — создаёт слой и сохраняет его как атрибут.
- Все поля, созданные через `self`, используются внутри метода `forward` и автоматически участвуют в обучении.
- Благодаря этому, когда мы вызываем `model.parameters()`, PyTorch знает, какие веса нужно обновлять.

Примеры:

```
self.fc1 = nn.Linear(784, 256)    # 784 входа, 256 выхода — первый слой
self.bn1 = nn.BatchNorm1d(256)    # Нормализация для 256 выходов
self.dropout1 = nn.Dropout(0.3)   # Убирает случайно 30% нейронов на обучении
```

Каждый компонент отвечает за свою функцию:

- `fc` — обучаем веса
- `bn` — стабилизируем обучение
- `dropout` — боремся с переобучением
