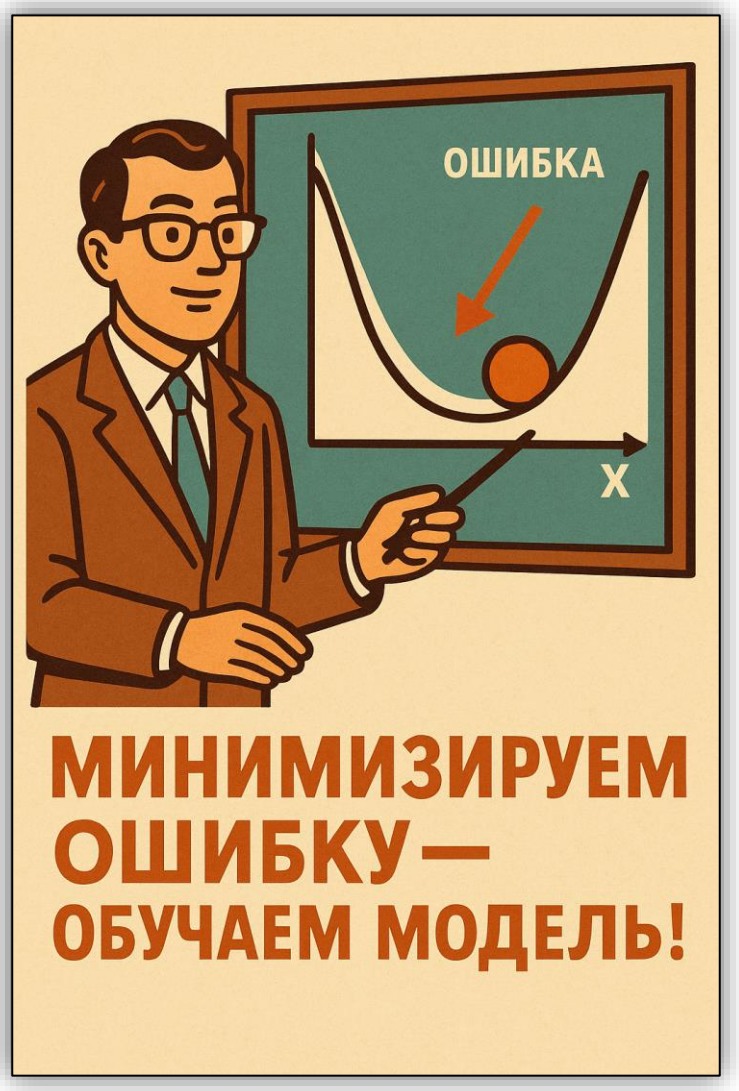
Методы оптимизации в машинном обучении

Лекция 4

Зачем нужна оптимизация

- Оптимизация - процесс поиска таких параметров, при которых ошибка модели минимальна.
- Идея: спускаемся по поверхности функции потерь к её минимуму.
- Пример: спуск шарика в долину - поиск оптимума.



Градиентный спуск

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

θ —параметры модели,
η — шаг обучения (learning rate),
$\nabla_\theta L(\theta_t)$— градиент функции потерь по параметрам.

- Поиск минимума функции потерь.
- Движение в сторону, противоположную градиенту.
- Шаг обучения определяет скорость.
- Каждое шаг — приближение к оптимальным параметрам.

Типы градиентного спуска

- Batch Gradient Descent — использует все данные, точный, но медленный
- Stochastic Gradient Descent (SGD) — один пример за шаг, быстрый, но шумный
- Mini-Batch Gradient Descent — небольшие батчи (32–256), компромисс между скоростью и стабильностью

В PyTorch:

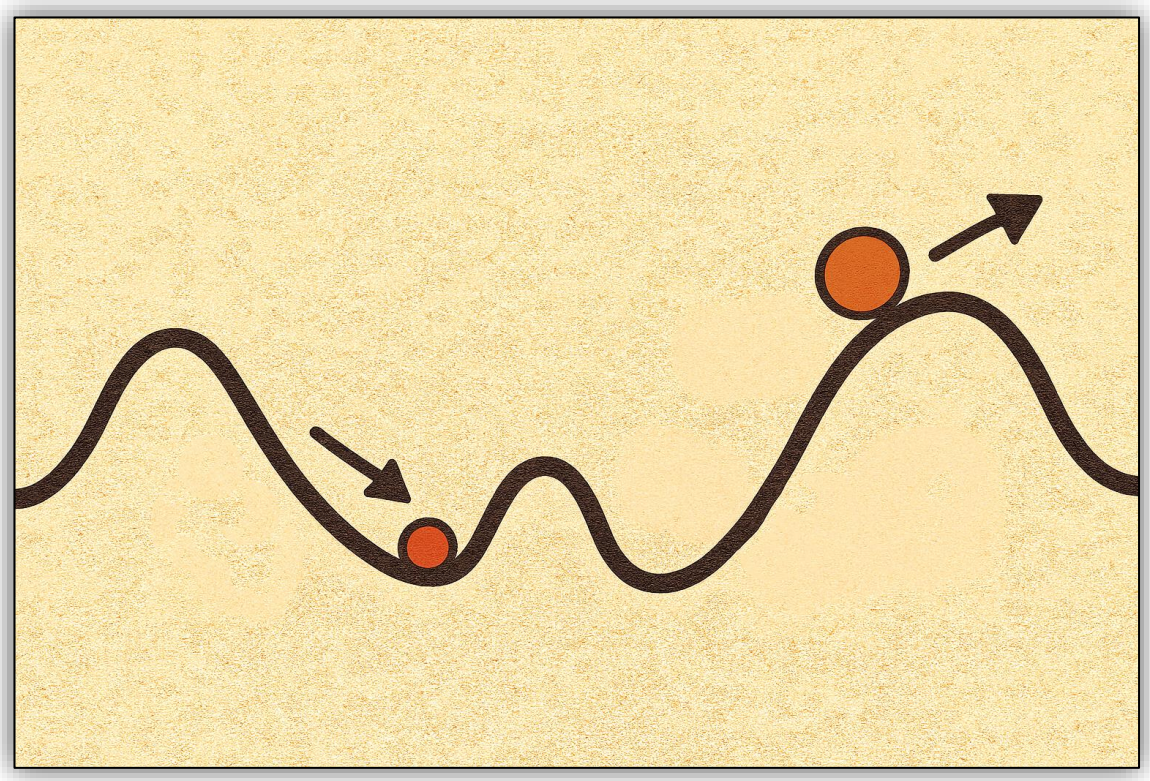
- `torch.optim.SGD` — универсальная реализация

Тип спуска задаётся размером batch_size:

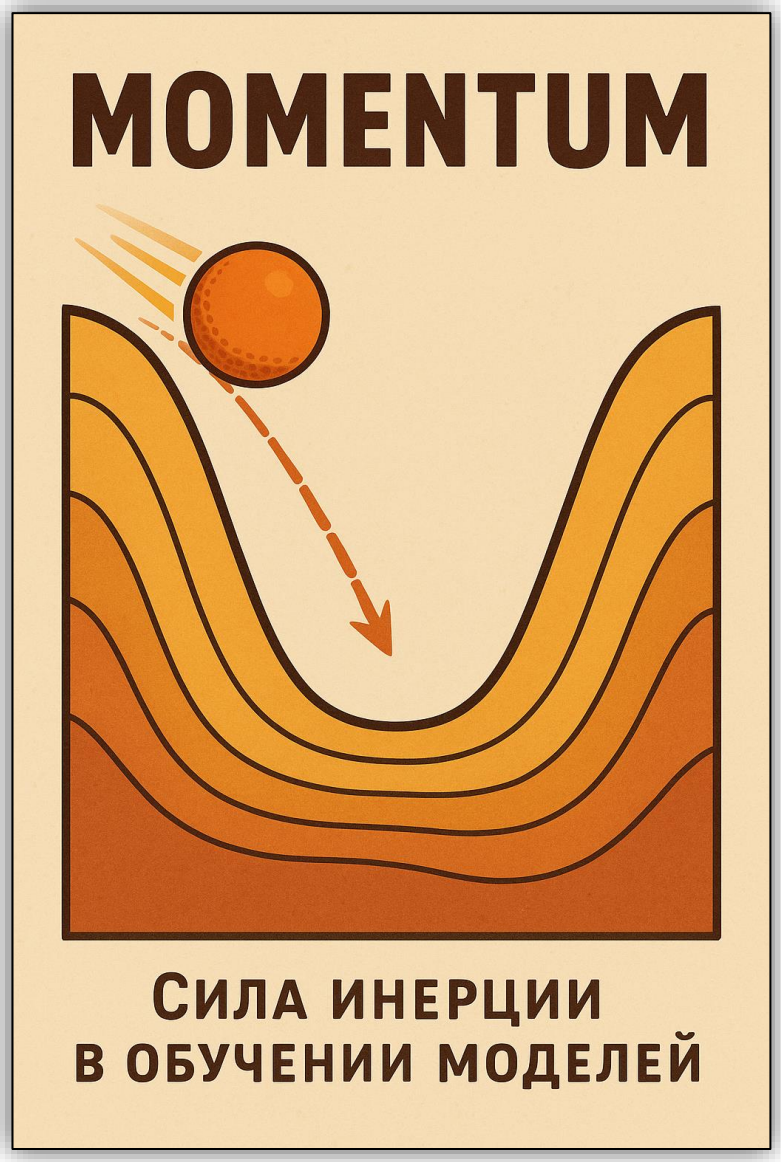
- batch_size = 1 → стохастический
- batch_size = len(dataset) → пакетный
- batch_size = 32–256 → мини-батчевый

Проблемы градиентного спуска

- Локальные минимумы — модель застревает не в лучшем решении
- Плато и седловые точки — градиент почти нулевой, обучение замирает
- Слишком большой шаг — “прыгаем” мимо минимума
- Слишком маленький шаг — обучение идёт слишком медленно
- Масштаб признаков влияет на направление спуска



Momentum (Импульс)



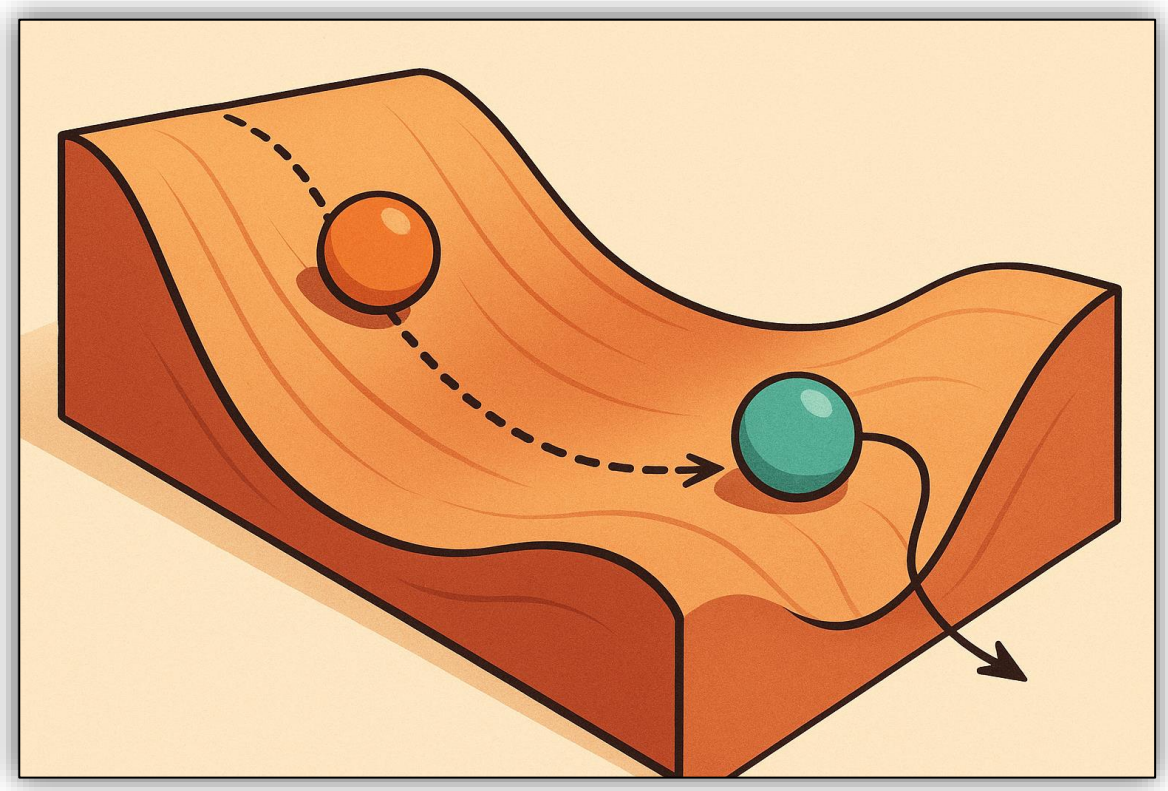
- Ускоряет обучение, сглаживая колебания
- Сохраняет часть предыдущего направления движения
- Помогает пройти через “ямы” и “плато” функции потерь
- Особенно эффективен при искривлённой поверхности

PyTorch:

```
torch.optim.SGD(..., momentum=0.9)
```

Nesterov Accelerated Gradient (NAG)

- Улучшает Momentum за счёт “заглядывания вперёд”
- Сначала делаем прогноз шага, потом считаем градиент
- Позволяет скорректировать движение до перескока минимума
- Делает обучение более стабильным и точным



PyTorch :

```
torch.optim.SGD(..., momentum=0.9,
nesterov=True)
```

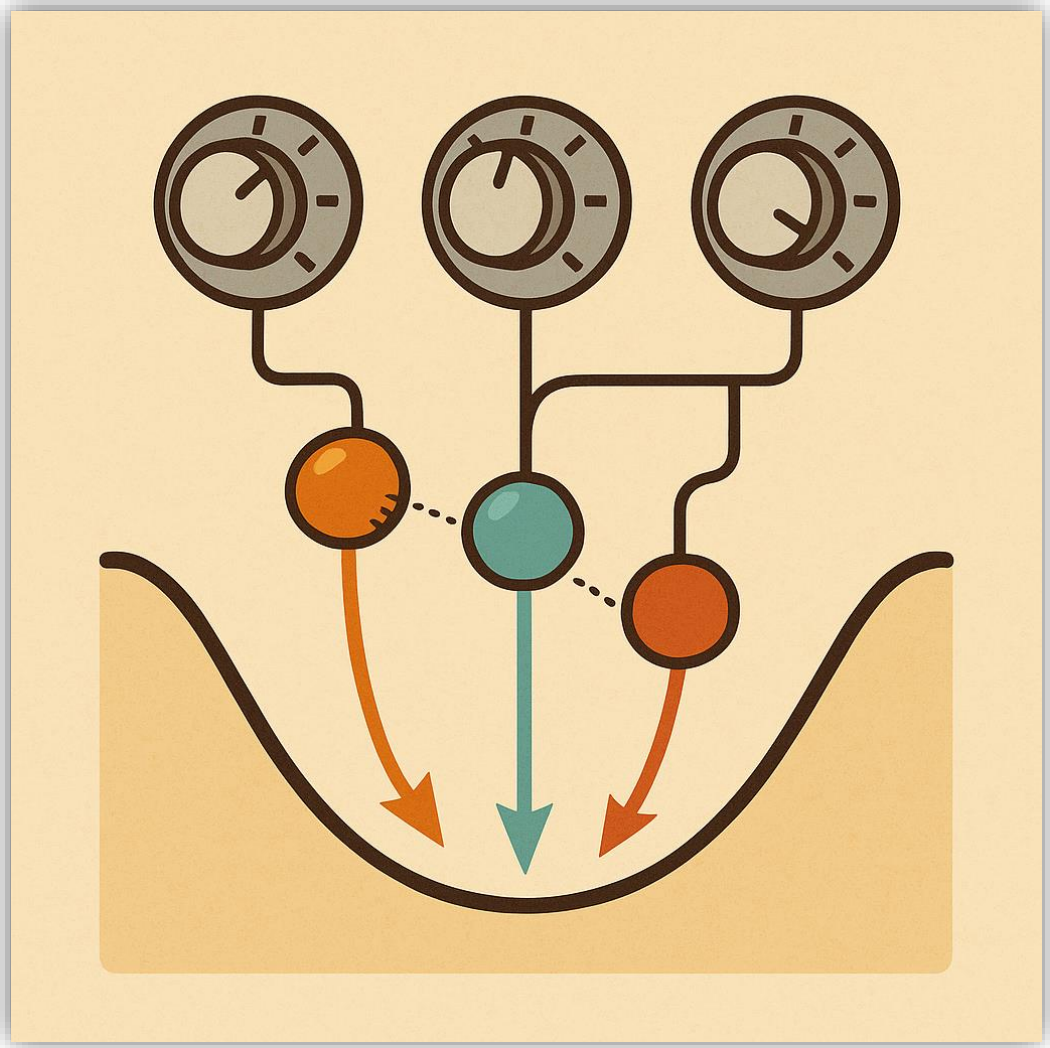
Адаптивные методы

- Идея: каждому параметру — свой шаг обучения
- Учитывают историю градиентов и их квадраты
- Позволяют автоматически подстраивать скорость обучения
- Меньше ручной настройки, быстрее сходимость
- Основные методы: AdaGrad, RMSProp, Adam

PyTorch :

`torch.optim.Adagrad` `torch.optim.RMSprop`

`torch.optim.Adam`.



AdaGrad (Adaptive Gradient)

- Первый адаптивный метод оптимизации
- Уменьшает шаг обучения по мере накопления градиентов
- Часто обновляемые параметры получают меньшие шаги
- Реже обновляемые — большие шаги
- Хорошо работает при разреженных данных (NLP, рекомендации)

PyTorch :

```
torch.optim.Adagrad(lr=0.01)
```

RMSProp (Root Mean Square Propagation)

- Развивает идею AdaGrad
- "Забывает" старые градиенты, сохраняя актуальные
- Поддерживает шаг обучения стабильным
- Хорошо работает на нестационарных задачах (где распределение данных меняется со временем) и при обучении нейросетей

PyTorch:
`torch.optim.RMSprop(lr=0.001)`

Adam (Adaptive Moment Estimation)

- Самый популярный оптимизатор для нейросетей
- Объединяет идеи Momentum и RMSProp
- Использует среднее и дисперсию градиентов
- Быстро сходится и устойчив к шуму
- Почти не требует ручной настройки

PyTorch:
`torch.optim.Adam(lr=0.001)`

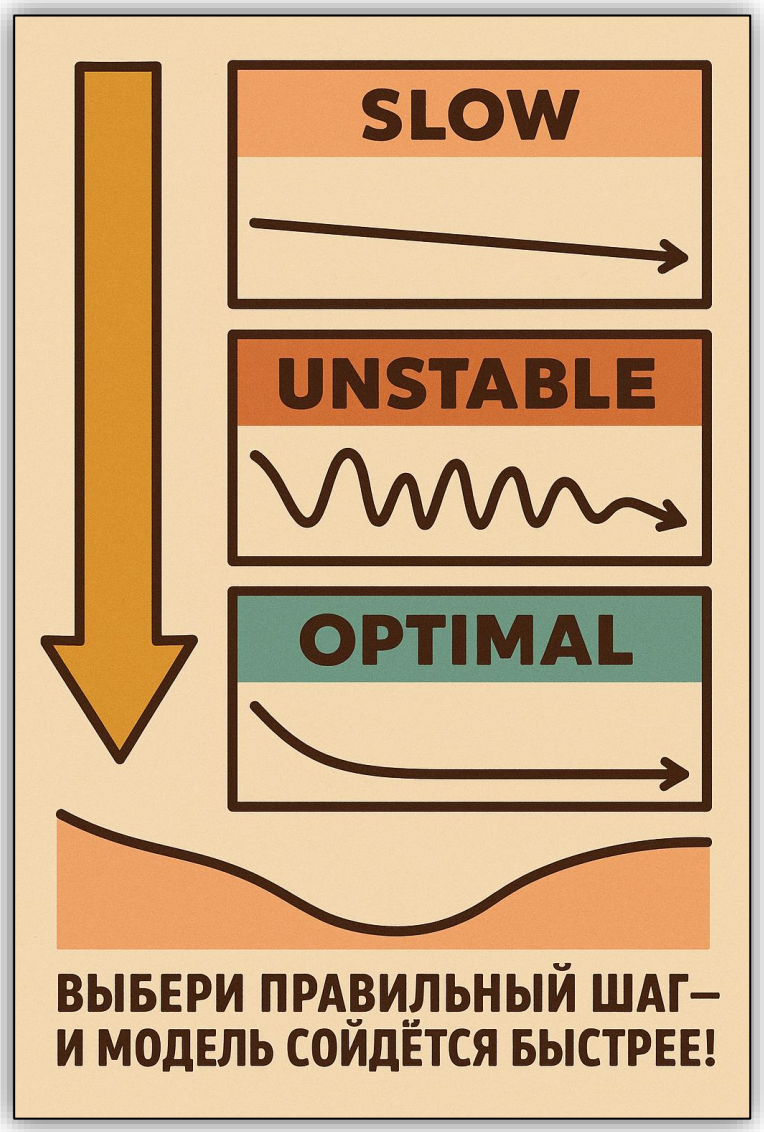
AdamW (Adam with Weight Decay)

- Модификация Adam с корректной регуляризацией весов
- Устраняет проблему "раздувания" весов при обучении
- Улучшает обобщающую способность модели
- Стал стандартом при обучении трансформеров и больших нейросетей

PyTorch:
`torch.optim.AdamW(lr=0.001, weight_decay=0.01)`

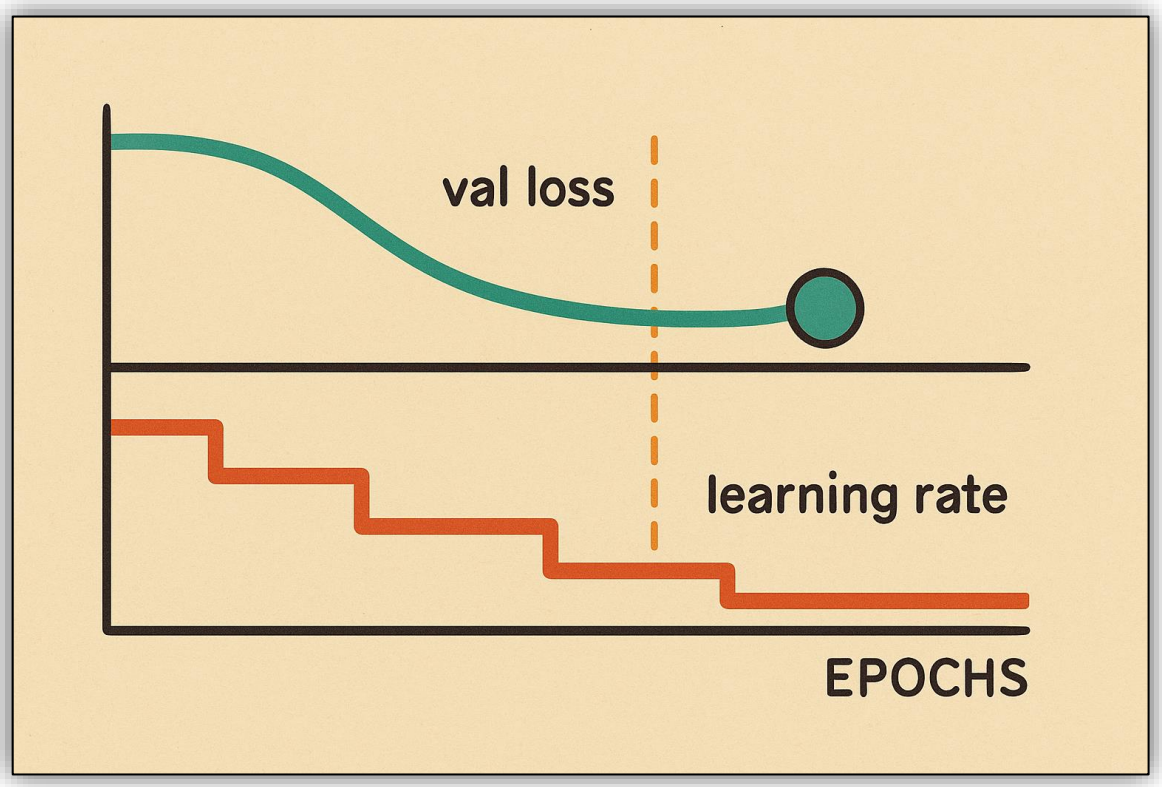
Выбор learning rate (шага обучения)

- Маленький **learning rate** → обучение идёт очень медленно
- Большой **learning rate** → "взрыв" loss, модель не сходится
- Важно найти баланс: быстрое, но стабильное обучение
- Полезно использовать **learning rate** schedule: step, cosine, warm-up
- Оптимальный **learning rate** подбирается экспериментально



Early Stopping и Scheduler

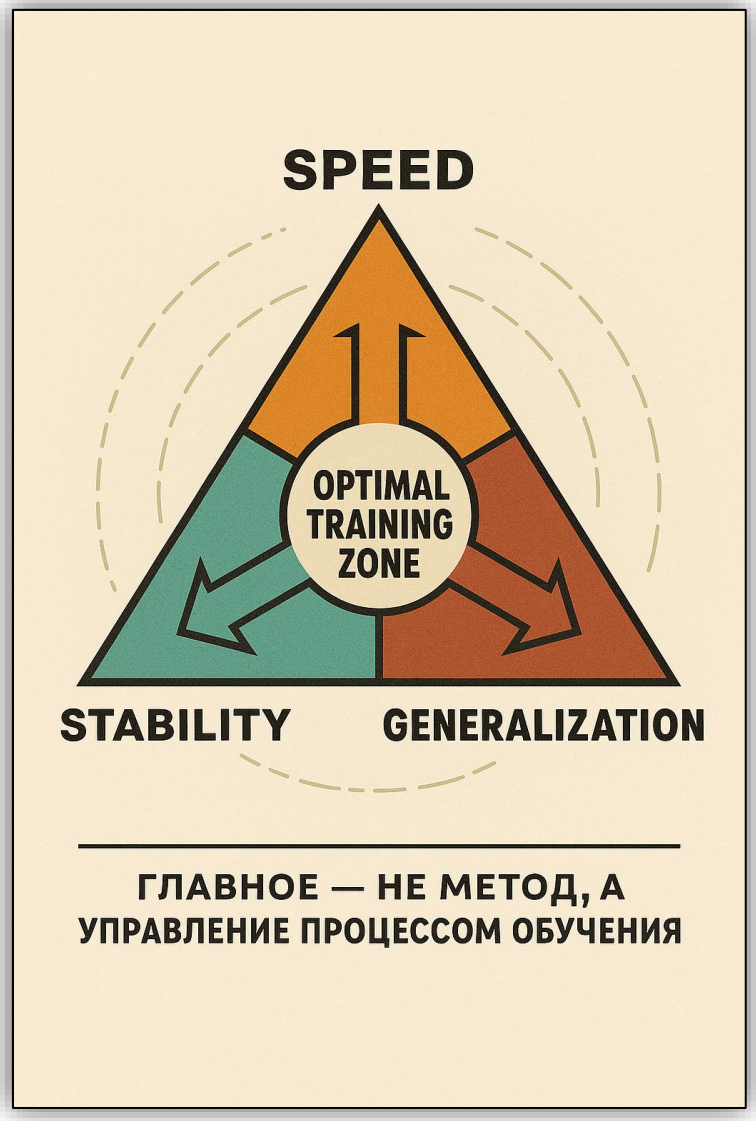
- Early Stopping — завершает обучение, если val loss не улучшается
  - предотвращает переобучение и экономит ресурсы
- Scheduler — управляет learning rate в процессе обучения
  - адаптирует скорость обучения по эпохам
  - StepLR, ExponentialLR, CosineAnnealingLR, Warm-up



PyTorch:

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
```

Итоги: методы оптимизации



- Цель оптимизации — найти параметры, минимизирующие функцию потерь
- Градиентный спуск — основа всех оптимизаторов
- Momentum и NAG — ускоряют сходимость
- AdaGrad, RMSProp, Adam — делают шаг обучения адаптивным
- AdamW — добавляет корректную регуляризацию весов
- Важно правильно подобрать lr, scheduler и early stopping