



> Конспект > 3 урок > Продвинутые техники обучения моделей

Содержание

Содержание

Нормализация

Какие бывают слои нормализации

Изменение поведения слоев нормализации в режимах обучения и применения

Dropout

Проблема нарушения баланса

Learning Rate Scheduler

Эксперименты

Воспроизводимость эксперимента

Как следовать хорошим практикам

Резюме

Практика

Dropout

Нормализация

Эксперименты

Подключение wandb

Как хранить конфиги

Полезный трюк: контролируем переобучение

Добиваемся воспроизводимости

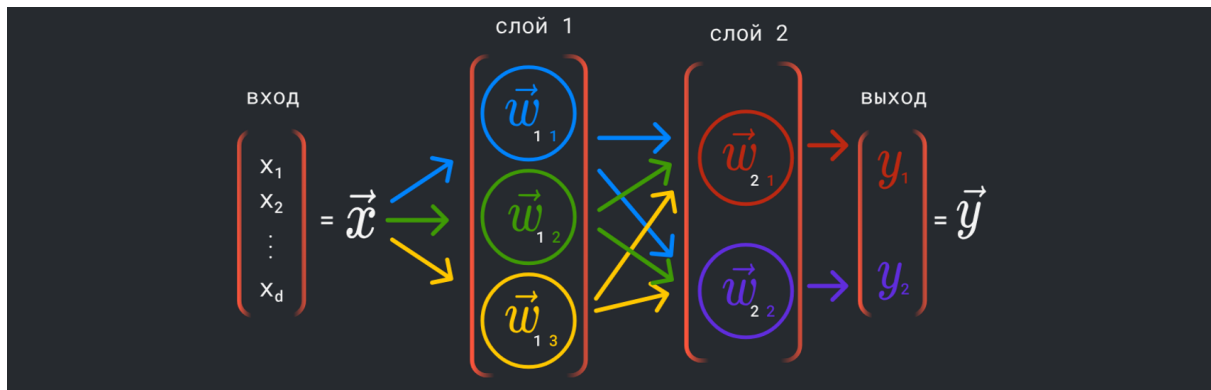
Пробуем новые подходы

Нормализация

Резюме

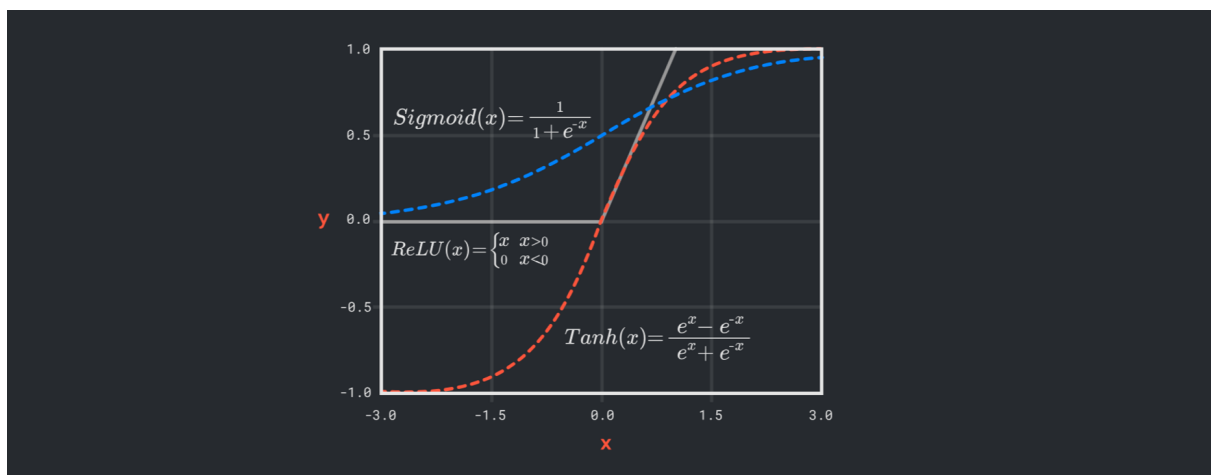
Нормализация

Для чего нужна нормализация



На прошлых занятиях мы обсуждали важность применения **нелинейных функций активации** после выхода полносвязных слоев в нейронных сетях. Это связано с тем, что добавление нескольких линейных слоев друг за другом эквивалентно одному линейному слою.

Использование нелинейных функций активации позволяет нашей модели извлекать более сложные и нелинейные зависимости в данных. Кроме того, они обычно отображают выход в некоторый отрезок, что может быть полезно для нормализации значений. Например, функция **сигмоиды** отображает произвольное число в отрезок от нуля до единицы. Это важно, поскольку нейронные сети предпочитают, чтобы значения находились в ограниченном диапазоне.



Теперь представим себе ситуацию, когда значения на выходе слоя начинают экспоненциально возрастать. В таком случае следующий слой может усилить эту тенденцию, значения могут выйти за пределы допустимого диапазона, что приведет к появлению значений NaN (Not a Number). Это может привести к нестабильности работы всей нейронной сети и ее коллапсу, так как NaN распространяется по всему вычислительному графу.

Такую ситуацию еще часто называют **Взрывом градиента**.

Как работает Нормализация

Нормализация — это важный инструмент в глубоком обучении, который помогает подогнать выходные значения модели в нужный диапазон, например, $[-1, 1]$, чтобы предотвратить нестабильность и распад сети.

Слой нормализации основан на математической концепции нормализации, которая включает вычитание среднего значения и деление на стандартное отклонение. В контексте нейронных сетей это означает следующее:

- Сначала вы вычисляете среднее значение по батчу входных данных. Это делается по каждой размерности батча. Например, если на вход поступает батч изображений размером $[100, 3]$, где 100 объектов имеют размерность 3, то вы вычисляете среднее для каждой из трех компонент.

$$\vec{\mu}_B = \frac{1}{N} \sum_{i=1}^N \vec{x}_i$$

- Затем вы вычисляете дисперсию, вычитая среднее значение и возводя результат в квадрат для каждой компоненты, это всегда будет одно число (скаляр).

$$\sigma_B^2 = \frac{1}{N} \sum_{i=1}^N (\vec{x}_i - \vec{\mu}_B)^2$$

- Далее выполняется стандартная нормализация — деление на корень из дисперсии. Однако дисперсия может быть нулевой, особенно при маленьких батчах данных, что приводит к делению на ноль. Чтобы избежать этой проблемы, к дисперсии добавляется небольшая константа (эпсилон), прежде чем брать корень. Это гарантирует, что не будет деления на ноль.

$$\hat{\vec{x}}_i = \frac{\vec{x}_i - \vec{\mu}_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Можно было бы этим ограничиться, отдавая на выход нормализованные значения, однако разработчики пошли дальше и предложили дополнительное улучшение этого процесса.

Они предложили умножать результат нормализации на коэффициент гамма и прибавлять к нему сдвиг бета. Важно отметить, что это умножение и прибавление происходят поэлементно, то есть каждый элемент выхода подвергается соответствующей операции с гаммой и бетой. Эти коэффициенты гамма и бета (γ , β) становятся **обучаемыми параметрами**.

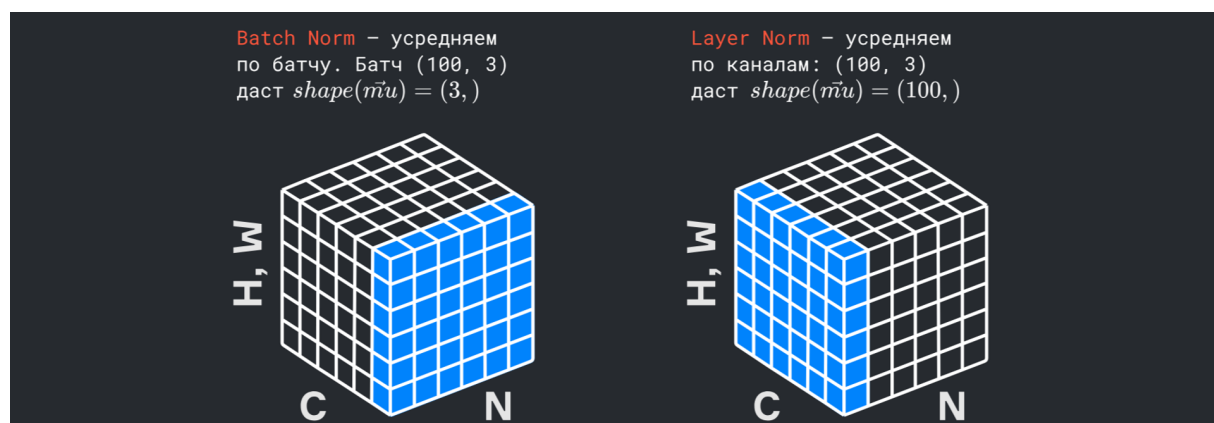
$$y_i = \vec{\gamma} * \hat{x}_i + \vec{\beta}$$

Мы можем ассоциировать этот процесс с тем, как в полносвязном слое у нас есть веса и смещения (**bias**), которые подвергаются обучению. В данном случае гамма и бета аналогичны этим параметрам: они настраиваются в процессе обучения сети.

Интересно, что поскольку гамма умножается поэлементно, а бета прибавляется поэлементно к нормализованному выходу, размерность этих коэффициентов такая же, как и у нормализованного выхода. Это означает, что каждый элемент выхода слоя нормализации подвергается умножению на свой коэффициент и сдвигу на соответствующее смещение.

Какие бывают слои нормализации

Рассмотрим два основных типа слоев нормализации: слой нормализации по батчу (Batch Normalization) и слой нормализации по слоям (Layer Normalization).



1. Batch Normalization (BatchNorm):

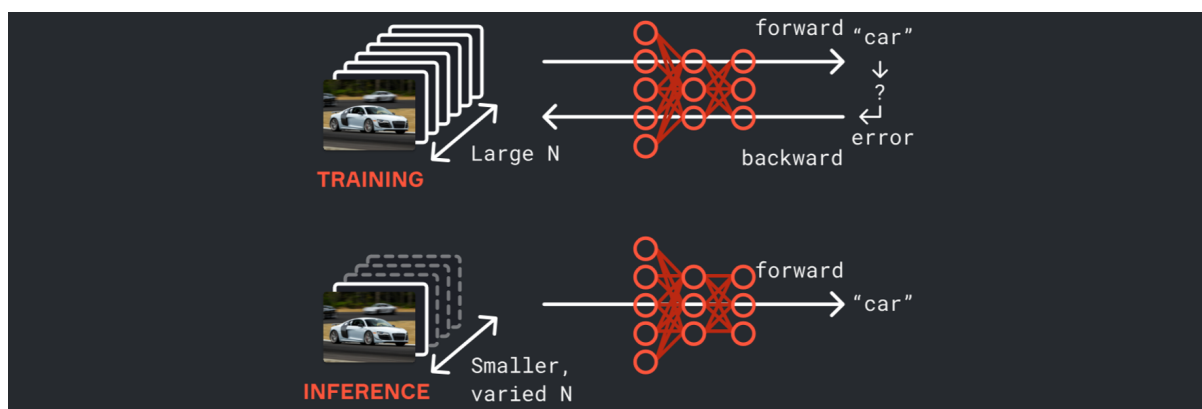
- **Принцип работы.** В BatchNorm нормализация выполняется по батчу данных. Для каждого признака вычисляется среднее и стандартное отклонение по всему батчу, затем данные нормализуются путем вычитания среднего и деления на стандартное отклонение.
- **Преимущества:**
 - Стабилизирует обучение: помогает в более быстрой и стабильной сходимости глубоких нейронных сетей.
 - Снижает проблему внутреннего сдвига (internal covariate shift): позволяет сети более эффективно обучаться, уменьшая зависимость между слоями.
- **Недостатки:**
 - Не подходит для небольших батчей: при работе с небольшими батчами, возможно переобучение или нестабильность обучения.
 - Затраты на вычисление: требует вычисления средних и стандартных отклонений на каждом шаге обучения.

2. Layer Normalization (LayerNorm):

- **Принцип работы.** В LayerNorm нормализация выполняется по слоям. Для каждого примера данных вычисляется среднее и стандартное отклонение по всем признакам в рамках одного примера.
- **Преимущества:**
 - Подходит для **маленьких батчей**: работает стабильно даже с небольшими батчами данных.
 - Независимость от размера батча: не требует оценки статистик батча, поэтому сеть может работать на новых примерах данных без необходимости пересчета статистик.
- **Недостатки:**
 - Может быть менее эффективным: в некоторых случаях LayerNorm может работать менее эффективно, чем BatchNorm, особенно при больших батчах данных.
 - Может уменьшить вариативность: в некоторых сценариях LayerNorm может снизить вариативность данных, что может привести к уменьшению выразительности модели.

В целом, выбор между Batch Normalization и Layer Normalization зависит от конкретного контекста задачи, особенностей данных и требований к производительности.

Изменение поведения слоев нормализации в режимах обучения и применения



Следующая особенность слоев нормализации касается их поведения в режиме обучения и применения.

Проблема:

- При использовании слоев нормализации в режиме обучения мы рассчитываем средние и дисперсии по батчу данных.
- Однако, когда модель переходит в режим применения, возникает вопрос: как использовать статистику, учитывая, что модель видела большое количество данных в процессе обучения.

Решение:

- В режиме обучения (train) слой нормализации берет статистики только по текущему батчу данных.
- В режиме применения (eval или inference) слой нормализации использует статистику по всем прошлым данным, чтобы обеспечить более точную оценку средних и дисперсий.

Пример:

- Например, во время обучения модель видела миллионы изображений или текстовых данных.
- В режиме обучения слой нормализации учитывает только статистику текущего батча данных.
- Однако, при применении модели мы хотим использовать более точные статистики, основанные на всех доступных данных, а не только на текущем

батче.

Таким образом, различное поведение слоев нормализации в режимах обучения и применения позволяет использовать более точные статистики в процессе применения модели. Это достигается за счет использования статистики по всем прошлым данным в режиме применения, в отличие от статистики по текущему батчу в режиме обучения.

Каким образом считается эта статистика?

Предположим, что во время обучения модель видела большой объем данных.

В режиме обучения слой нормализации не только учитывает среднее по текущему батчу, но и обновляет скользящее среднее, чтобы включить в него информацию о всех данных, просмотренных ранее:

$$\vec{\mu}_g^{(new)} = \alpha \vec{\mu}_g^{(old)} + (1 - \alpha) \vec{\mu}_B, \text{ где}$$

$\alpha \vec{\mu}_g^{(old)}$ — текущее значение скользящего среднего

$\vec{\mu}_B$ — среднее по батчу

α — коэффициент затухания, который определяет вес текущего батча относительно предыдущего скользящего среднего, принимает значения в диапазоне $[0, 1]$.

Тем самым мы как бы замешиваем таким образом текущее среднее с новыми данными, частично забывая старое значение скользящего среднего и принимая новое.

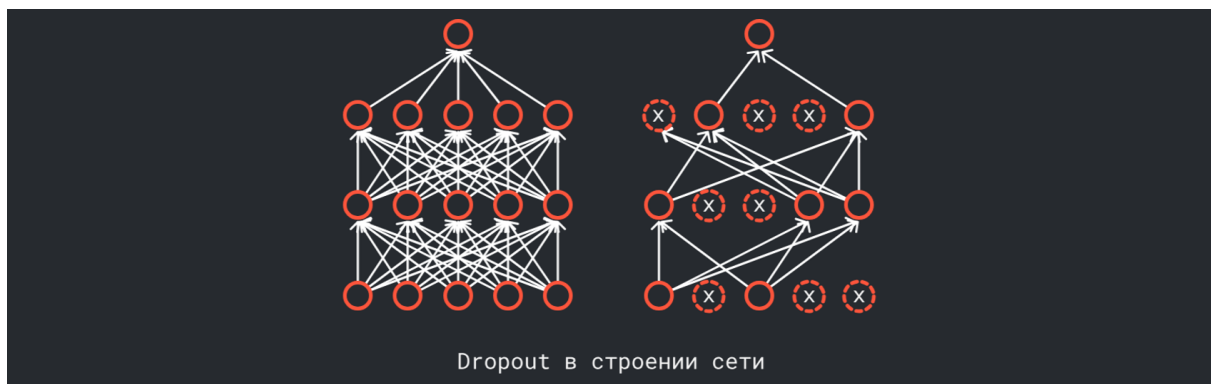
То же самое делаем для дисперсии:

$$\sigma_g^2 = \alpha (\sigma_g^2)^{(old)} + (1 - \alpha) \sigma_B^2$$

Таким образом, когда вы проходите по сети в режиме обучения (train), значение скользящего среднего аккумулирует в себе информацию обо всех средних всех батчей.

На этапе применения модели (inference, eval) значения среднего и дисперсии уже не вычисляются по текущему батчу, а берутся из скользящего среднего.

Dropout



Слой **Dropout** представляет собой метод регуляризации, применяемый в нейронных сетях для борьбы с переобучением.

Когда он нужен?

- В нейронных сетях с большим количеством параметров может возникнуть соблазн «заучить» все данные обучающей выборки, что приведет к переобучению.
- Представьте ситуацию, где число параметров сопоставимо с количеством объектов в обучающей выборке. В этом случае каждый параметр сети будет «видеть» только один объект из выборки.

Принцип работы:

- На каждом слое сети применяется Dropout, который случайным образом «отключает» некоторые нейроны, превращая их выход в ноль с вероятностью p .
- То есть, для каждого элемента выхода предыдущего слоя выпадает монетка: если выпадает «орел» (вероятность p), значение остается, а если «решка» (вероятность $1-p$), значение превращается в ноль.
- Это позволяет предотвратить нейросеть от слишком сильной адаптации к обучающим данным и улучшает ее обобщающую способность.

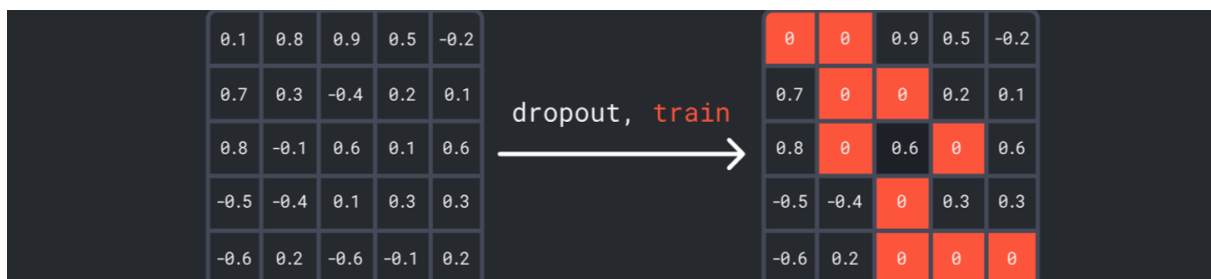
Зачем это нужно?

- Dropout помогает сделать нейросеть менее чувствительной к конкретным данным обучающей выборки, улучшая ее обобщающую способность.
- Это позволяет модели лучше обобщать данные, что особенно полезно в случае ограниченного количества обучающих примеров или в случае наличия шума в данных.

Преимущества и особенности:

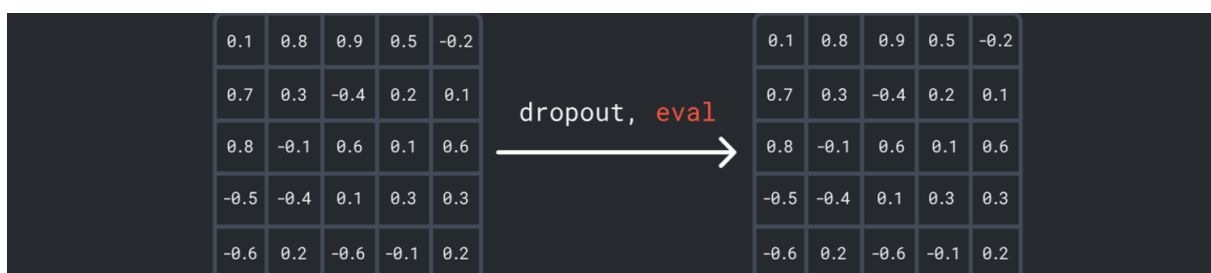
- Помогает предотвратить переобучение, улучшая обобщающую способность модели.
- Применение Dropout делает нейросеть более устойчивой к изменениям в данных и улучшает ее обобщающую способность.
- Эффективен в случае наличия большого количества параметров в сети или небольшого количества обучающих данных.
- Обычно применяется во время обучения, а при инференсе слой Dropout обычно отключается.

Зануление элементов происходит только на этапе обучения!



Dropout в режиме обучения

Когда мы применяем модель, dropout ничего не зануляет.



Dropout в режиме применения

Проблема нарушения баланса

- На тренировке модель «выкручивает» веса, чтобы извлекать информацию из ослабленного сигнала ($s \cdot p$, где p — вероятность отключения нейронов в

Dropout).

- На инференсе поступает больше данных, что нарушает баланс: модель, настроенная на меньший сигнал, может работать менее эффективно с более крупными данными.

Решение:

1. Увеличивающий коэффициент на этапе тренировки:

- На этапе тренировки обнуляются часть выходов из Dropout, но оставшиеся умножаются на увеличивающий коэффициент.
- Этот коэффициент можно вычислить как единица, деленная на вероятность выживания $\frac{1}{(1-p)}$

2. Уменьшающий коэффициент на этапе инференса:

- На этапе инференса, чтобы скомпенсировать увеличение сигнала, его умножают на уменьшающий коэффициент.
- Этот коэффициент вычисляется как вероятность выживания $(1 - p)$.

На этапе train	На этапе inference
Умножаем выход на $\frac{1}{1-p}$	Ничего не делаем
ЛИБО	
Ничего не делаем	Умножаем на $(1-p)$
Итого $out_{eval} = \underbrace{(1-p)}_{\text{доля оставшихся}} * \underbrace{1/(1-p)}_{\text{нормируем}} * out_{train} = out_{train}$ в обоих случаях.	

Таким образом, как увеличивающий, так и уменьшающий коэффициенты компенсируют потерю сигнала на этапе тренировки и изменение его мощности на этапе инференса.

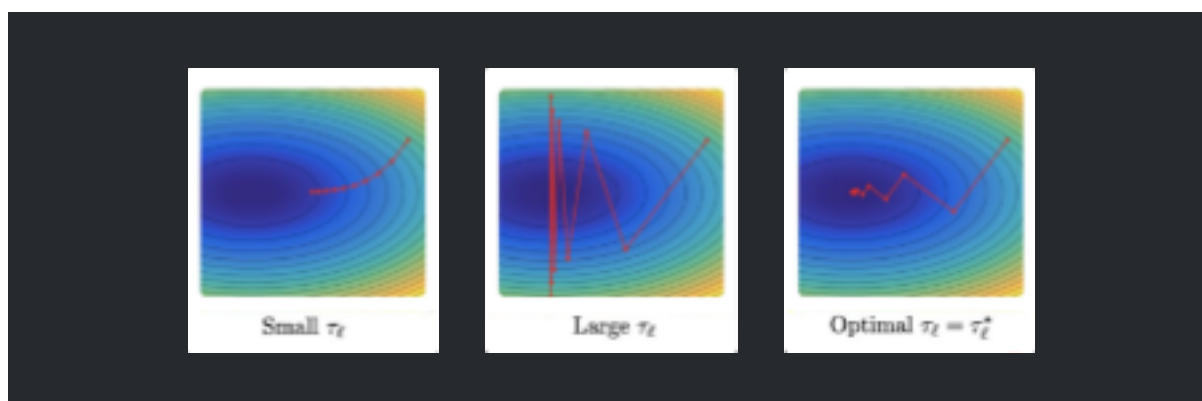
Это позволяет сохранить силу сигнала и избежать ухудшения работы модели на этапе инференса.

Learning Rate Scheduler

Когда мы обучаем нейронные сети или любые модели машинного обучения, мы часто используем градиентный спуск. Этот метод позволяет минимизировать функцию потерь, обновляя веса модели в направлении, противоположном градиенту функции потерь. Однако установка слишком большого или маленького значения для шага градиента, называемого **learning rate**, может привести к проблемам.

Если установить learning rate слишком большим, это может привести к тому, что изменения весов будут слишком большими, и модель "разбежится", как показано на графике. В таком случае, процесс обучения становится нестабильным, и модель может не сойтись к оптимальным значениям.

С другой стороны, если установить learning rate слишком маленьким, обучение может замедлиться, и сходимость модели к оптимуму потребует большого количества эпох. Также это может привести к застреванию в локальных оптимумах.



Однако, по мере того как сеть начинает приближаться к оптимуму, становится важно уменьшать learning rate, чтобы обучение приближалось к желаемому результату. Это аналогично тому, как поезд замедляется перед тем, как припарковаться на вокзале. Начиная быстро и приближаясь к цели, поезд затем замедляется, чтобы точно остановиться в нужном месте.

Таким образом, возникает идея изменять learning rate по мере обучения. Этот процесс называется Learning Rate Scheduler (планировщик скорости обучения).

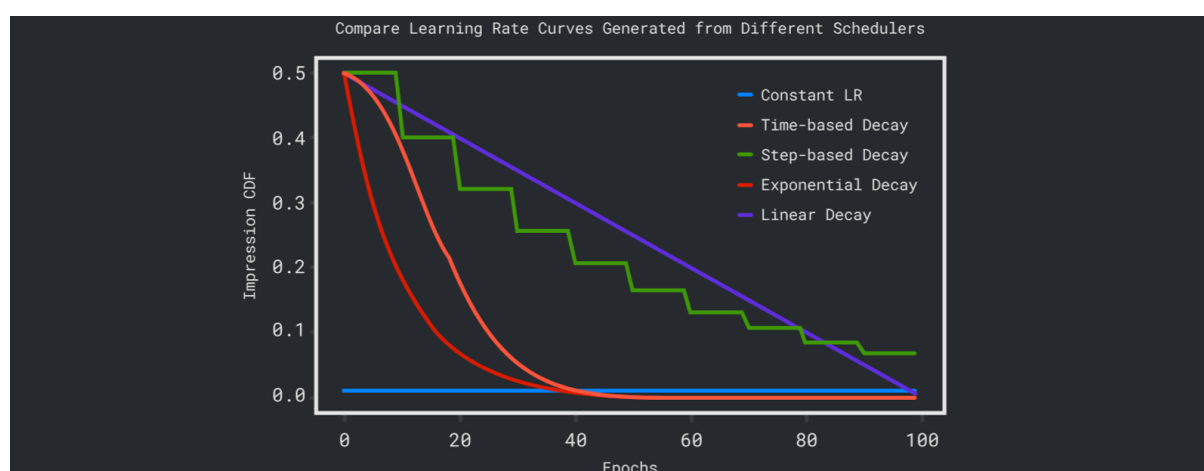
Learning Rate Scheduler — это объект или алгоритм в Python, который отвечает за изменение learning rate в процессе обучения нейронной сети. Его суть заключается в том, чтобы адаптировать скорость обучения в зависимости от текущего состояния обучения.

На входе Learning Rate Scheduler принимает начальное значение **learning rate**, а затем изменяет его в соответствии с определенными внутренними

закономерностями в ходе обучения.

Существует несколько способов изменения learning rate, которые могут быть использованы в Learning Rate Scheduler. Например:

- Constant Learning Rate — константное значение learning rate на протяжении всего обучения.
- Убывание learning rate — learning rate уменьшается с течением времени, может происходить постепенно или через равные интервалы.
- Линейное убывание — learning rate уменьшается линейно с течением времени до определенного значения.






В PyTorch есть специальные классы Learning Rate Scheduler, который позволяет использовать различные методы изменения learning rate и эффективно управлять ими в процессе обучения нейронных сетей.




Эксперименты

После изучения таких концепций, как **Batch Normalization**, **Dropout** и **Learning Rate Scheduler**, мы понимаем, что они могут существенно улучшить работу наших нейронных сетей. Однако когда мы формулируем гипотезы и пробуем различные подходы, мы обязательно сталкиваемся с множеством экспериментов. И это может стать сложным, так как каждое изменение может повлиять на работу модели, а копирование других подходов не всегда эффективно.

Поэтому важно иметь систему правил, чтобы не запутаться в процессе исследования и настройки моделей.

- Первое и, вероятно, самое важное правило — в каждом эксперименте следует изменять только один параметр или компонент модели. Это позволяет четко определить, какие изменения приводят к улучшению или ухудшению результатов.
- Важно визуализировать результаты экспериментов, рисуя графики и сохраняя метрики. Это помогает наглядно отслеживать изменения в работе модели и делать выводы о ее эффективности.
- Использовать один и тот же набор графиков и метрик — так их можно будет сравнивать между собой.
- Также важно учитывать, что подходы, которые работают в одном случае, могут быть неэффективны в другом. Поэтому необходимо постоянно проверять гипотезы, смотреть на метрики и адаптировать стратегию в соответствии с полученными результатами.
- Систематический подход к экспериментам и строгое соблюдение правил помогут избежать путаницы и эффективно настраивать нейронные сети для достижения желаемых результатов.

	
<ul style="list-style-type: none"> – Добавить один слой в сеть, запустить – Поменять LR, запустить – Поменять SGD на Adam, запустить 	<p>«Оно долго учится, давай я сразу и слой добавлю, и LR поменяю»</p> <p>«Ой, да чего там Adam поменяет, давай вместе с ним LR поменяю еще»</p>
<div style="border: 1px solid orange; padding: 5px; display: inline-block;">  Сложно сказать, что повлияло на метрики, если было несколько изменений </div>	

	
<ul style="list-style-type: none"> – По графикам вы увидели, что обучение сошло – По метрике вы видите, что пробили бейзлайн 	<p>«Наверное, не переобучилась, вряд ли еще один слой сломает все»</p> <p>«Ну, я чуть-чуть уменьшил LR, так что обучение должно по-прежнему сойтись»</p>
<div style="border: 1px solid orange; padding: 5px; display: inline-block;">  Без графиков и метрик остается только гадать, как ведет себя модель </div>	

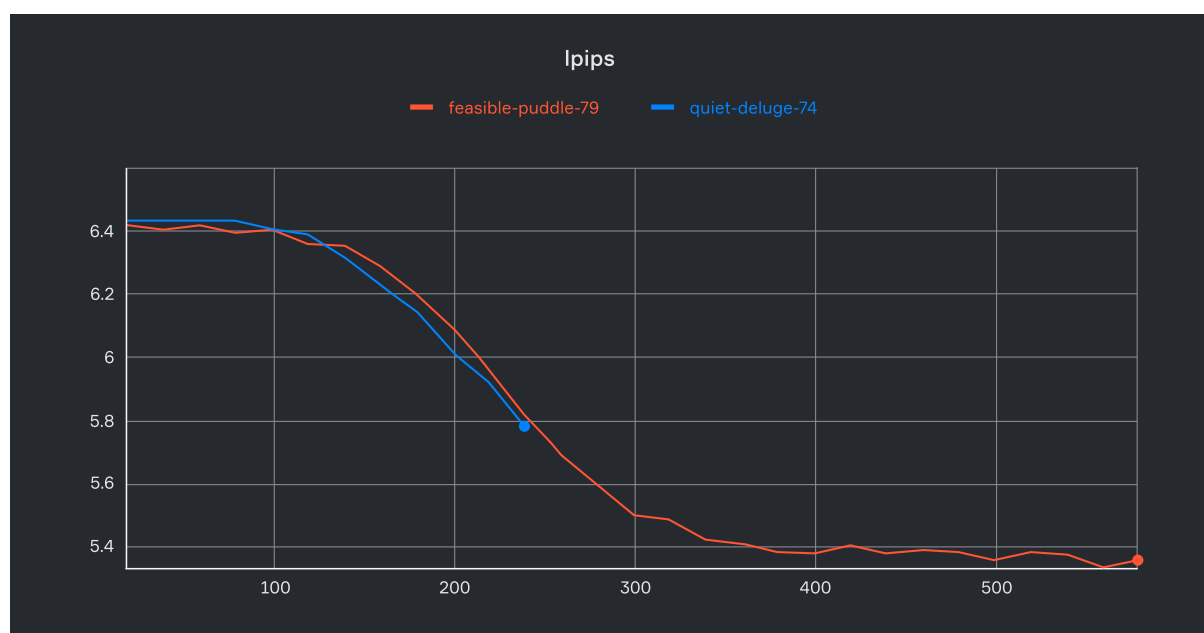
Воспроизводимость эксперимента

Воспроизводимость — это ключевое свойство не только в машинном обучении, но и в программировании в целом. Это означает, что при использовании одного и того же кода и данных результаты должны быть одинаковыми при каждом запуске.

В мире глубокого обучения это свойство особенно важно, но его достижение не всегда легко. Даже если код и данные остаются неизменными, результаты могут сильно отличаться от запуска к запуску. Это может привести к затруднениям в понимании работы модели и принятии решений на основе полученных результатов.

На практике во время прототипирования модели важно иметь воспроизводимые результаты. Это позволяет исследователям и разработчикам уверенно оценивать эффективность различных подходов и принимать обоснованные решения.

Пример невоспроизводимого эксперимента иллюстрирует, как при одинаковых условиях результаты могут значительно отличаться. Даже если графики похожи, они могут иметь разные траектории и нести различные значения метрик. Это может привести к неправильным выводам о качестве модели и привести к ложным улучшениям или ухудшениям.




Пример невоспроизводимого эксперимента


С другой стороны, воспроизводимый эксперимент демонстрирует, что результаты остаются стабильными даже при повторном запуске. Это позволяет с большей уверенностью судить о влиянии изменений и эффективности применяемых методов.

Таким образом, воспроизводимость играет важную роль в оценке и разработке моделей машинного обучения, обеспечивая надежность и консистентность

результатов.



- Метрики и графики сохраняются в облако
- Все из команды могут ознакомиться с результатами
- К коду эксперимента можно вернуться в любой момент
- Любой может **воспроизвести эксперимент**



«Напишу в чатик, что все ок с экспериментом, можно выкатывать»

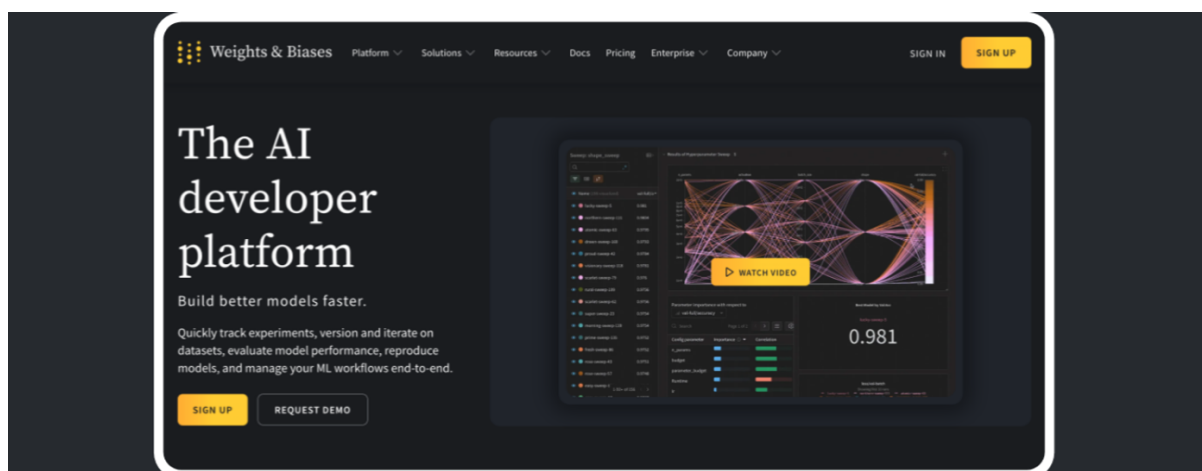
«На дейлике покажу скриншот с моего ноута, пусть поверят мне»

«А я удалил уже файлы с метриками, ведь они уменьшились»

Как следовать хорошим практикам

Основные моменты, о которых следует помнить, чтобы следовать правилам экспериментов и обеспечить воспроизводимость результатов:

1. **Один эксперимент — одно изменение.** В каждом эксперименте изменяйте только один параметр или компонент модели, чтобы понять его эффект.
2. **Единый набор графиков и метрик.** Определите заранее набор графиков и метрик для каждого эксперимента и придерживайтесь их одинаково для сравнения результатов.
3. **Сохранение графиков и метрик.** Используйте инструменты, такие как [wandb](#) (Weights and Biases), для автоматического сохранения и сравнения метрик и графиков. позволяет легко отслеживать результаты и делиться ими с коллегами.



- Wandb бесплатный для личного использования
- Легко вызвать в коде:

```
wandb.log({'metric_1':value_1})
```

- Автоматически рисует все графики
- Хранит все запуски

4. **Обеспечение воспроизводимости.** Фиксируйте все случайные семена и параметры для обеспечения воспроизводимости результатов. Также убедитесь, что данные и алгоритмы, которые могут быть недетерминированными (например, алгоритмы на видеокартах), также фиксированы для воспроизводимости.
5. **Прирост производительности за счет воспроизводимости.** Помните, что обеспечение воспроизводимости может потребовать дополнительных усилий и вложений, но это гарантирует надежность и консистентность результатов, особенно на этапе экспериментов.
6. **Компромисс между воспроизводимостью и производительностью в продакшене.** В продакшене иногда приходится идти на компромисс между воспроизводимостью и производительностью, учитывая, что в продакшене часто важнее скорость работы решения. Поэтому на этапе продакшена воспроизводимость часто отключают.

Соблюдение этих правил поможет вам уверенно проводить эксперименты, сравнивать различные подходы и добиваться стабильных результатов.

Резюме

В ходе урока мы ознакомились с несколькими важными аспектами глубокого обучения:

1. **Нормализация.** Изучили два распространенных метода нормализации данных в нейронных сетях — Batch Normalization (Batch Norm) и Layer Normalization (LayerNorm). Узнали, что Batch Norm эффективнее при использовании больших батчей данных, в то время как LayerNorm может быть предпочтительнее при работе с небольшими батчами, особенно в области обработки естественного языка (NLP).
2. **Dropout.** Изучили принцип работы слоя Dropout, который помогает предотвратить переобучение путем случайного исключения некоторых нейронов в процессе обучения.

3. **Learning Rate Scheduler.** Узнали о том, что такое Learning Rate Scheduler и как он может быть использован для динамической регуляции скорости обучения в течение процесса обучения нейронной сети.
4. **Эксперименты и воспроизводимость.** Рассмотрели важность проведения экспериментов в глубоком обучении и техники, которые помогают сохранить воспроизводимость результатов, такие как установка случайного зерна и фиксация гиперпараметров.

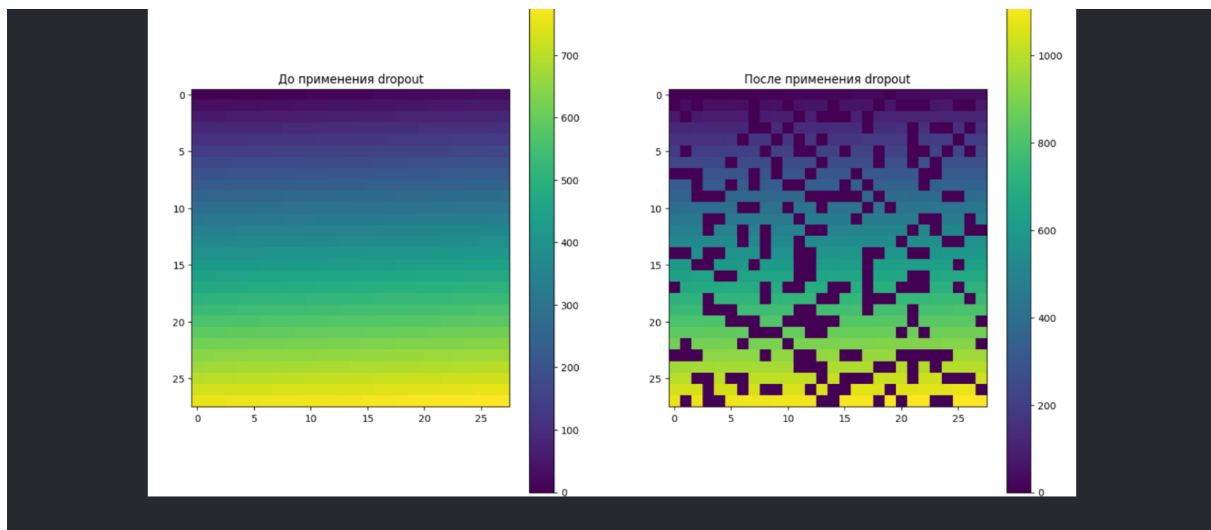
Эти основные концепции и инструменты играют ключевую роль в процессе обучения и оптимизации нейронных сетей, позволяя исследователям и практикам эффективно работать с данными и моделями.

Практика

Dropout

Напомним, что Dropout берет входной тензор и зануляет каждый элемент с вероятностью p . Эта вероятность p — гиперпараметр. Она задается заранее и не обучается на данных.

```
dropout_layer = nn.Dropout(p=0.3)
# Визуализируем тензор со случайным распределением до и после примен
fig, ax = plt.subplots(1, 2, figsize=(16, 9))
plt.colorbar(ax[0].imshow(tensor.numpy()), ax=ax[0])
ax[0].set_title("До применения dropout")
tensor_after_dropout = dropout_layer(tensor)
plt.colorbar(ax[1].imshow(tensor_after_dropout.numpy()), ax=ax[1])
ax[1].set_title("После применения dropout");
```



Видим, что некоторые пиксели как бы "выбились", т.е. обнулились.

Теперь заведем тензор той же размерности $[28, 28]$ и на этот раз заполним его единицами:

```
dropout_layer.train() #перевели слой в режим обучения
ones = torch.ones((28, 28))
fig, ax = plt.subplots(1, 2, figsize=(16, 9))
after_dropout = dropout_layer(ones)
plt.colorbar(ax[0].imshow(after_dropout.numpy()), ax=ax[0])
# В режиме train Dropout незануленные значения увеличивает в 1.42 =
ax[0].set_title(
    f"Dropout в режиме train, max(after_dropout)={torch.max(after_dr"
)
```

В режиме eval слой dropout просто пропускает сигнал в неизменном виде (как будто слоя dropout нет вовсе). Перевести слой в режим inference можно через:

```
dropout_layer.eval()
```

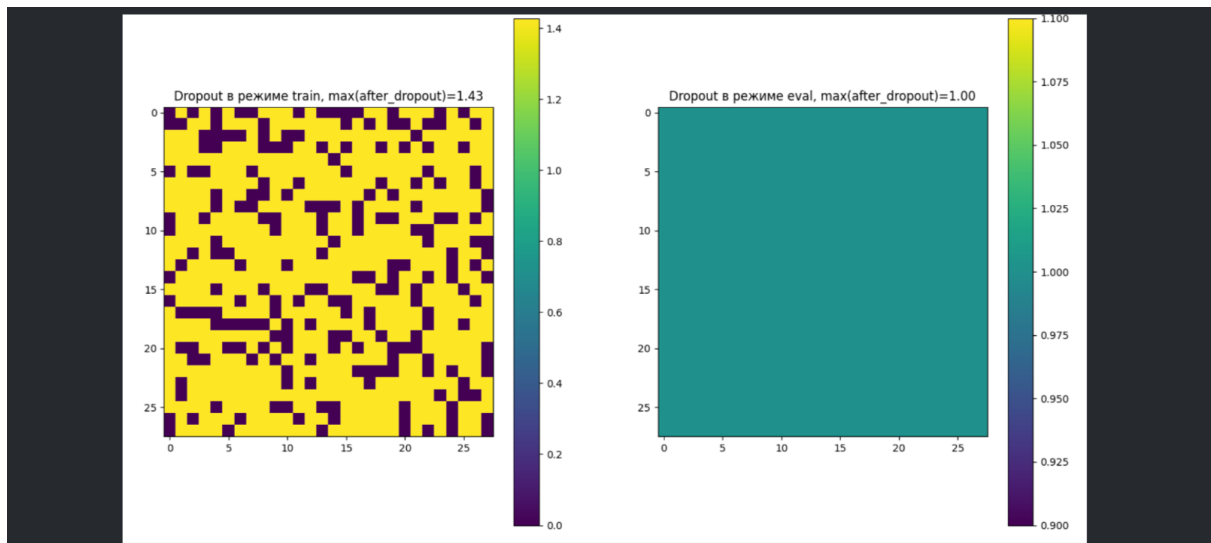
Либо

```
dropout_layer.train(False)
```

Визуализируем и увидим, что теперь не умножается на 1.42:

```
after_dropout = dropout_layer(ones)
plt.colorbar(ax[1].imshow(dropout_layer(ones).numpy()), ax=ax[1])
```

```
ax[1].set_title(
    f'Dropout в режиме eval, max(after_dropout)={torch.max(after_dropout)}');
```



Выход слоя dropout в режиме train и eval

Нормализация

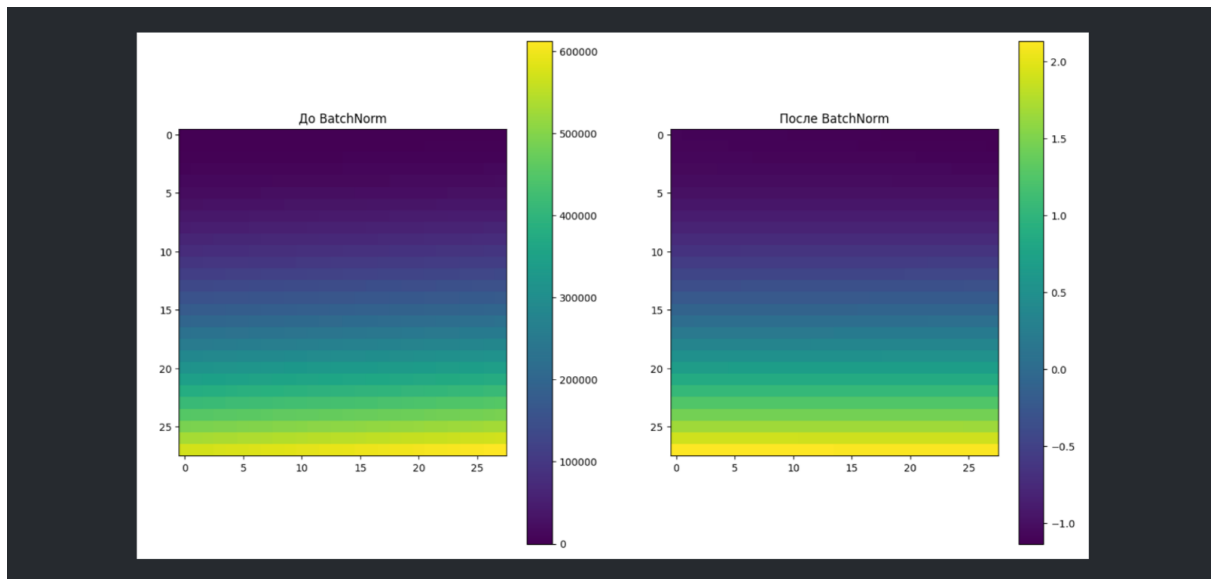
Существует 2 вида BatchNorm — $1d$ и $2d$.

$1d$ — это вектора (N, d),

$2d$ — это картинки (статистики считать будет по-другому)

Как создать слой нормализации:

```
batch_norm_layer = nn.BatchNorm1d(
    num_features=28,
    # momentum = 1 - alpha
    momentum=0.1,
    eps=1e-5,
)
# Два обучаемых параметра - weight (gamma) и bias (beta)
print(*batch_norm_layer.named_parameters(), sep="\n")
requires_grad=True # означает, что мы имеем дело с обучаемыми параме
```



Выход слоя без нормализации и после

По графикам можно увидеть два идентичных распределения, однако разброс второго распределения значительно меньше $(-1, 2)$, а значит модель с меньшей вероятностью разойдется.

Эксперименты

Вернемся к модели классификации изображений из прошлого урока и попробуем ее улучшить. А именно:

- настроим сохранение метрик в wandb, обучим бейзлайн и сохраним его метрики;
- добьемся воспроизводимости обучения;
- попробуем применить LR Scheduler, сравним метрики;
- попробуем добавить Dropout и Batch Normalization, сравним метрики;

Подключение wandb

```
#импорт необходимых библиотек
import tqdm
import wandb
import time
```

Теперь можно запускать любой код и логировать любые метрики. Пример, как это может выглядеть:

```
def simple_train_loop():
    # Сначала нужно вызвать wandb.init()
    # Это создаст эксперимент в wandb.ai и привяжет его к текущему з
    wandb.init(
        # Более детальное описание аргументов: https://docs.wandb.ai/gui
        project="my-first-wandb-project",
        notes="I created it in my DL course",
        # Можно так же передать config - словарь с любым содержимым.
        # Обычно туда кладут гиперпараметры, настройки обработки данных,
        config={"seed": 0, "my-custom_string": "asb"},
    )
    # Теперь запускаем наш код обучения, подготовки данных и т.п. ка
    for i in tqdm.trange(300):
        # Имитируем долгое обучение
        time.sleep(0.1)
        # Нужно добавить эту строку, чтобы записать в wandb
        wandb.log({"iteration": i, "loss": 12 - i ** 0.5})
        # Запуск автоматически завершится, когда скрипт (т.е. но
    # Но можно явно завершить:
    wandb.finish()
```

Как хранить конфиги

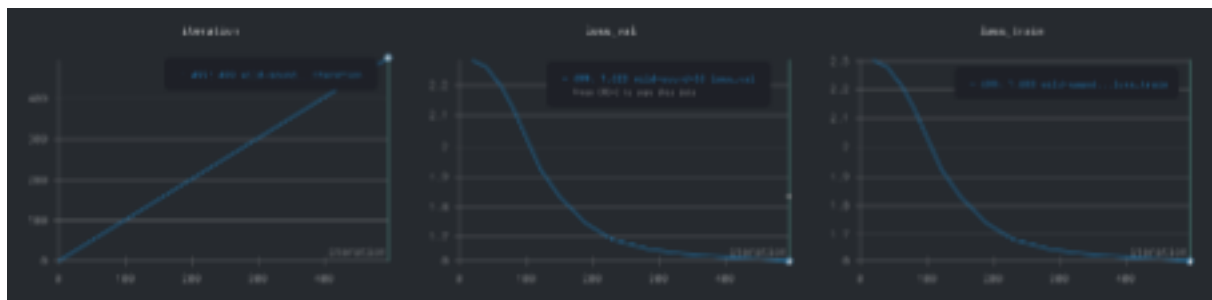
Конфиги можно хранить в словаре, а можно создать отдельный класс (более удобно, т.к. будут подсказки в редакторе):

```
@dataclass
class TrainConfig:
    eval_every: int = 10
    lr: float = 1e-2
    total_iterations: int = 3000
```

Полезный трюк: контролируем переобучение

Поскольку качество в процессе обучения оценивается только по тренировочной выборке, будет полезно каждые n итераций проверять качество на валидационной выборке, чтобы убедиться, что модель не переобучается:

```
if (i + 1) % config.eval_every == 0:
    with torch.no_grad():
        model.eval()
        loss_val = F.cross_entropy(model(X_val), y_val)
        model.train()
        metrics.update({"loss_val": loss_val.cpu().item()})
```



Добиваемся воспроизводимости

Чтобы быть уверенными, что изменения при перезапуске кода являются результатом нашего эксперимента (добавление слоев, настройка гиперпараметров и тп), а не случайности, необходимо эту случайность зафиксировать везде, где она появляется:

- разбиение на train/val/test — это видно по `shuffle=True`;
- инициализация весов модели - веса всех слоев генерируются из случайного распределения;
- могут быть алгоритмы внутри слоев, но у нас таких нету.

Чтобы добиться воспроизводимости, нужно:

1. Добавить `random_state=...` в `train_test_split` — он умеет принимать такой параметр.
2. Зафиксировать seed у PyTorch **перед** созданием модели через `torch.random.manual_seed`.

Пробуем новые подходы

Добавим в конфиг два новых типа (LR Scheduler и Adam):

```
import typing as tp
from torch.optim.lr_scheduler import ExponentialLR, LinearLR, StepLR
from torch.optim.adam import Adam

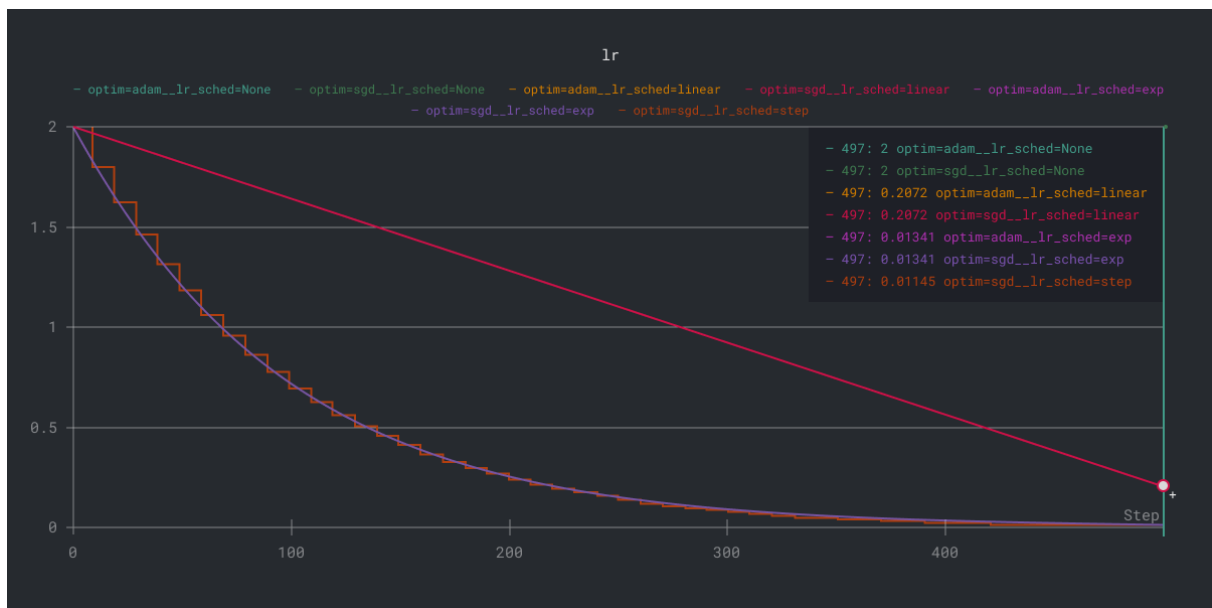
@dataclass
class TrainConfig:
    eval_every: int = 10
    lr: float = 1e-2
    total_iterations: int = 3000
    scheduler_type: tp.Literal["exp", "linear", "step"] | None = None
    optimizer_type: tp.Literal["sgd", "adam"] = "sgd"
```

Создаем разный **optimizer** в зависимости от конфига:

```
if config.optimizer_type == "sgd":
    optim = SGD(model.parameters(), lr=config.lr)
else:
    optim = Adam(model.parameters(), lr=config.lr)
```

Для scheduler точно так же как и для оптимайзера нужно отдельно вызвать `.step()`, но после обучения и валидации (см. пример в [документации](#))

```
if scheduler is not None:
    scheduler.step()
    metrics.update({"lr": scheduler.get_last_lr()[0]})
else:
    # Чтобы иметь одинаковый набор графиков в wandb
    metrics.update({"lr": config.lr})
    wandb.log(metrics)
```



Три разных шедулера



График лосса на трейне с разными шедулерами



$$\bar{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}},$$

а γ и β - обучаемые параметры,

ε - гиперпараметр, обычно это 10^{-5} (нужен, чтобы не было деления на ноль).

- Помимо этого Batch Normalization считает скользящее среднее по всем батчам, которые через него прошли. Делается это так:

1. Берем $\mu = 0, \sigma = 0$.
2. Как только проходит новый батч, считаем среднее μ_B и дисперсию σ_B^2 по нему.
3. Обновляем $\mu \leftarrow \alpha\mu + (1 - \alpha)\mu_B$. Здесь α — гиперпараметр.
4. Обновляем $\sigma \leftarrow \alpha\sigma + (1 - \alpha)\sigma_B$.

В режиме **eval/inference** слой Batch Normalization фиксирует γ и β (которые обучались ранее) и преобразовывает все проходящие через него батчи по формуле:

$$y = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

т.е. по сути то же самое, что и в

`train`, только среднее и дисперсия берутся глобальные по всем данным.

Придерживаемся правила один эксперимент – одно изменение и убираем предыдущее добавление Dropout и добавляем Batchnorm:

```
class BatchNormModel(nn.Module):
    def __init__(self, num_classes: int, p_dropout: float = 0.5):
        super().__init__()
        hidden_dim = 256
        self.net = nn.Sequential(
            nn.Linear(in_features=28 * 28, out_features=hidden_dim),
            nn.ReLU(),
            nn.BatchNorm1d(num_features=hidden_dim),
            nn.Linear(in_features=hidden_dim, out_features=num_classes),
            nn.Softmax(dim=1),
        )

    def forward(self, x: torch.Tensor):
        x = x.reshape((-1, 28 * 28))
        return self.net(x)
```

```

torch.random.manual_seed(seed)
config = TrainConfig(
    eval_every=20,
    lr=2,
    total_iterations=500,
    scheduler_type=None,
    optimizer_type="sgd",
)

model = BatchNormModel(num_classes=len(ohe.classes_))
optim = train_loop(
    model, X_train, y_train, X_val, y_val, config=config, run_name="
)

```



Видим, что даже без слоя дропаута модель сошлась значительно быстрее и достигла оптимума.

Резюме

1. Посмотрели на работу Batch Normalization и Dropout в PyTorch.
2. Познакомились с wandb и тем, как с его помощью логировать метрики и графики.
3. Узнали, как добиваться воспроизводимости на практике.
4. Познакомились в LR Scheduler, попробовали его в эксперименте.

5. Попробовали Batch Normalization и Dropout в имеющейся сети, сравнили качество.
6. Лучшую модель сохранили на диск.



PyTorch является продуктом, финансируемым компанией Meta. Компания Meta Platforms Inc., по решению Тверского районного суда города Москвы от 21.03.2022, признана экстремистской организацией, её деятельность на территории России запрещена.