



**Big models**

**Daniil Merkulov**

**Optimization methods. MIPT**

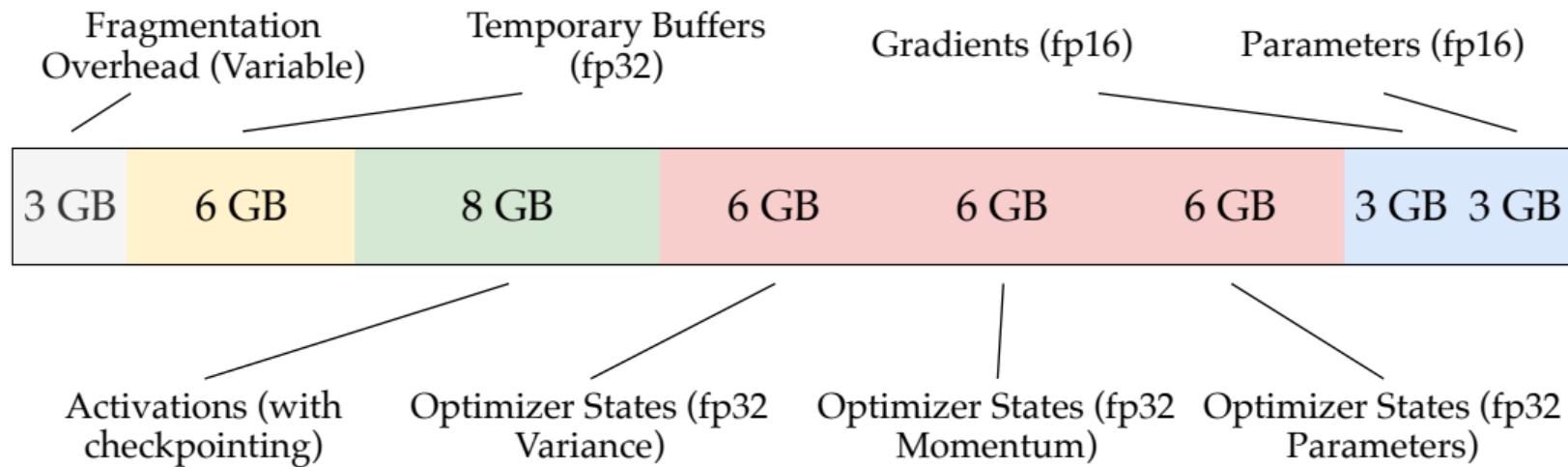
## Обучение больших моделей

## Потребление памяти при обучении GPT-2<sup>1</sup>



- Размер модели 1.5 В. Веса модели в fp16 занимают всего 3 GB, однако, для наивного обучения не хватит GPU даже на 32 GB

## Потребление памяти при обучении GPT-2<sup>1</sup>



- Размер модели 1.5 В. Веса модели в fp16 занимают всего 3 GB, однако, для наивного обучения не хватит GPU даже на 32 GB
- Для использования Adam в режиме mixed precision необходимо хранить 3 (!) копии модели в fp32.

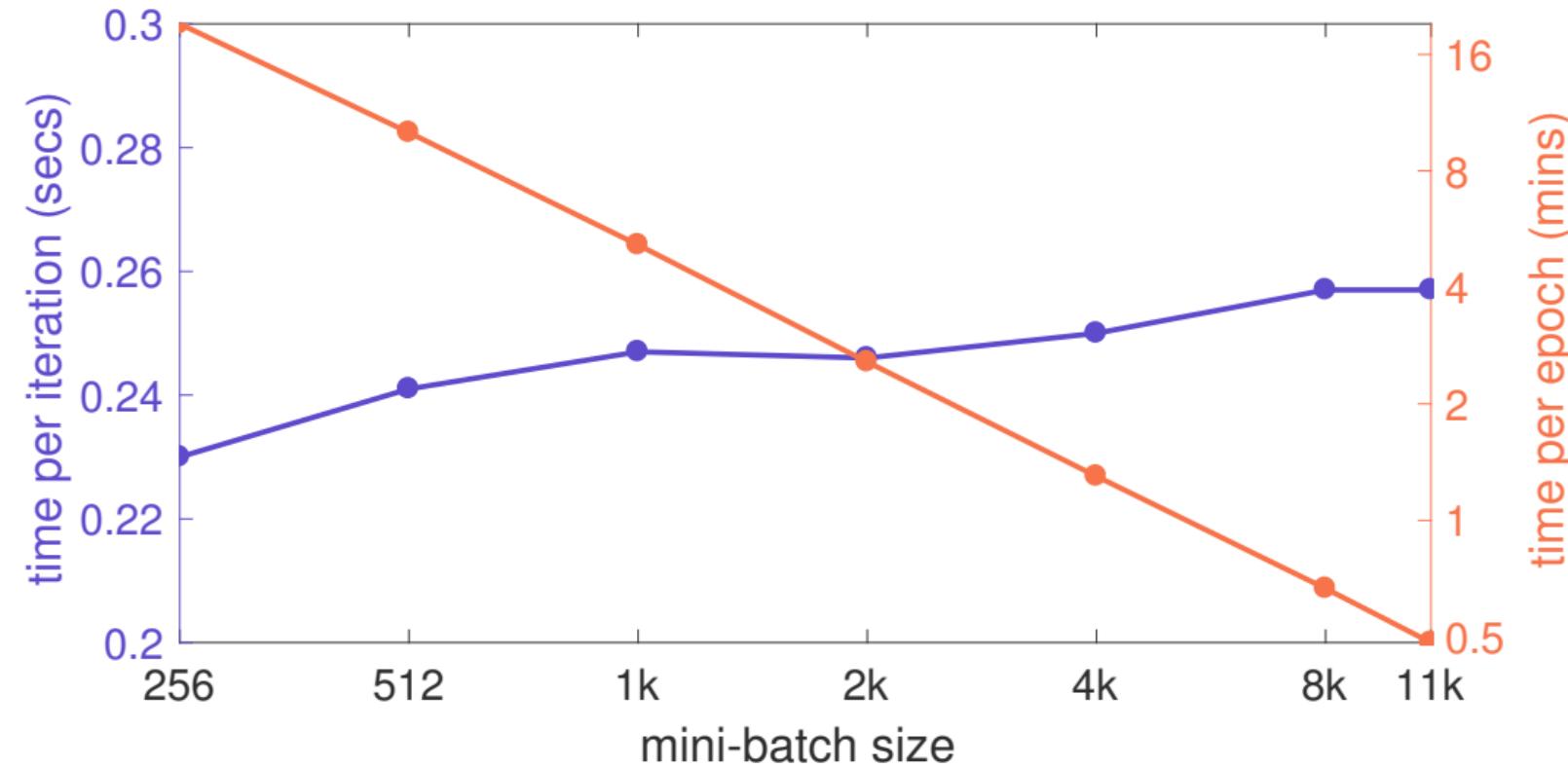
## Потребление памяти при обучении GPT-2<sup>1</sup>



- Размер модели 1.5 ГБ. Веса модели в fp16 занимают всего 3 GB, однако, для наивного обучения не хватит GPU даже на 32 GB
- Для использования Adam в режиме mixed precision необходимо хранить 3 (!) копии модели в fp32.
- Активации в наивном режиме могут занимать гораздо больше памяти: для длины последовательности 1K и размера батча 32 нужно 60 GB для хранения всех промежуточных активаций. Чекпоинтинг активаций позволяет сократить потребление до 8 GB за счёт их перевычисления (33% computational overhead)

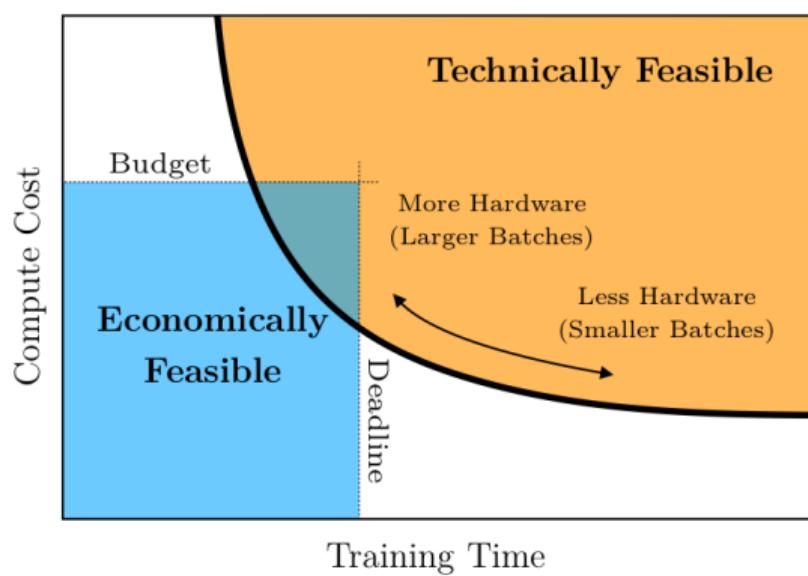
<sup>1</sup>ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

## Large batch training<sup>2</sup>

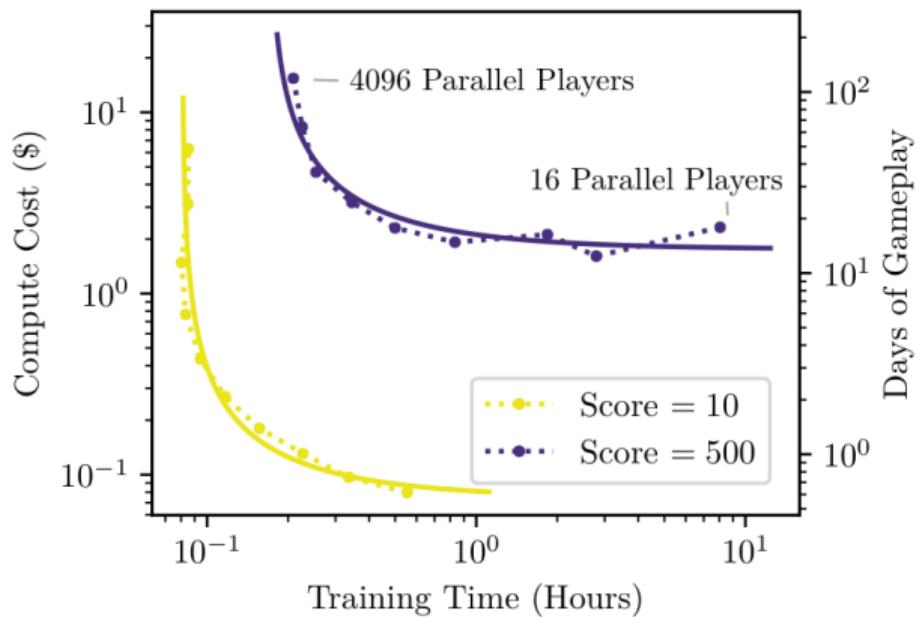


<sup>2</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

## Large batch training<sup>3</sup>



Atari Breakout - Pareto Fronts



<sup>3</sup>An Empirical Model of Large-Batch Training

## Large batch training <sup>4</sup>



<sup>4</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

## Linear and square root scaling rules

When training with large batches, the learning rate must be adjusted to maintain convergence speed and stability. The **linear scaling rule**<sup>5</sup> suggests multiplying the learning rate by the same factor as the increase in batch size:

$$\alpha_{\text{new}} = \alpha_{\text{base}} \cdot \frac{\text{Batch Size}_{\text{new}}}{\text{Batch Size}_{\text{base}}}$$

The **square root scaling rule**<sup>6</sup> proposes scaling the learning rate with the square root of the batch size increase:

$$\alpha_{\text{new}} = \alpha_{\text{base}} \cdot \sqrt{\frac{\text{Batch Size}_{\text{new}}}{\text{Batch Size}_{\text{base}}}}$$

Authors claimed, that it suits for adaptive optimizers like Adam, RMSProp and etc. while linear scaling rule serves well for SGD.

<sup>5</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

<sup>6</sup>Learning Rates as a Function of Batch Size: A Random Matrix Theory Approach to Neural Network Training

## Gradual warmup<sup>7</sup>

Gradual warmup helps to avoid instability when starting with large learning rates by slowly increasing the learning rate from a small value to the target value over a few epochs. This is defined as:

$$\alpha_t = \alpha_{\max} \cdot \frac{t}{T_w}$$

where  $t$  is the current iteration and  $T_w$  is the warmup duration in iterations. In the original paper, authors used first 5 epochs for gradual warmup.



Рис. 1: no warmup



Рис. 2: constant warmup

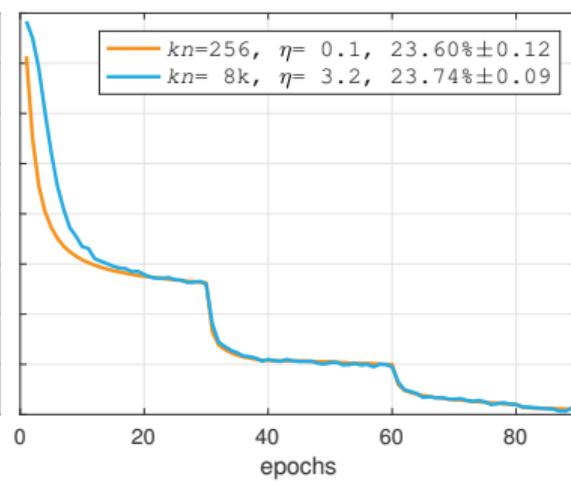
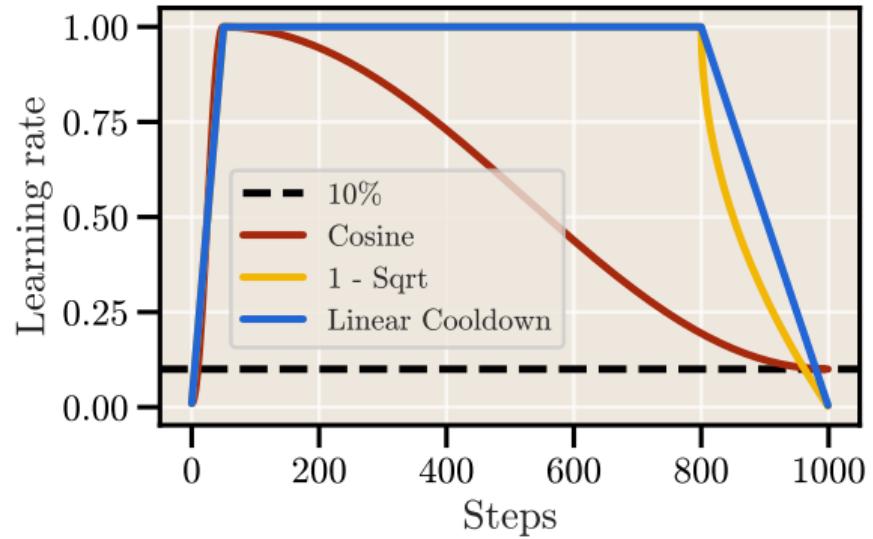


Рис. 3: gradual warmup

<sup>7</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

## Cooldown<sup>8 9</sup>



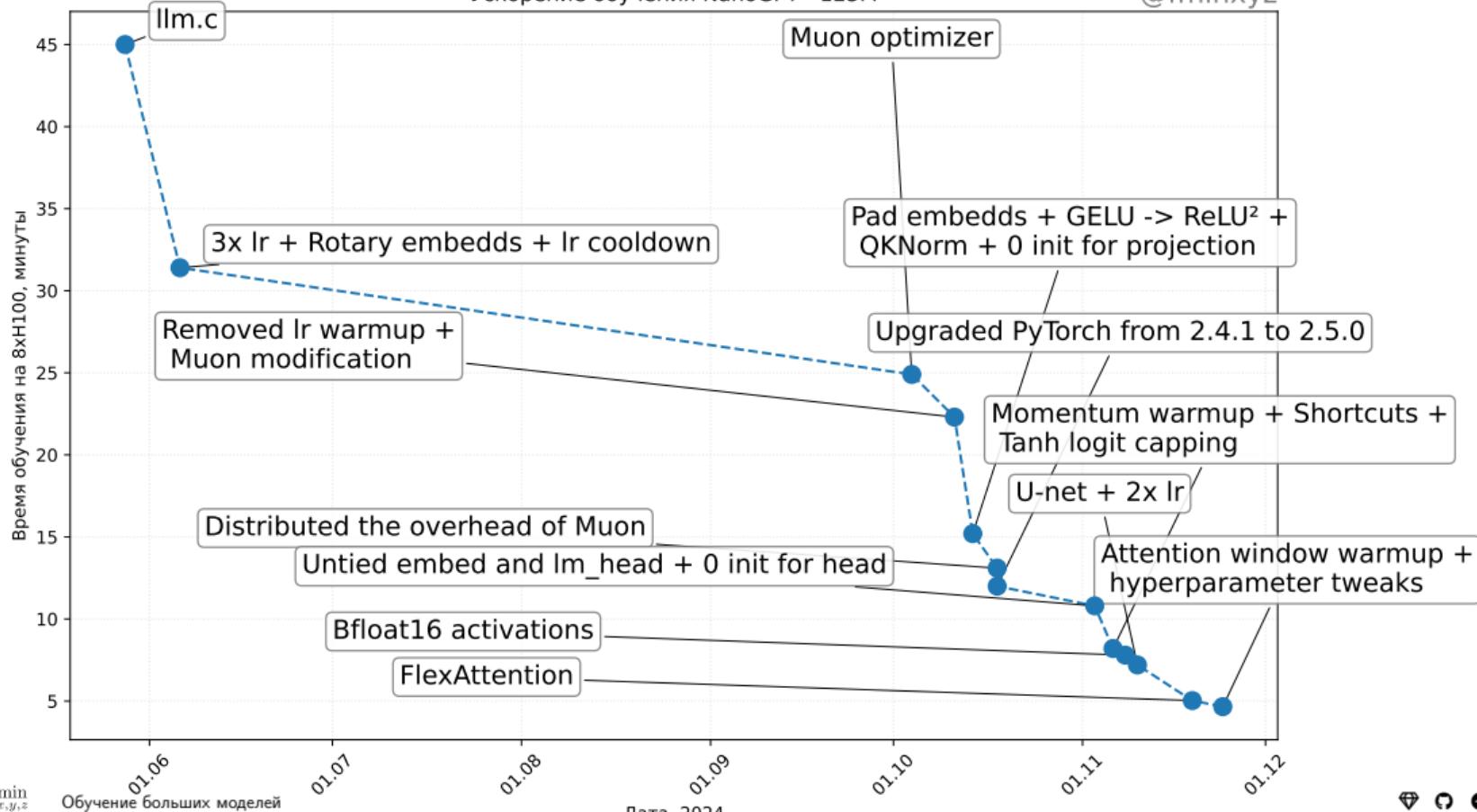
<sup>8</sup>Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations

<sup>9</sup>Scaling Vision Transformers

# NanoGPT speedrun

Ускорение обучения NanoGPT - 125M

@fminxyz



## Работают ли трюки, если увеличить размер модели?

Scaling up the NanoGPT (124M) speedrun

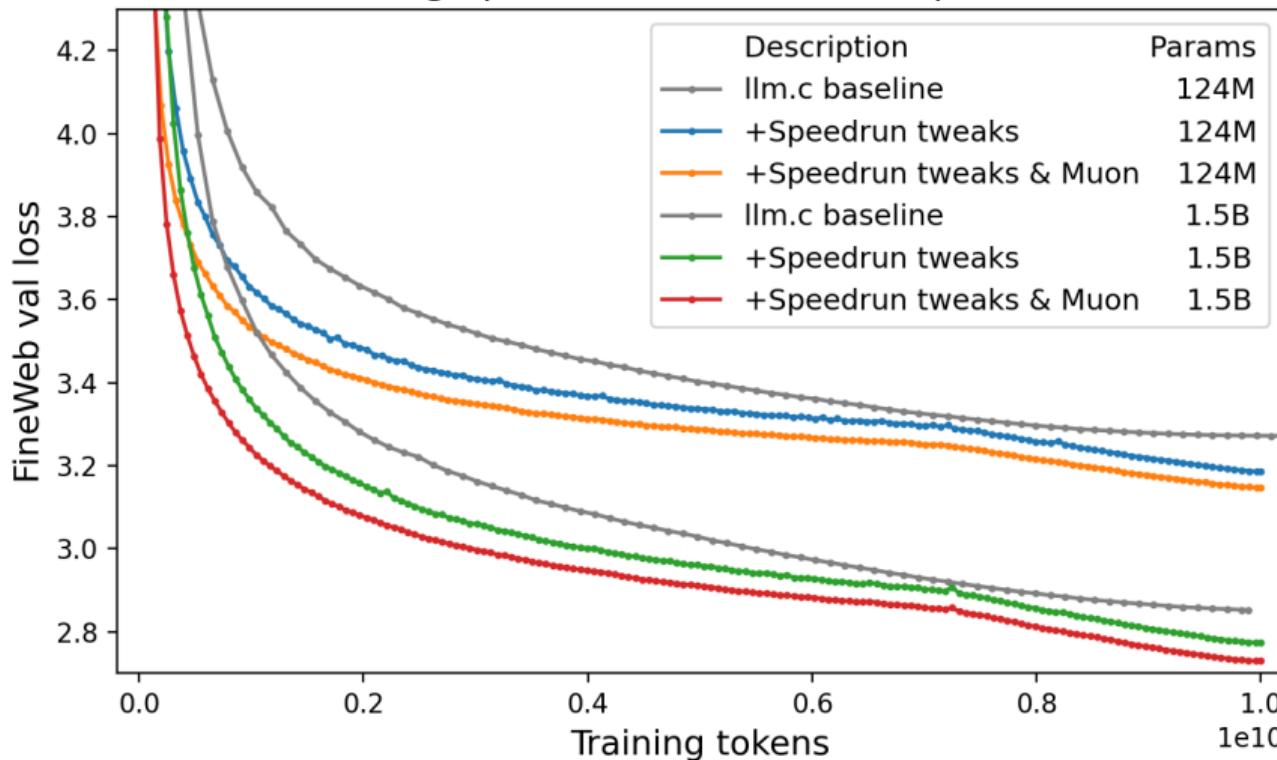


Рис. 5: Источник

## Работают ли трюки, если увеличить размер модели?



Рис. 6: Источник

Градиентный спуск сходится к локальному минимуму



# Градиентный спуск сходится к локальному минимуму



Стохастический градиентный спуск  
выпрыгивает из локальных минимумов



Эта задача оптимизации даже сложнее, чем кажется

## Impact of initialization<sup>10</sup>

💡 Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a “good” solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?

## Impact of initialization<sup>10</sup>

💡 Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a “good” solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?
- Random: Initialize randomly, e.g., via the Gaussian  $N(0, \sigma^2)$ , where std  $\sigma$  depends on the number of neurons in a given layer. *Symmetry breaking*.

## Impact of initialization<sup>10</sup>

💡 Properly initializing a NN important. NN loss is highly nonconvex; optimizing it to attain a “good” solution hard, requires careful tuning.

- Don't initialize all weights to be the same — why?
- Random: Initialize randomly, e.g., via the Gaussian  $N(0, \sigma^2)$ , where std  $\sigma$  depends on the number of neurons in a given layer. *Symmetry breaking*.
- One can find more useful advices here

<sup>10</sup>On the importance of initialization and momentum in deep learning Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton

# Влияние инициализации весов нейронной сети на сходимость методов<sup>11</sup>

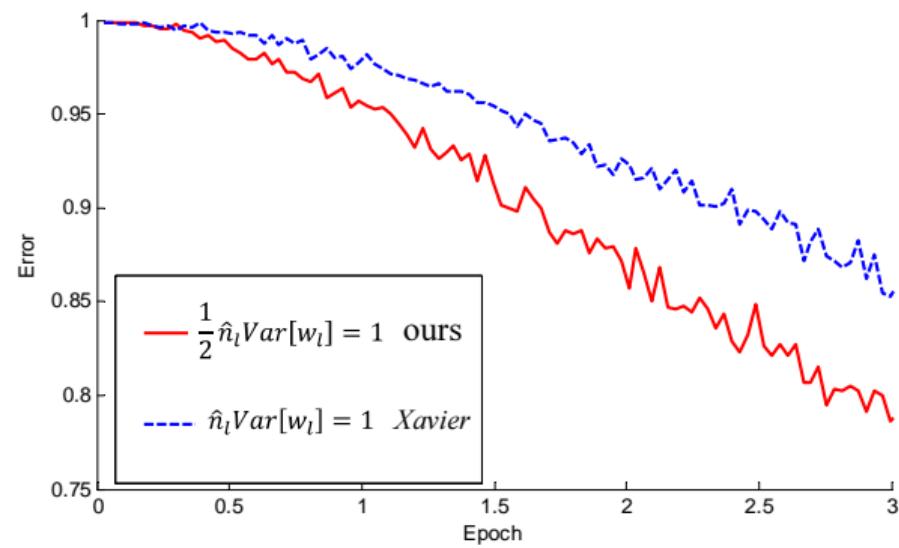


Рис. 7: 22-layer ReLU net: good init converges faster

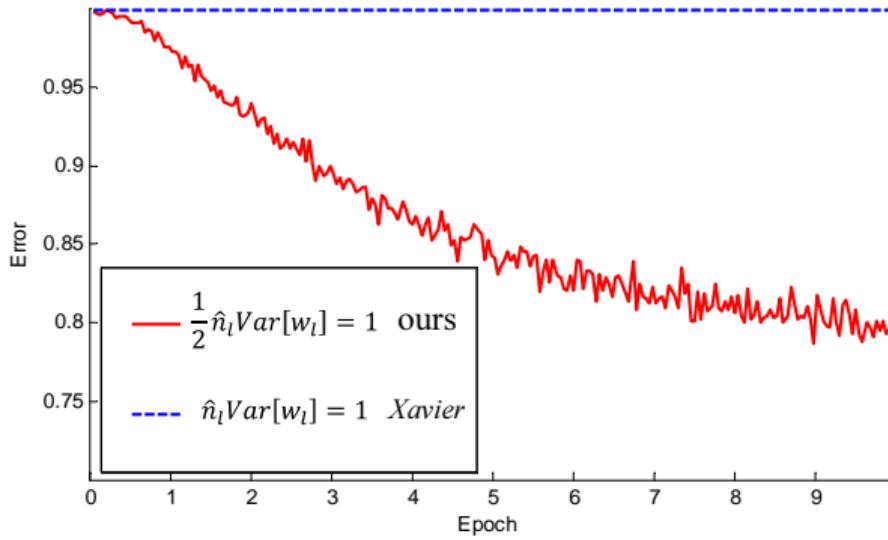


Рис. 8: 30-layer ReLU net: good init is able to converge

<sup>11</sup>Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>

- SVRG / SAG дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.

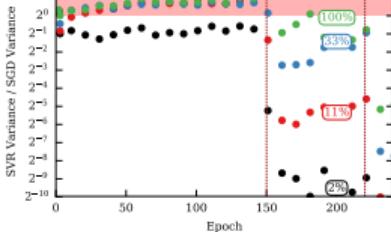


Рис. 9: DenseNet

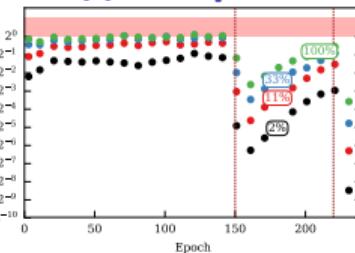


Рис. 10: Small ResNet



Рис. 11: LeNet-5

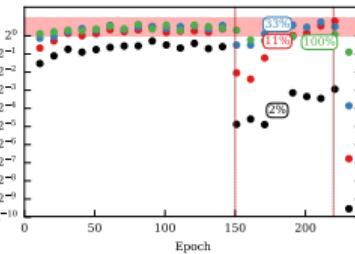


Рис. 12: ResNet-110

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>



Рис. 9: DenseNet

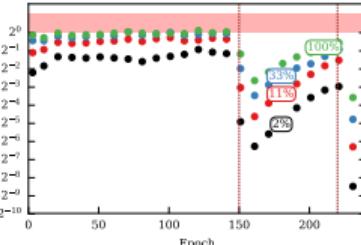


Рис. 10: Small ResNet

- SVRG / SAG дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.
- Измеренное отношение «дисперсия SGD / дисперсия SVRG» остаётся  $\lesssim 2$  для большинства слоёв - то есть реальное снижение шума минимально.



Рис. 11: LeNet-5

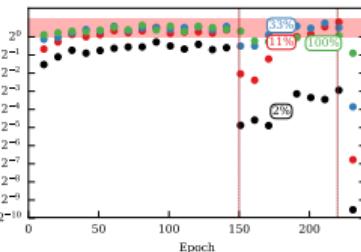


Рис. 12: ResNet-110

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>

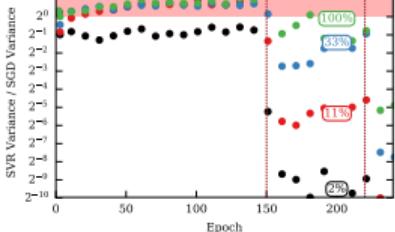


Рис. 9: DenseNet

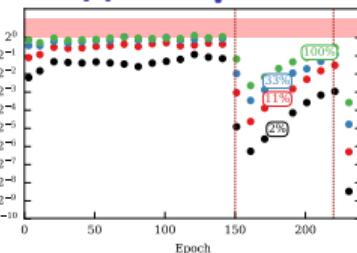


Рис. 10: Small ResNet

- SVRG / SAG дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.
- Измеренное отношение «дисперсия SGD / дисперсия SVRG» остаётся  $\lesssim 2$  для большинства слоёв - то есть реальное снижение шума минимально.
- Возможные причины:

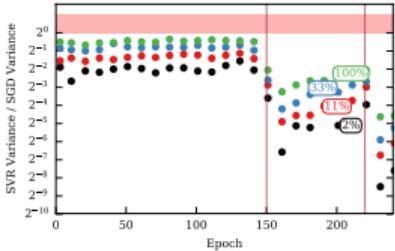


Рис. 11: LeNet-5

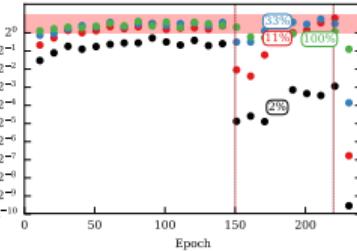


Рис. 12: ResNet-110

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>

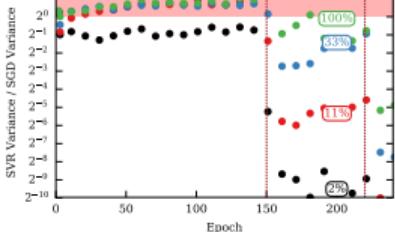


Рис. 9: DenseNet

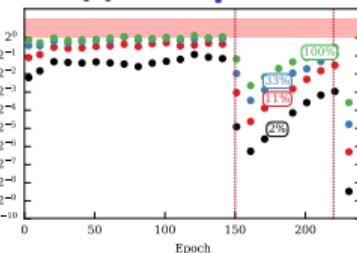


Рис. 10: Small ResNet

- SVRG / SAG дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.
- Измеренное отношение «дисперсия SGD / дисперсия SVRG» остаётся  $\lesssim 2$  для большинства слоёв - то есть реальное снижение шума минимально.
- Возможные причины:
  - Аугментация данных делает опорный градиент  $g_{\text{ref}}$  устаревшим уже после пары minibatch-ей.

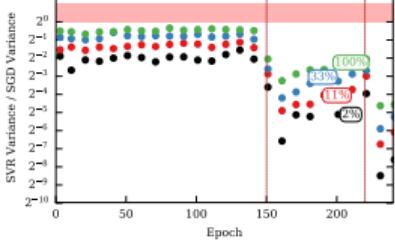


Рис. 11: LeNet-5

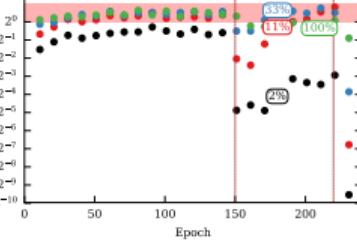


Рис. 12: ResNet-110

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>

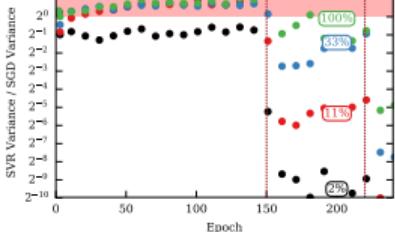


Рис. 9: DenseNet

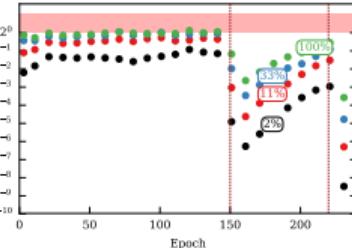


Рис. 10: Small ResNet

- **SVRG / SAG** дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.
- Измеренное отношение «дисперсия SGD / дисперсия SVRG» остаётся  $\lesssim 2$  для большинства слоёв - то есть реальное снижение шума минимально.
- Возможные причины:
  - **Аугментация данных** делает опорный градиент  $g_{\text{ref}}$  устаревшим уже после пары minibatch-ей.
  - **BatchNorm** и **Dropout** добавляют внутреннюю стохастичность, которую невозможно компенсировать прошлым  $g_{\text{ref}}$ .

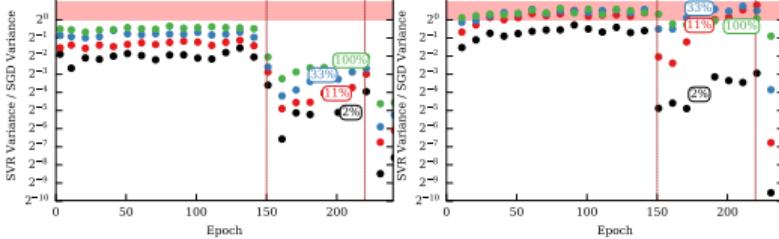


Рис. 11: LeNet-5

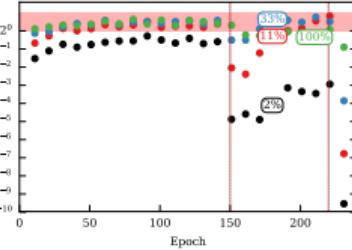


Рис. 12: ResNet-110

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>



Рис. 9: DenseNet

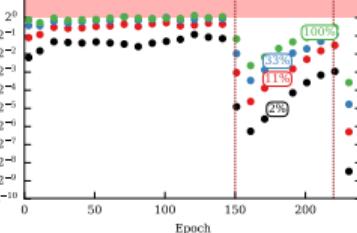


Рис. 10: Small ResNet

- **SVRG / SAG** дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.
- Измеренное отношение «дисперсия SGD / дисперсия SVRG» остаётся  $\lesssim 2$  для большинства слоёв - то есть реальное снижение шума минимально.
- Возможные причины:
  - **Аугментация данных** делает опорный градиент  $g_{\text{ref}}$  устаревшим уже после пары minibatch-ей.
  - **BatchNorm** и **Dropout** добавляют внутреннюю стохастичность, которую невозможно компенсировать прошлым  $g_{\text{ref}}$ .
  - Дополнительный полный проход по датасету (для подсчёта  $g_{\text{ref}}$ ) съедает потенциальную экономию итераций.

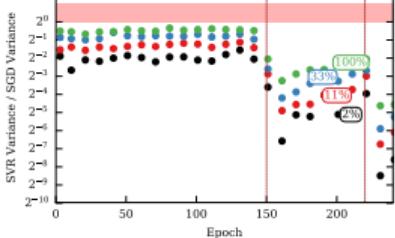


Рис. 11: LeNet-5

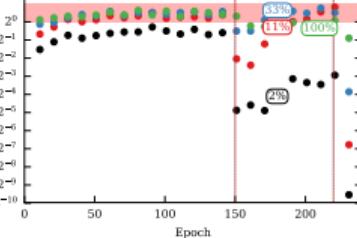


Рис. 12: ResNet-110

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>

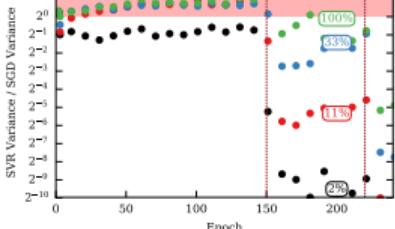


Рис. 9: DenseNet

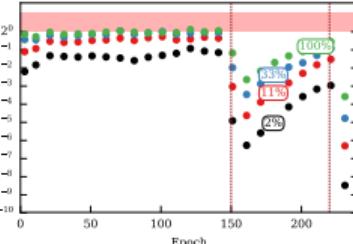


Рис. 10: Small ResNet

- **SVRG / SAG** дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.
- Измеренное отношение «дисперсия SGD / дисперсия SVRG» остаётся  $\lesssim 2$  для большинства слоёв - то есть реальное снижение шума минимально.
- Возможные причины:
  - **Аугментация данных** делает опорный градиент  $g_{\text{ref}}$  устаревшим уже после пары minibatch-ей.
  - **BatchNorm** и **Dropout** добавляют внутреннюю стохастичность, которую невозможно компенсировать прошлым  $g_{\text{ref}}$ .
  - Дополнительный полный проход по датасету (для подсчёта  $g_{\text{ref}}$ ) съедает потенциальную экономию итераций.
- «Стриминговые» модификации SVRG, рассчитанные на аугментацию, снижают теоретическое смещение, но также проигрывают SGD по времени и качеству.

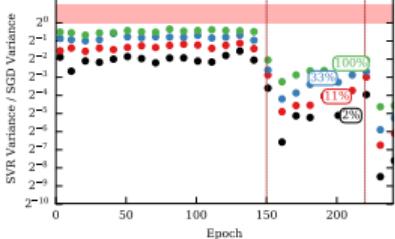


Рис. 11: LeNet-5

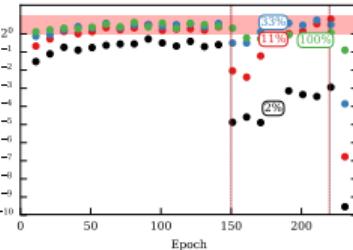


Рис. 12: ResNet-110

# Методы уменьшения дисперсии: почему не работают на глубоких сетях? <sup>12</sup>

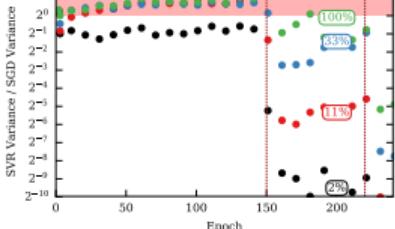


Рис. 9: DenseNet

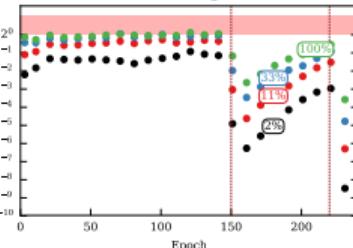


Рис. 10: Small ResNet



Рис. 11: LeNet-5

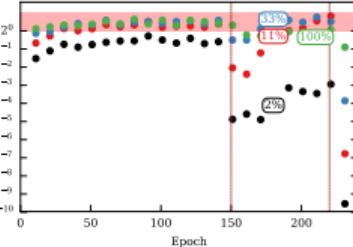


Рис. 12: ResNet-110

- **SVRG / SAG** дают убедительные выигрыши в выпуклых задачах, но на CIFAR-10 (LeNet-5) и ImageNet (ResNet-18) не опережают обычный SGD.
- Измеренное отношение «дисперсия SGD / дисперсия SVRG» остаётся  $\lesssim 2$  для большинства слоёв - то есть реальное снижение шума минимально.
- Возможные причины:
  - **Аугментация данных** делает опорный градиент  $g_{ref}$  устаревшим уже после пары minibatch-ей.
  - **BatchNorm** и **Dropout** добавляют внутреннюю стохастичность, которую невозможно компенсировать прошлым  $g_{ref}$ .
  - Дополнительный полный проход по датасету (для подсчёта  $g_{ref}$ ) съедает потенциальную экономию итераций.
- «Стриминговые» модификации SVRG, рассчитанные на аугментацию, снижают теоретическое смещение, но также проигрывают SGD по времени и качеству.
- **Вывод:** существующие методы уменьшения дисперсии непрактичны для современных глубоких сетей; будущие решения должны учитывать стохастичность архитектуры и данных (аугментация, BatchNorm, Dropout).

$f \xrightarrow{12}$  On the Ineffectiveness of Variance Reduced Optimization for Deep Learning  
Эта задача оптимизации даже сложнее, чем кажется

## Adam работает хуже для CV, чем для LLM? <sup>13</sup>



Рис. 13: CNNs on MNIST and CIFAR10

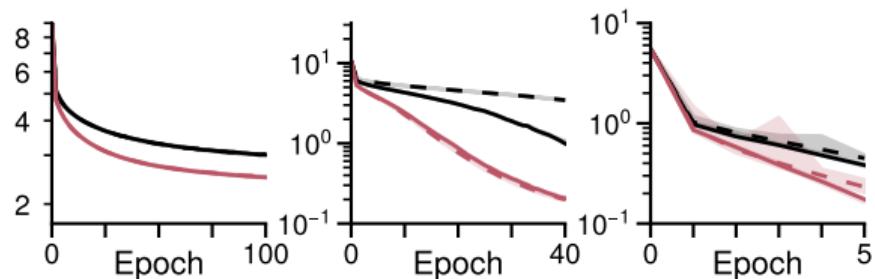


Рис. 14: Transformers on PTB, WikiText2, and SQuAD

Черные линии - SGD; красные линии - Adam.

<sup>13</sup>Linear attention is (maybe) all you need (to understand transformer optimization)

# Почему Adam работает хуже для CV, чем для LLM? <sup>14</sup>

Потому что шум градиентов в языковых моделях имеет тяжелые хвосты?



<sup>14</sup>Linear attention is (maybe) all you need (to understand transformer optimization)

## Почему Adam работает хуже для CV, чем для LLM? <sup>15</sup>

Нет! Метки имеют тяжелые хвосты!

В компьютерном зрении датасеты часто сбалансированы: 1000 котиков, 1000 песелей и т.д.  
В языковых датасетах почти всегда не так: слово *the* встречается часто, слово *tie* на порядки реже

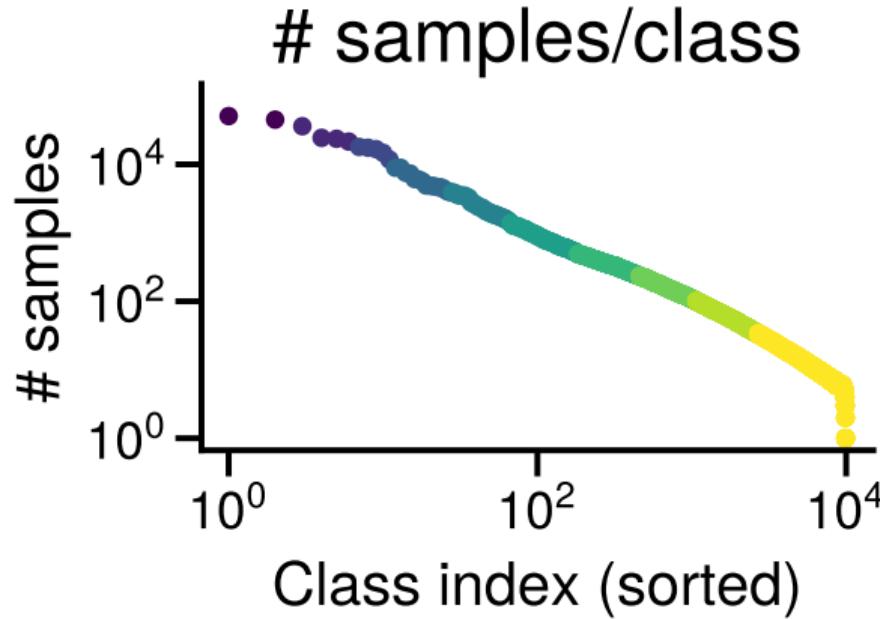


Рис. 15: Распределение частоты токенов в PTB

<sup>15</sup>Heavy-Tailed Class Imbalance and Why Adam Outperforms Gradient Descent on Language Models

# Почему Adam работает хуже для CV, чем для LLM? <sup>16</sup>

SGD медленно прогрессирует на редких классах



SGD не добивается прогресса на низкочастотных классах, в то время как Adam добивается. Обучение GPT-2 S на WikiText-103. (a) Распределение классов, отсортированных по частоте встречаемости, разбитых на группы, соответствующие  $\approx 10\%$  данных. (b) Значение функции потерь при обучении. (c, d) Значение функции потерь при обучении для каждой группы при использовании SGD и Adam.

<sup>16</sup>Heavy-Tailed Class Imbalance and Why Adam Outperforms Gradient Descent on Language Models

## Визуализация с помощью проекции на прямую

- Обозначим начальную точку как  $w_0$ , представляющую собой веса нейронной сети при инициализации. Веса, полученные после обучения, обозначим как  $\hat{w}$ .

$$L(\alpha) = L(w_0 + \alpha w_1), \text{ where } \alpha \in [-b, b].$$

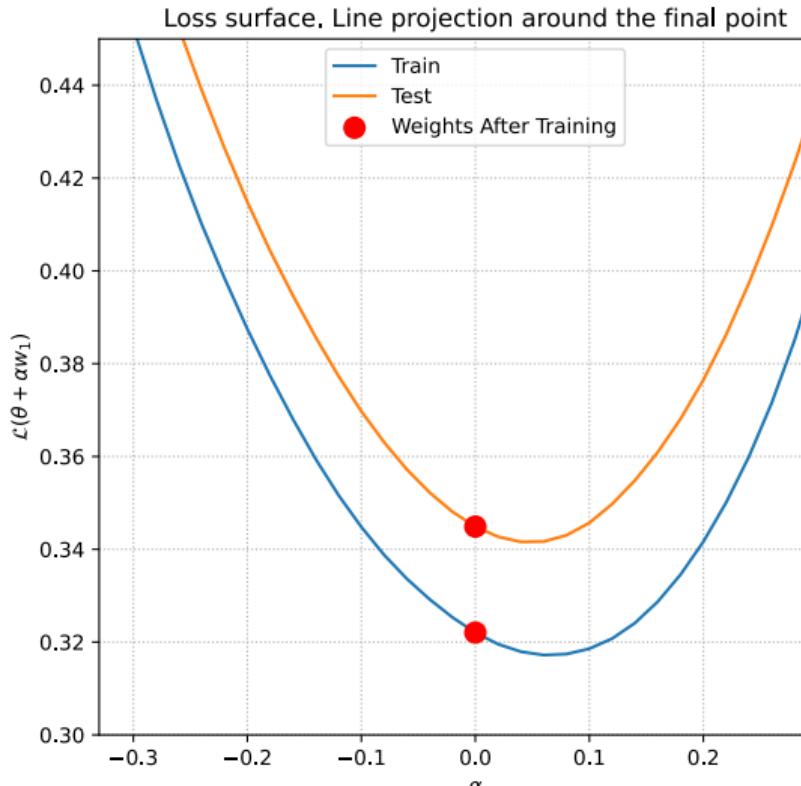
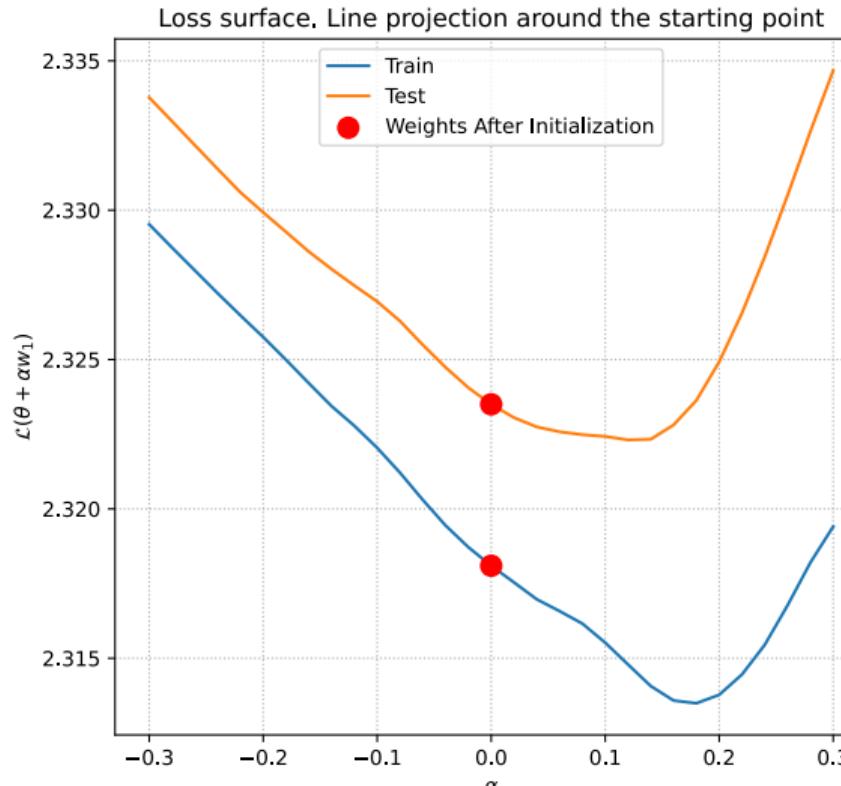
## Визуализация с помощью проекции на прямую

- Обозначим начальную точку как  $w_0$ , представляющую собой веса нейронной сети при инициализации. Веса, полученные после обучения, обозначим как  $\hat{w}$ .
- Генерируем случайный вектор такой же размерности и нормы  $w_1 \in \mathbb{R}^p$ , затем вычисляем значение функции потерь вдоль этого вектора:

$$L(\alpha) = L(w_0 + \alpha w_1), \text{ where } \alpha \in [-b, b].$$

# Проекция функции потерь нейронной сети на прямую

No Dropout



# Проекция функции потерь нейронной сети на прямую

Dropout 0.2

Loss surface, Line projection around the starting point



Loss surface, Line projection around the final point



## Проекция функции потерь нейронной сети на плоскость

- Мы можем расширить эту идею и построить проекцию поверхности потерь на плоскость, которая задается 2 случайными векторами.

$L(\alpha, \beta) = L(w_0 + \alpha w_1 + \beta w_2)$ , where  $\alpha, \beta \in [-b, b]^2$ .

### No Dropout. Plane projection of loss surface.



## Проекция функции потерь нейронной сети на плоскость

- Мы можем расширить эту идею и построить проекцию поверхности потерь на плоскость, которая задается 2 случайными векторами.
- Два случайных гауссовых вектора в пространстве большой размерности с высокой вероятностью ортогональны.

$$L(\alpha, \beta) = L(w_0 + \alpha w_1 + \beta w_2), \text{ where } \alpha, \beta \in [-b, b]^2.$$

No Dropout. Plane projection of loss surface.



Может ли быть полезно изучение таких проекций? <sup>17</sup>



Рис. 19: The loss surface of ResNet-56  
without skip connections



Рис. 20: The loss surface of ResNet-56 with skip connections

<sup>17</sup>Visualizing the Loss Landscape of Neural Nets, Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein

# Может ли быть полезно изучение таких проекций, если серьезно? <sup>18</sup>



Рис. 21: Examples of a loss landscape of a typical CNN model on FashionMNIST and CIFAR10 datasets found with MPO. Loss values are color-coded according to a logarithmic scale

<sup>18</sup>Loss Landscape Sightseeing with Multi-Point Optimization, Ivan Skorokhodov, Mikhail Burtsev  
 $f \rightarrow \min_{x,y,z}$  Эта задача оптимизации даже сложнее, чем кажется

## Ширина локальных минимумов

Узкие и широкие локальные минимумы



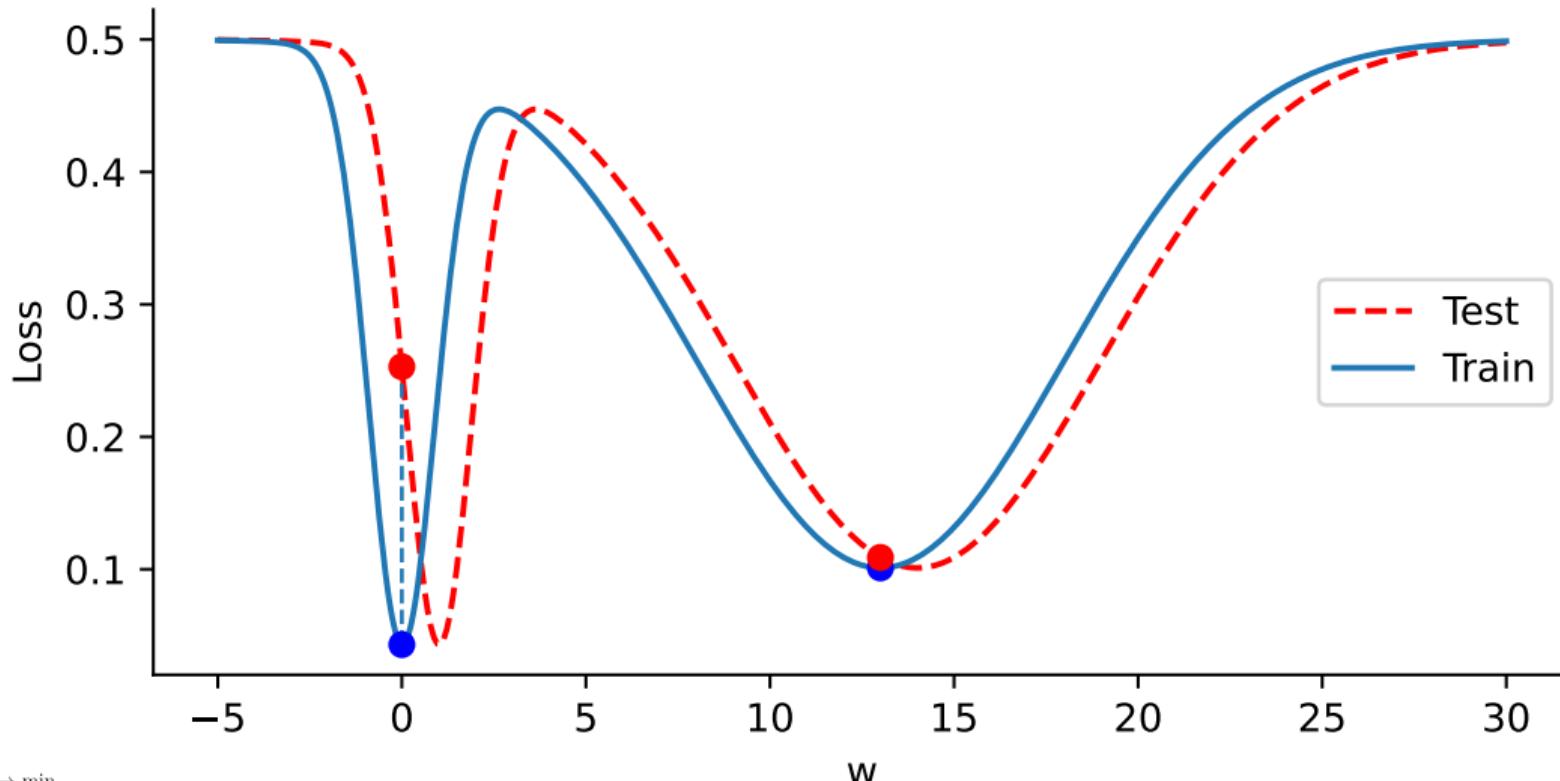
## Ширина локальных минимумов

Узкие и широкие локальные минимумы

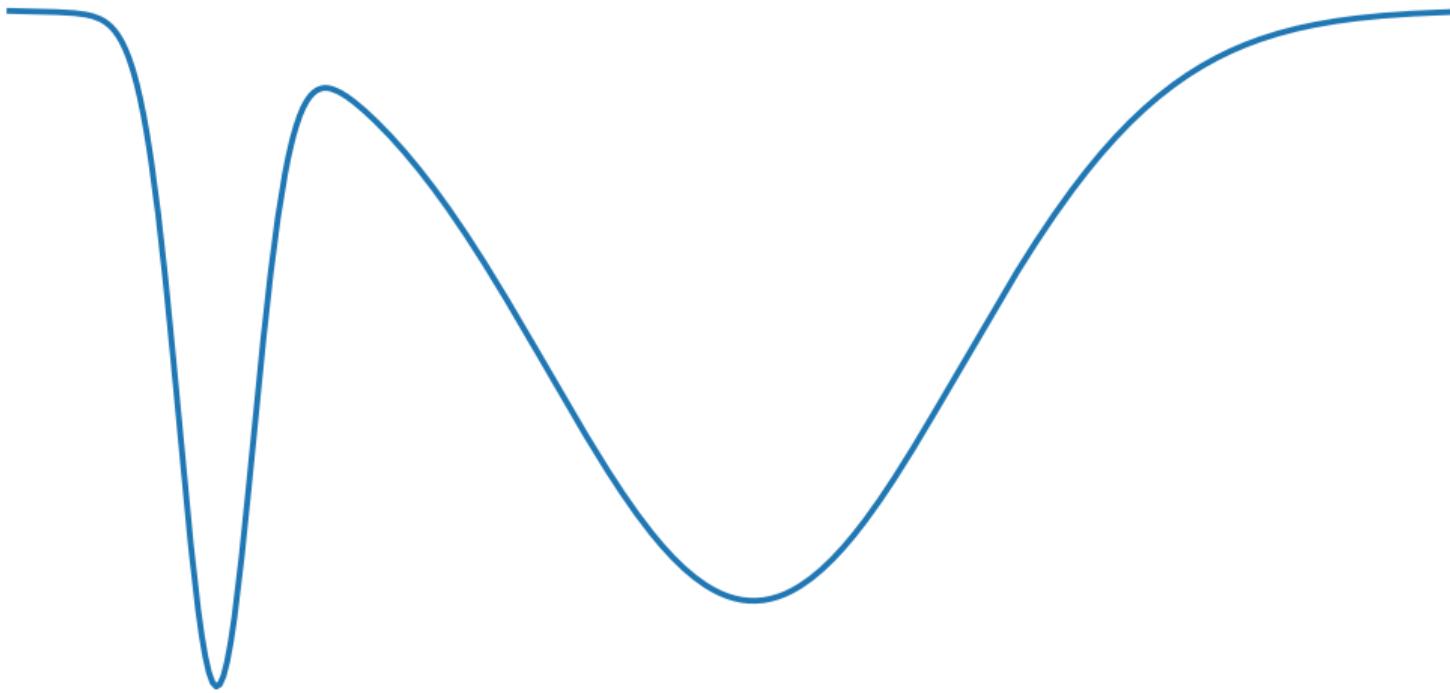


## Ширина локальных минимумов

### Узкие и широкие локальные минимумы



Градиентный спуск с маленьким шагом  
сходится в узкий локальный минимум



Градиентный спуск с большим шагом  
избегает узкого локального минимума



## Модели не сходятся к стационарным точкам, но это не страшно<sup>19</sup>



<sup>19</sup>NN Weights Do Not Converge to Stationary Points

Modular Division (training on 50% of data)



Рис. 22: Training transformer with 2 layers, width 128, and 4 attention heads, with a total of about  $4 \cdot 10^5$  non-embedding parameters. Reproduction of experiments (~ half an hour) is available [here](#)

- Рекомендую посмотреть лекцию Дмитрия Ветрова **Удивительные свойства функции потерь в нейронной сети** (Surprising properties of loss landscape in overparameterized models). видео, Презентация

Modular Division (training on 50% of data)



Рис. 22: Training transformer with 2 layers, width 128, and 4 attention heads, with a total of about  $4 \cdot 10^5$  non-embedding parameters. Reproduction of experiments (~ half an hour) is available here

- Рекомендую посмотреть лекцию Дмитрия Ветрова **Удивительные свойства функции потерь в нейронной сети** (Surprising properties of loss landscape in overparameterized models). видео, Презентация
- Автор канала Свидетели Градиента собирает интересные наблюдения и эксперименты про гроккинг.

# Grokking<sup>20</sup>

Modular Division (training on 50% of data)



Рис. 22: Training transformer with 2 layers, width 128, and 4 attention heads, with a total of about  $4 \cdot 10^5$  non-embedding parameters. Reproduction of experiments ( $\sim$  half an hour) is available here

- Рекомендую посмотреть лекцию Дмитрия Ветрова **Удивительные свойства функции потерь в нейронной сети** (Surprising properties of loss landscape in overparameterized models). видео, Презентация
- Автор канала Свидетели Градиента собирает интересные наблюдения и эксперименты про гроккинг.
- Также есть видео с его докладом **Чем не является гроккинг**.

<sup>20</sup>Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets, Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, Vedant Misra

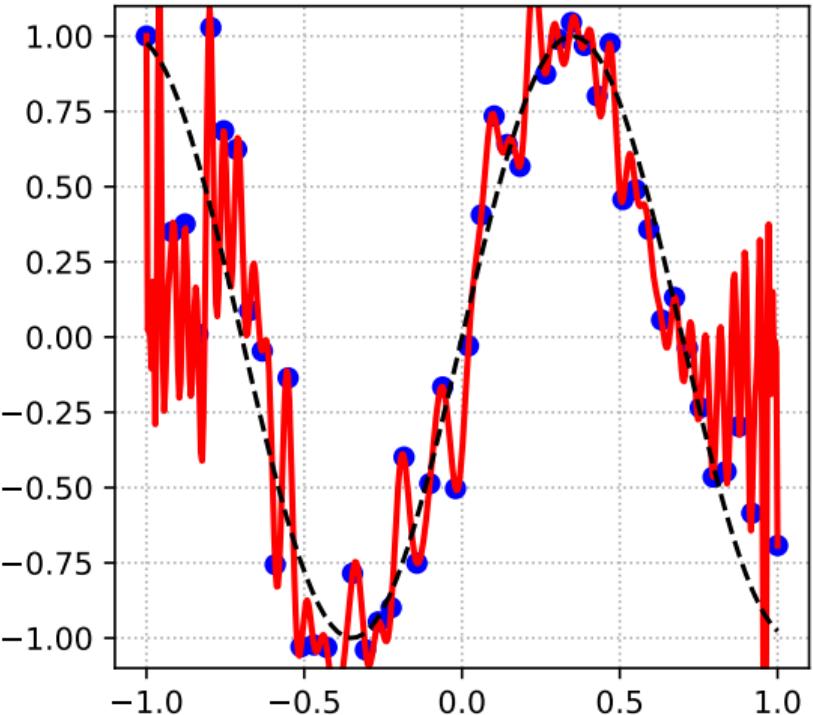
## Double Descent<sup>21</sup>



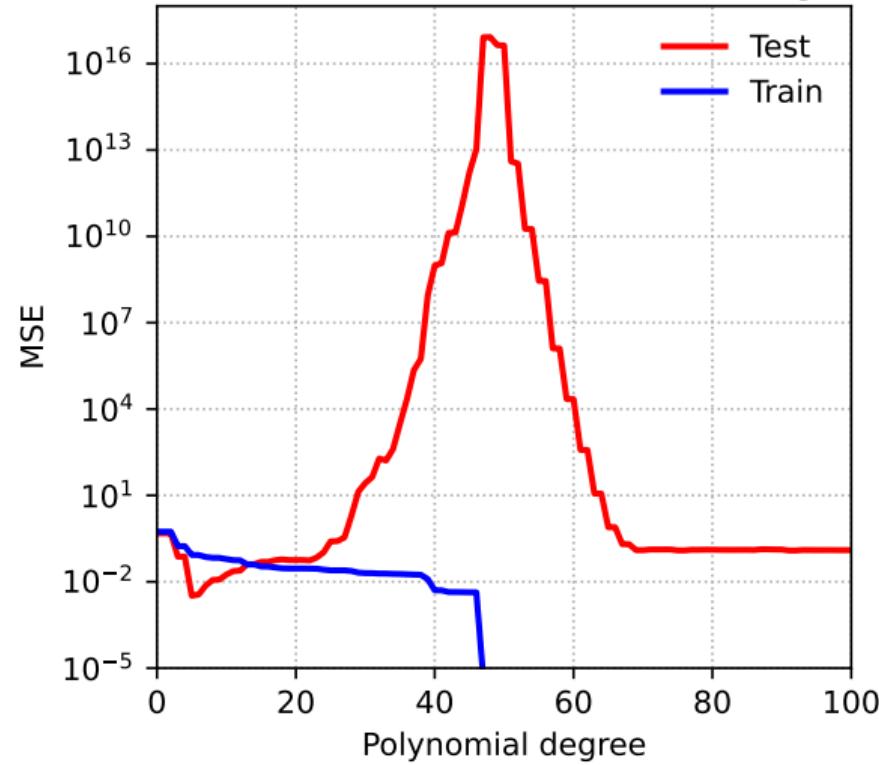
<sup>21</sup>Reconciling modern machine learning practice and the bias-variance trade-off, Mikhail Belkin, Daniel Hsu, Siyuan Ma, Soumik Mandal

## Double Descent

Polynomial Fitting



@fminxyz

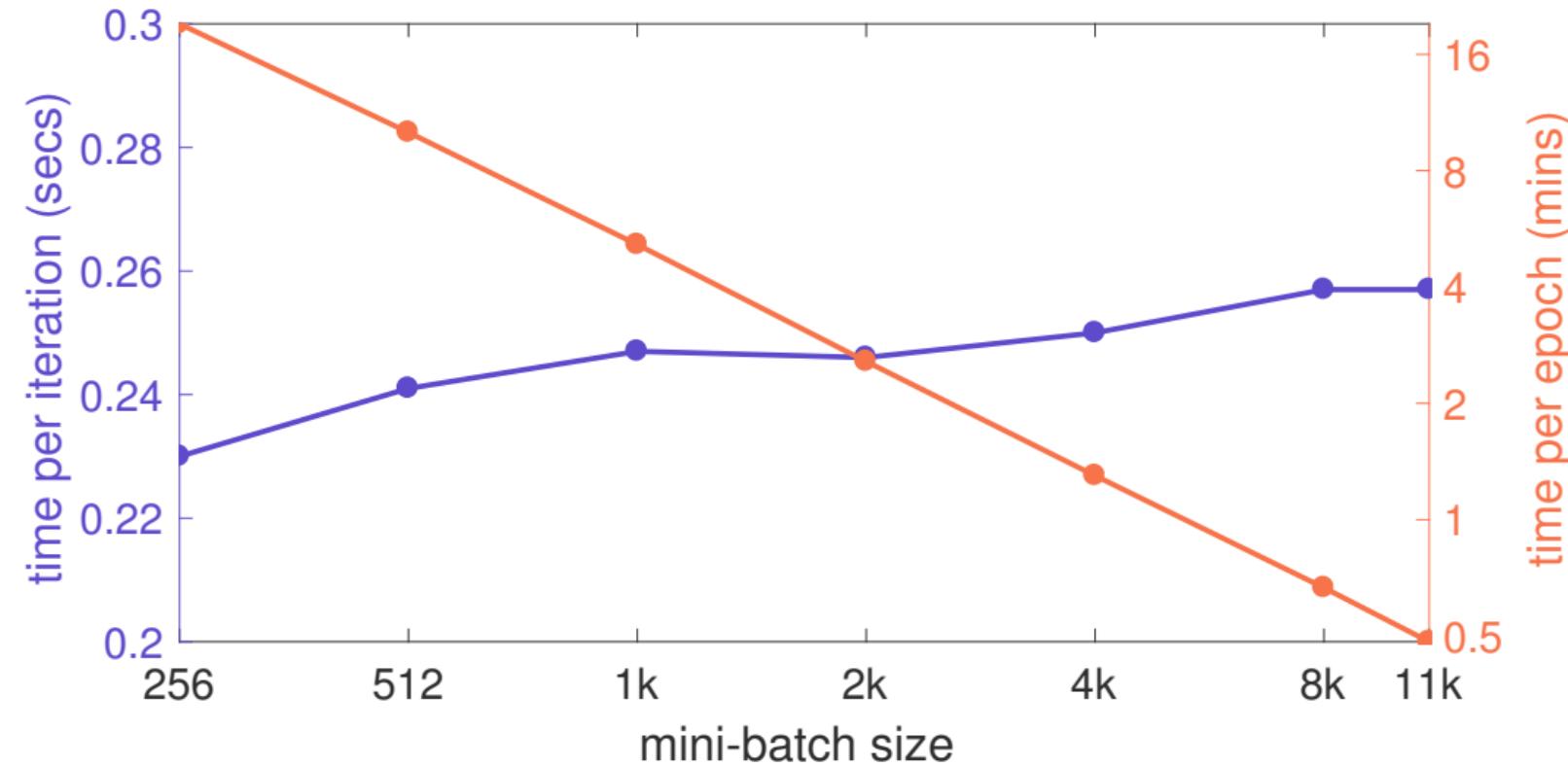


## Exponential learning rate

- Exponential Learning Rate Schedules for Deep Learning

## Large batch training

## Large batch training <sup>22</sup>



<sup>22</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

## Large batch training<sup>23</sup>



<sup>23</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

## Large batch training<sup>24</sup>

Effective batch size ( $kn$ )	$\alpha$	top-1 error (%)
256	0.05	$23.92 \pm 0.10$
256	0.10	$23.60 \pm 0.12$
256	0.20	$23.68 \pm 0.09$
8k	$0.05 \cdot 32$	$24.27 \pm 0.08$
8k	$0.10 \cdot 32$	$23.74 \pm 0.09$
8k	$0.20 \cdot 32$	$24.05 \pm 0.18$
8k	0.10	$41.67 \pm 0.10$
8k	$0.10 \cdot \sqrt{32}$	$26.22 \pm 0.03$

Comparison of learning rate scaling rules. ResNet-50 trained on ImageNet. A reference learning rate of  $\alpha = 0.1$  works best for  $kn = 256$  (23.68% error). The linear scaling rule suggests  $\alpha = 0.1 \cdot 32$  when  $kn = 8k$ , which again gives best performance (23.74% error). Other ways of scaling  $\alpha$  give worse results.

<sup>24</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

## Linear and square root scaling rules

When training with large batches, the learning rate must be adjusted to maintain convergence speed and stability. The **linear scaling rule**<sup>25</sup> suggests multiplying the learning rate by the same factor as the increase in batch size:

$$\alpha_{\text{new}} = \alpha_{\text{base}} \cdot \frac{\text{Batch Size}_{\text{new}}}{\text{Batch Size}_{\text{base}}}$$

The **square root scaling rule**<sup>26</sup> proposes scaling the learning rate with the square root of the batch size increase:

$$\alpha_{\text{new}} = \alpha_{\text{base}} \cdot \sqrt{\frac{\text{Batch Size}_{\text{new}}}{\text{Batch Size}_{\text{base}}}}$$

Authors claimed, that it suits for adaptive optimizers like Adam, RMSProp and etc. while linear scaling rule serves well for SGD.

<sup>25</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

<sup>26</sup>Learning Rates as a Function of Batch Size: A Random Matrix Theory Approach to Neural Network Training

## Gradual warmup <sup>27</sup>

Gradual warmup helps to avoid instability when starting with large learning rates by slowly increasing the learning rate from a small value to the target value over a few epochs. This is defined as:

$$\alpha_t = \alpha_{\max} \cdot \frac{t}{T_w}$$

where  $t$  is the current iteration and  $T_w$  is the warmup duration in iterations. In the original paper, authors used first 5 epochs for gradual warmup.



Рис. 23: no warmup



Рис. 24: constant warmup



Рис. 25: gradual warmup

<sup>27</sup>Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour  
 $f \rightarrow \min_{x,y,z}$  Large batch training

## Gradient accumulation

Gradient accumulation allows the effective batch size to be increased without requiring larger memory by accumulating gradients over several mini-batches:

### Without gradient accumulation

```
for i, (inputs, targets) in enumerate(data):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

## Gradient accumulation

Gradient accumulation allows the effective batch size to be increased without requiring larger memory by accumulating gradients over several mini-batches:

Without gradient accumulation

```
for i, (inputs, targets) in enumerate(data):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

With gradient accumulation

```
for i, (inputs, targets) in enumerate(data):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()

    if (i+1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

## MultiGPU training

## Data Parallel training

1. Parameter server sends the full copy of the model to each device

## Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes

## Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients

## Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients
4. Parameter server updates the model

## Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients
4. Parameter server updates the model

## Data Parallel training

1. Parameter server sends the full copy of the model to each device
2. Each device makes forward and backward passes
3. Parameter server gathers gradients
4. Parameter server updates the model

Per device batch size:  $b$ . Overall batchsize:  $Db$ . Data parallelism involves splitting the data across multiple GPUs, each with a copy of the model. Gradients are averaged and weights updated synchronously:



## Distributed Data Parallel training

Distributed Data Parallel (DDP)<sup>28</sup> extends data parallelism across multiple nodes. Each node computes gradients locally, then synchronizes with others. Below one can find differences from the PyTorch site. This is used by default in Accelerate library.

DataParallel	DistributedDataParallel
More overhead; model is replicated and destroyed at each forward pass Only supports single-node parallelism	Model is replicated only once Supports scaling to multiple machines
Slower; uses multithreading on a single process and runs into Global Interpreter Lock (GIL) contention	Faster (no GIL contention) because it uses multiprocessing

<sup>28</sup>Getting Started with Distributed Data Parallel

## Naive model parallelism

Model parallelism divides the model across multiple GPUs. Each GPU handles a subset of the model layers, reducing memory load per GPU. Allows to work with the models, that won't fit in the single GPU Poor resource utilization.



Рис. 27: Model parallelism

## Pipeline model parallelism (GPipe) 29

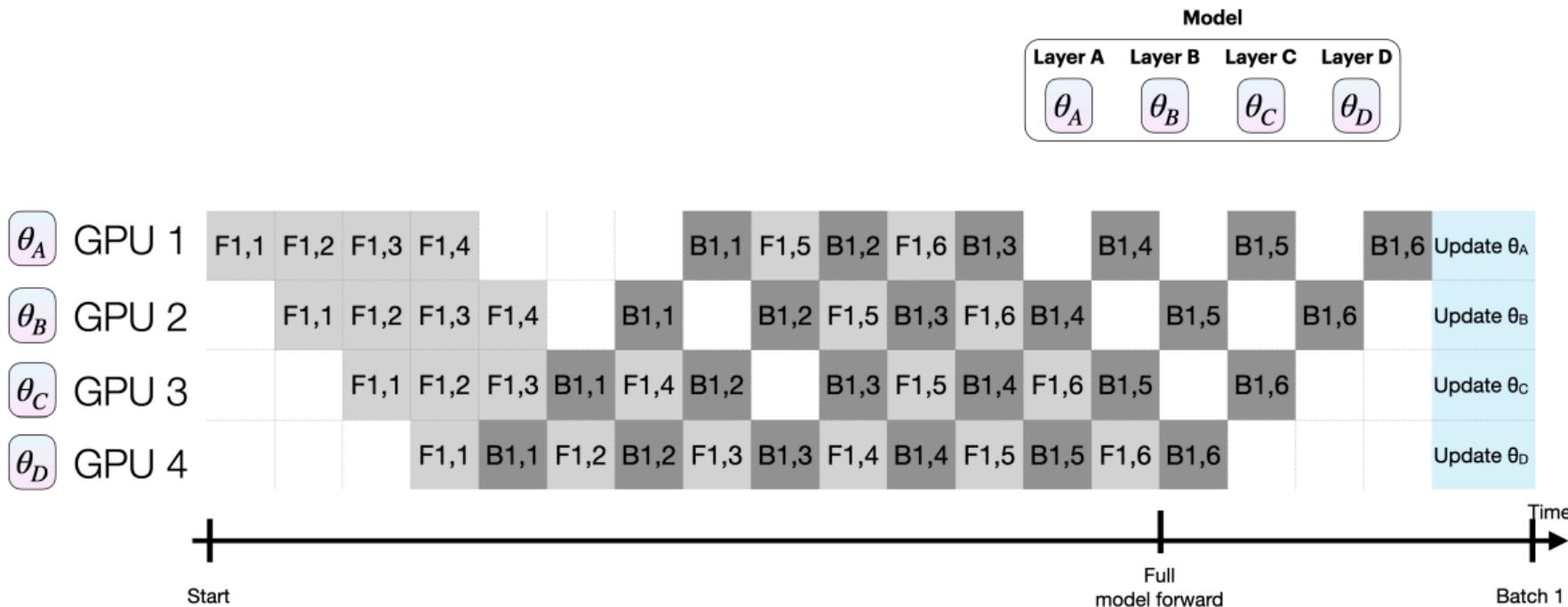
GPipe splits the model into stages, each processed sequentially. Micro-batches are passed through the pipeline, allowing for overlapping computation and communication:



<sup>29</sup>GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

## Pipeline model parallelism (PipeDream) 30

PipeDream uses asynchronous pipeline parallelism, balancing forward and backward passes across the pipeline stages to maximize utilization and reduce idle time:



<sup>30</sup>PipeDream: Generalized Pipeline Parallelism for DNN Training

<sup>31</sup>ZeRO: Memory Optimizations Toward Training Trillion Parameter Models



LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where  $\Delta W = AB^T$ , with  $A$  and  $B$  being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- $A$  is initialized as usual, while  $B$  is initialized with zeroes in order to start from identity mapping



LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where  $\Delta W = AB^T$ , with  $A$  and  $B$  being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- $A$  is initialized as usual, while  $B$  is initialized with zeroes in order to start from identity mapping
- $r$  is typically selected between 2 and 64



LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where  $\Delta W = AB^T$ , with  $A$  and  $B$  being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- $A$  is initialized as usual, while  $B$  is initialized with zeroes in order to start from identity mapping
- $r$  is typically selected between 2 and 64
- Usually applied to attention modules



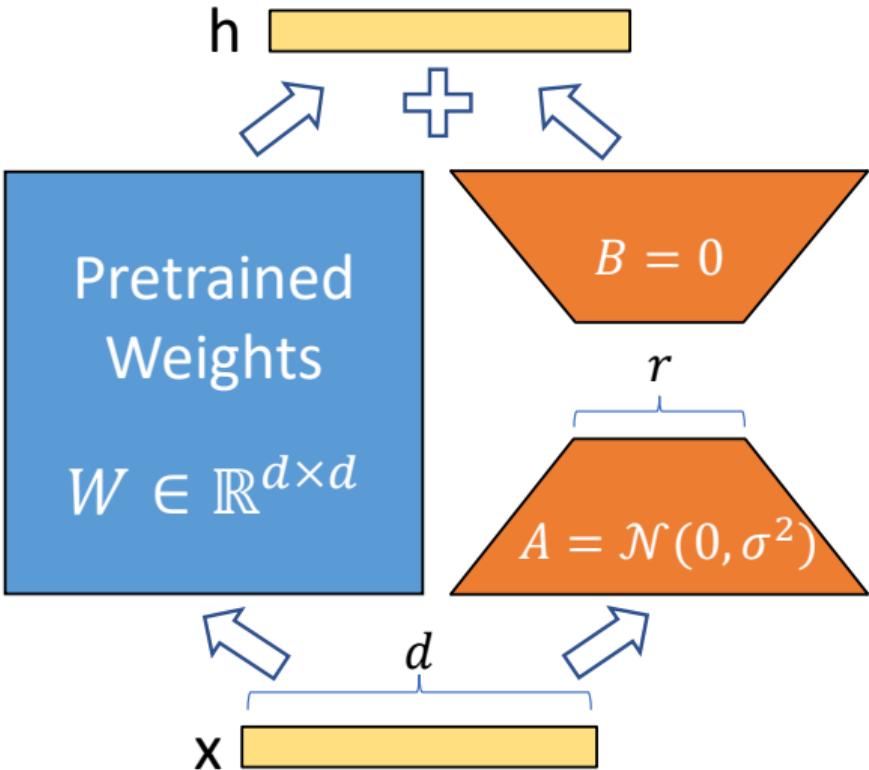
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where  $\Delta W = AB^T$ , with  $A$  and  $B$  being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- $A$  is initialized as usual, while  $B$  is initialized with zeroes in order to start from identity mapping
- $r$  is typically selected between 2 and 64
- Usually applied to attention modules

<sup>32</sup>LoRA: Low-Rank Adaptation of Large Language Models



LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

$$W_{\text{new}} = W + \Delta W$$

where  $\Delta W = AB^T$ , with  $A$  and  $B$  being low-rank matrices. This reduces computational and memory overhead while maintaining model performance.

- $A$  is initialized as usual, while  $B$  is initialized with zeroes in order to start from identity mapping
- $r$  is typically selected between 2 and 64
- Usually applied to attention modules

$$h = W_{\text{new}}x = Wx + \Delta Wx = Wx + AB^Tx$$

<sup>32</sup>LoRA: Low-Rank Adaptation of Large Language Models

# Использование представления Кашина для квантизации весов LLM<sup>33</sup>



Рис. 28: Схема алгоритма, позволяющего квантизировать веса нейросети с помощью матричного разложения.

<sup>33</sup>Quantization of Large Language Models with an Overdetermined Basis

## Тренды

# Notable AI Models

## Training compute (FLOP)



Рис. 29: Динамика вычислений, необходимых для обучения моделей. Источник

# Notable AI Models

EPOCH AI

Training compute (FLOP)

483 Results



Рис. 30: Динамика вычислений, необходимых для обучения нейросетевых моделей. Источник

## Notable AI Models

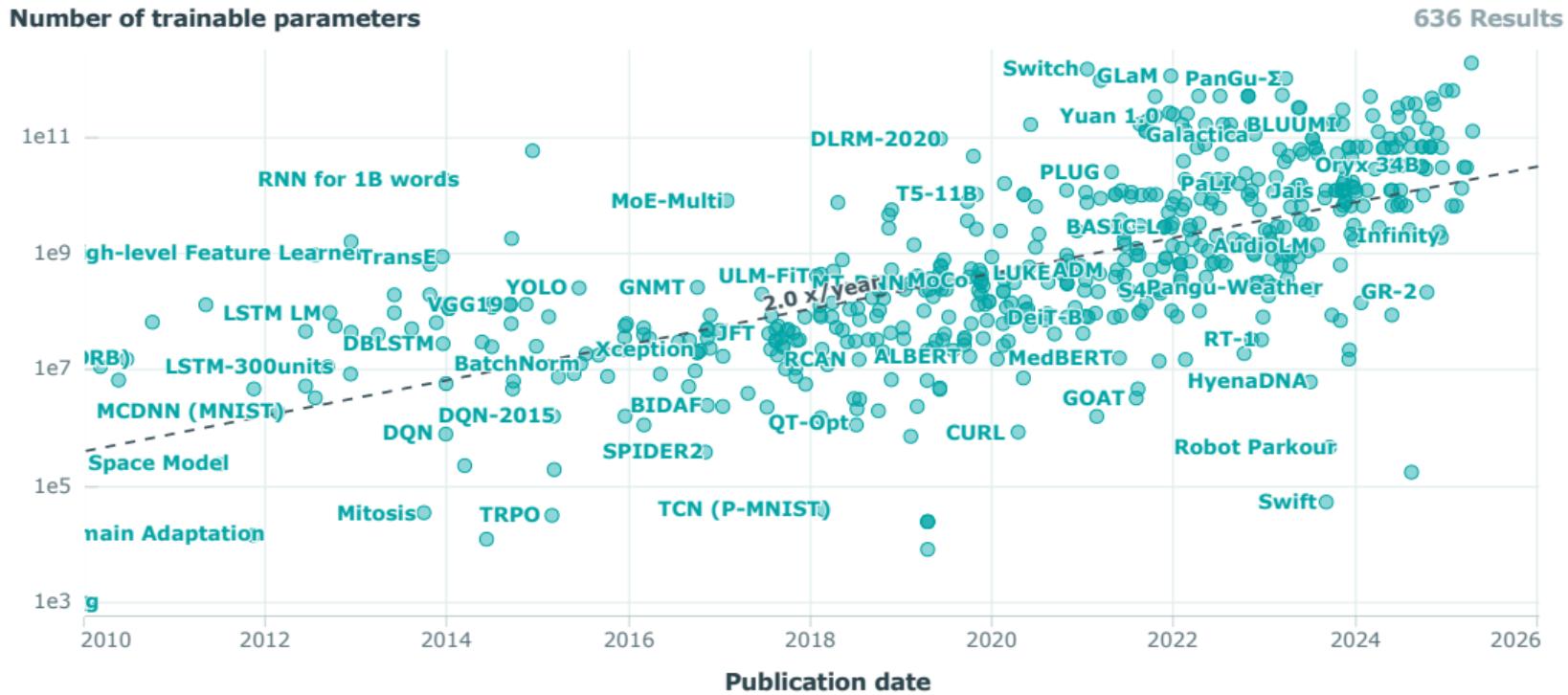


Рис. 31: Динамика количества обучаемых параметров нейросетевых моделей. Источник