

# **Advanced Stochastic Gradient Methods: Adaptive gradient methods + introduction to neural network training**

**Daniil Merkulov**

Optimization methods. MIPT

## Finite-sum problem

## Finite-sum problem

We consider classic finite-sample average minimization:

$$\min_{x \in \mathbb{R}^p} f(x) = \min_{x \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^n f_i(x)$$

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .

## Finite-sum problem

We consider classic finite-sample average minimization:

$$\min_{x \in \mathbb{R}^p} f(x) = \min_{x \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^n f_i(x)$$

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .
- Convergence with constant  $\alpha$  or line search.

## Finite-sum problem

We consider classic finite-sample average minimization:

$$\min_{x \in \mathbb{R}^p} f(x) = \min_{x \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^n f_i(x)$$

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .
- Convergence with constant  $\alpha$  or line search.

## Finite-sum problem

We consider classic finite-sample average minimization:

$$\min_{x \in \mathbb{R}^p} f(x) = \min_{x \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^n f_i(x)$$

The gradient descent acts like follows:

$$x_{k+1} = x_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(x) \quad (\text{GD})$$

- Iteration cost is linear in  $n$ .
- Convergence with constant  $\alpha$  or line search.

Let's/ switch from the full gradient calculation to its unbiased estimator, when we randomly choose  $i_k$  index of point at each iteration uniformly:

$$x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k) \quad (\text{SGD})$$

With  $p(i_k = i) = \frac{1}{n}$ , the stochastic gradient is an unbiased estimate of the gradient, given by:

$$\mathbb{E}[\nabla f_{i_k}(x)] = \sum_{i=1}^n p(i_k = i) \nabla f_i(x) = \sum_{i=1}^n \frac{1}{n} \nabla f_i(x) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(x) = \nabla f(x)$$

This indicates that the expected value of the stochastic gradient is equal to the actual gradient of  $f(x)$ .

## Adaptivity or scaling

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{k-1} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{k-1} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{k-1} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.
- Can drastically improve over SGD in sparse problems.

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{k-1} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.
- Can drastically improve over SGD in sparse problems.
- Main weakness is the monotonic accumulation of gradients in the denominator. AdaDelta, Adam, AMSGrad, etc. improve on this, popular in training deep neural networks.

## Adagrad (Duchi, Hazan, and Singer 2010)

Very popular adaptive method. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$ , and update for  $j = 1, \dots, p$ :

$$v_j^{(k)} = v_j^{k-1} + (g_j^{(k)})^2$$
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- AdaGrad does not require tuning the learning rate:  $\alpha > 0$  is a fixed constant, and the learning rate decreases naturally over iterations.
- The learning rate of rare informative features diminishes slowly.
- Can drastically improve over SGD in sparse problems.
- Main weakness is the monotonic accumulation of gradients in the denominator. AdaDelta, Adam, AMSGrad, etc. improve on this, popular in training deep neural networks.
- The constant  $\epsilon$  is typically set to  $10^{-6}$  to ensure that we do not suffer from division by zero or overly large step sizes.

## RMSProp (Tieleman and Hinton, 2012)

An enhancement of AdaGrad that addresses its aggressive, monotonically decreasing learning rate. Uses a moving average of squared gradients to adjust the learning rate for each weight. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$  and update rule for  $j = 1, \dots, p$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- RMSProp divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.

## RMSProp (Tieleman and Hinton, 2012)

An enhancement of AdaGrad that addresses its aggressive, monotonically decreasing learning rate. Uses a moving average of squared gradients to adjust the learning rate for each weight. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$  and update rule for  $j = 1, \dots, p$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- RMSProp divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- Allows for a more nuanced adjustment of learning rates than AdaGrad, making it suitable for non-stationary problems.

## RMSProp (Tieleman and Hinton, 2012)

An enhancement of AdaGrad that addresses its aggressive, monotonically decreasing learning rate. Uses a moving average of squared gradients to adjust the learning rate for each weight. Let  $g^{(k)} = \nabla f_{i_k}(x^{(k-1)})$  and update rule for  $j = 1, \dots, p$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{g_j^{(k)}}{\sqrt{v_j^{(k)} + \epsilon}}$$

### Notes:

- RMSProp divides the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight.
- Allows for a more nuanced adjustment of learning rates than AdaGrad, making it suitable for non-stationary problems.
- Commonly used in training neural networks, particularly in recurrent neural networks.

## Adadelta (Zeiler, 2012)

An extension of RMSProp that seeks to reduce its dependence on a manually set global learning rate. Instead of accumulating all past squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size  $w$ . Update mechanism does not require learning rate  $\alpha$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$\tilde{g}_j^{(k)} = \frac{\sqrt{\Delta x_j^{(k-1)} + \epsilon}}{\sqrt{v_j^{(k)} + \epsilon}} g_j^{(k)}$$

$$x_j^{(k)} = x_j^{(k-1)} - \tilde{g}_j^{(k)}$$

$$\Delta x_j^{(k)} = \rho \Delta x_j^{(k-1)} + (1 - \rho)(\tilde{g}_j^{(k)})^2$$

### Notes:

- Adadelta adapts learning rates based on a moving window of gradient updates, rather than accumulating all past gradients. This way, learning rates adjusted are more robust to changes in model's dynamics.

## Adadelta (Zeiler, 2012)

An extension of RMSProp that seeks to reduce its dependence on a manually set global learning rate. Instead of accumulating all past squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size  $w$ . Update mechanism does not require learning rate  $\alpha$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$\tilde{g}_j^{(k)} = \frac{\sqrt{\Delta x_j^{(k-1)} + \epsilon}}{\sqrt{v_j^{(k)} + \epsilon}} g_j^{(k)}$$

$$x_j^{(k)} = x_j^{(k-1)} - \tilde{g}_j^{(k)}$$

$$\Delta x_j^{(k)} = \rho \Delta x_j^{(k-1)} + (1 - \rho)(\tilde{g}_j^{(k)})^2$$

### Notes:

- Adadelta adapts learning rates based on a moving window of gradient updates, rather than accumulating all past gradients. This way, learning rates adjusted are more robust to changes in model's dynamics.
- The method does not require an initial learning rate setting, making it easier to configure.

## Adadelta (Zeiler, 2012)

An extension of RMSProp that seeks to reduce its dependence on a manually set global learning rate. Instead of accumulating all past squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size  $w$ . Update mechanism does not require learning rate  $\alpha$ :

$$v_j^{(k)} = \gamma v_j^{(k-1)} + (1 - \gamma)(g_j^{(k)})^2$$

$$\tilde{g}_j^{(k)} = \frac{\sqrt{\Delta x_j^{(k-1)} + \epsilon}}{\sqrt{v_j^{(k)} + \epsilon}} g_j^{(k)}$$

$$x_j^{(k)} = x_j^{(k-1)} - \tilde{g}_j^{(k)}$$

$$\Delta x_j^{(k)} = \rho \Delta x_j^{(k-1)} + (1 - \rho)(\tilde{g}_j^{(k)})^2$$

### Notes:

- Adadelta adapts learning rates based on a moving window of gradient updates, rather than accumulating all past gradients. This way, learning rates adjusted are more robust to changes in model's dynamics.
- The method does not require an initial learning rate setting, making it easier to configure.
- Often used in deep learning where parameter scales differ significantly across layers.

## Adam (Kingma and Ba, 2014) <sup>1</sup> <sup>2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$

$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction:

$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$

$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update:

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.

## Adam (Kingma and Ba, 2014) <sup>1</sup> <sup>2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$

$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction:

$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$

$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update:

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире

## Adam (Kingma and Ba, 2014) <sup>1</sup> <sup>2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$

$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction:

$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$

$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update:

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье

## Adam (Kingma and Ba, 2014) <sup>1</sup> <sup>2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$

$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction:

$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$

$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update:

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье
- Не сходится для некоторых простых задач (даже выпуклых)

## Adam (Kingma and Ba, 2014) <sup>1</sup> <sup>2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$

$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction:

$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$

$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update:

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье
- Не сходится для некоторых простых задач (даже выпуклых)
- Почему-то очень хорошо работает для некоторых сложных задач

## Adam (Kingma and Ba, 2014) <sup>1</sup> <sup>2</sup>

Combines elements from both AdaGrad and RMSProp. It considers an exponentially decaying average of past gradients and squared gradients.

EMA:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$

$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

Bias correction:

$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}$$

$$\hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

Update:

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

### Notes:

- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Одна из самых цитируемых научных работ в мире
- В 2018-2019 годах вышли статьи, указывающие на ошибку в оригинальной статье
- Не сходится для некоторых простых задач (даже выпуклых)
- Почему-то очень хорошо работает для некоторых сложных задач
- Гораздо лучше работает для языковых моделей, чем для задач компьютерного зрения - почему?

<sup>1</sup>Adam: A Method for Stochastic Optimization

<sup>2</sup>On the Convergence of Adam and Beyond

## AdamW (Loshchilov & Hutter, 2017)

Addresses a common issue with  $\ell_2$  regularization in adaptive optimizers like Adam. Standard  $\ell_2$  regularization adds  $\lambda\|x\|^2$  to the loss, resulting in a gradient term  $\lambda x$ . In Adam, this term gets scaled by the adaptive learning rate  $(\sqrt{\hat{v}_j} + \epsilon)$ , coupling the weight decay to the gradient magnitudes.

AdamW decouples weight decay from the gradient adaptation step.

Update rule:

$$m_j^{(k)} = \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)}$$
$$v_j^{(k)} = \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2$$

$$\hat{m}_j = \frac{m_j^{(k)}}{1 - \beta_1^k}, \quad \hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k}$$

$$x_j^{(k)} = x_j^{(k-1)} - \alpha \left( \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon} + \lambda x_j^{(k-1)} \right)$$

Notes:

- The weight decay term  $\lambda x_j^{(k-1)}$  is added *after* the adaptive gradient step.

## AdamW (Loshchilov & Hutter, 2017)

Addresses a common issue with  $\ell_2$  regularization in adaptive optimizers like Adam. Standard  $\ell_2$  regularization adds  $\lambda\|x\|^2$  to the loss, resulting in a gradient term  $\lambda x$ . In Adam, this term gets scaled by the adaptive learning rate  $(\sqrt{\hat{v}_j} + \epsilon)$ , coupling the weight decay to the gradient magnitudes.

AdamW decouples weight decay from the gradient adaptation step.

Update rule:

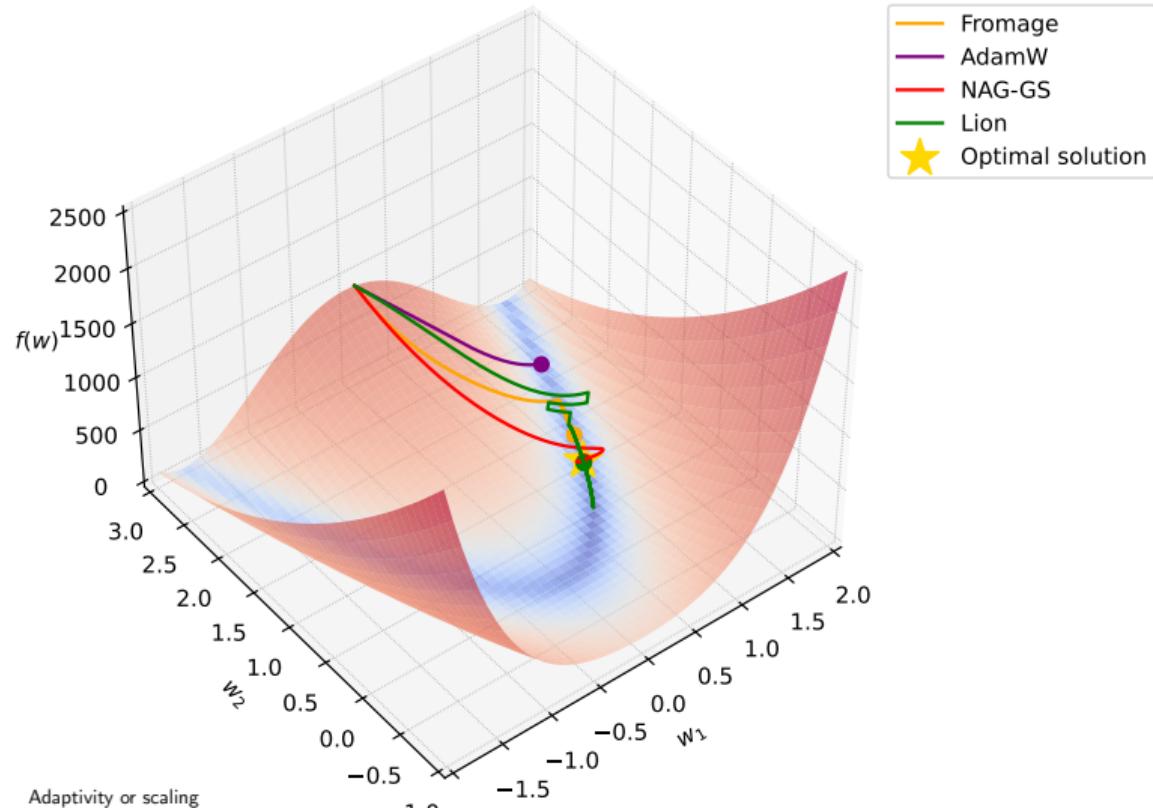
$$\begin{aligned} m_j^{(k)} &= \beta_1 m_j^{(k-1)} + (1 - \beta_1) g_j^{(k)} \\ v_j^{(k)} &= \beta_2 v_j^{(k-1)} + (1 - \beta_2) (g_j^{(k)})^2 \\ \hat{m}_j &= \frac{m_j^{(k)}}{1 - \beta_1^k}, \quad \hat{v}_j = \frac{v_j^{(k)}}{1 - \beta_2^k} \\ x_j^{(k)} &= x_j^{(k-1)} - \alpha \left( \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon} + \lambda x_j^{(k-1)} \right) \end{aligned}$$

Notes:

- The weight decay term  $\lambda x_j^{(k-1)}$  is added *after* the adaptive gradient step.
- Widely adopted in training transformers and other large models. Default choice for huggingface trainer.

# A lot of them

Rosenbrock Function.  
Adaptive stochastic gradient algorithms.  
Learning rate 0.003



## How to compare them? AlgoPerf benchmark<sup>3 4</sup>

- **AlgoPerf Benchmark:** Compares NN training algorithms with two rulesets:

## How to compare them? AlgoPerf benchmark<sup>3 4</sup>

- **AlgoPerf Benchmark:** Compares NN training algorithms with two rulesets:
  - **External Tuning:** Simulates hyperparameter tuning with limited resources (5 trials, quasirandom search). Scored on median fastest time to target across 5 studies.

## How to compare them? AlgoPerf benchmark<sup>3 4</sup>

- **AlgoPerf Benchmark:** Compares NN training algorithms with two rulesets:
  - **External Tuning:** Simulates hyperparameter tuning with limited resources (5 trials, quasirandom search). Scored on median fastest time to target across 5 studies.
  - **Self-Tuning:** Simulates automated tuning on a single machine (fixed/inner-loop tuning, 3x budget). Scored on median runtime across 5 studies.

## How to compare them? AlgoPerf benchmark<sup>3 4</sup>

- **AlgoPerf Benchmark:** Compares NN training algorithms with two rulesets:
  - **External Tuning:** Simulates hyperparameter tuning with limited resources (5 trials, quasirandom search). Scored on median fastest time to target across 5 studies.
  - **Self-Tuning:** Simulates automated tuning on a single machine (fixed/inner-loop tuning, 3x budget). Scored on median runtime across 5 studies.
- **Scoring:** Aggregates workload scores using performance profiles. Profiles plot the fraction of workloads solved within a time factor  $\tau$  relative to the fastest submission. Final score: normalized area under the profile (1.0 = fastest on all workloads).

## How to compare them? AlgoPerf benchmark<sup>3 4</sup>

- **AlgoPerf Benchmark:** Compares NN training algorithms with two rulesets:
  - **External Tuning:** Simulates hyperparameter tuning with limited resources (5 trials, quasirandom search). Scored on median fastest time to target across 5 studies.
  - **Self-Tuning:** Simulates automated tuning on a single machine (fixed/inner-loop tuning, 3x budget). Scored on median runtime across 5 studies.
- **Scoring:** Aggregates workload scores using performance profiles. Profiles plot the fraction of workloads solved within a time factor  $\tau$  relative to the fastest submission. Final score: normalized area under the profile (1.0 = fastest on all workloads).
- **Computational Cost:** Scoring required  $\sim 49,240$  total hours on 8x NVIDIA V100 GPUs (avg.  $\sim 3469$ h/external,  $\sim 1847$ h/self-tuning submission).

<sup>3</sup>Benchmarking Neural Network Training Algorithms

<sup>4</sup>Accelerating neural network training: An analysis of the AlgoPerf competition

## AlgoPerf benchmark

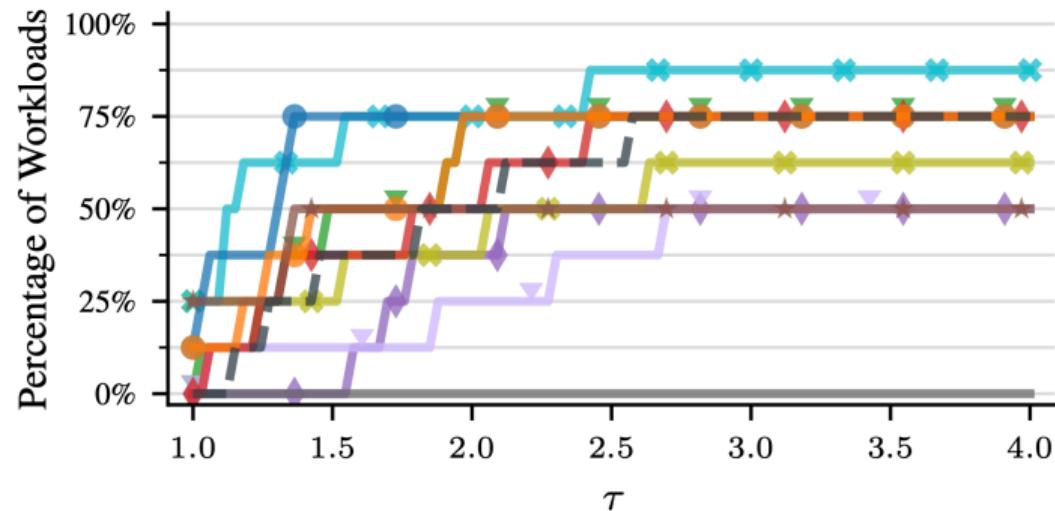
**Summary of fixed base workloads in the AlgoPerf benchmark.** Losses include cross-entropy (CE), mean absolute error (L1), and Connectionist Temporal Classification loss (CTC). Additional evaluation metrics are structural similarity index measure (SSIM), (word) error rate (ER & WER), mean average precision (mAP), and bilingual evaluation underway score (BLEU). The \runtime budget is that of the external tuning ruleset, the self-tuning ruleset allows 3 $\times$  longer training.

| Task                          | Dataset     | Model       | Loss | Metric | Validation Target | Runtime Budget |
|-------------------------------|-------------|-------------|------|--------|-------------------|----------------|
| Clickthrough rate prediction  | CRITEO 1TB  | DLRMSMALL   | CE   | CE     | 0.123735          | 7703           |
| MRI reconstruction            | FASTMRI     | U-NET       | L1   | SSIM   | 0.7344            | 8859           |
| Image classification          | IMAGENET    | ResNet-50   | CE   | ER     | 0.22569           | 63,008         |
|                               |             | ViT         | CE   | ER     | 0.22691           | 77,520         |
| Speech recognition            | LIBRISPEECH | Conformer   | CTC  | WER    | 0.085884          | 61,068         |
|                               |             | DeepSpeech  | CTC  | WER    | 0.119936          | 55,506         |
| Molecular property prediction | OGBG        | GNN         | CE   | mAP    | 0.28098           | 18,477         |
| Translation                   | WMT         | Transformer | CE   | BLEU   | 30.8491           | 48,151         |

# AlgoPerf benchmark

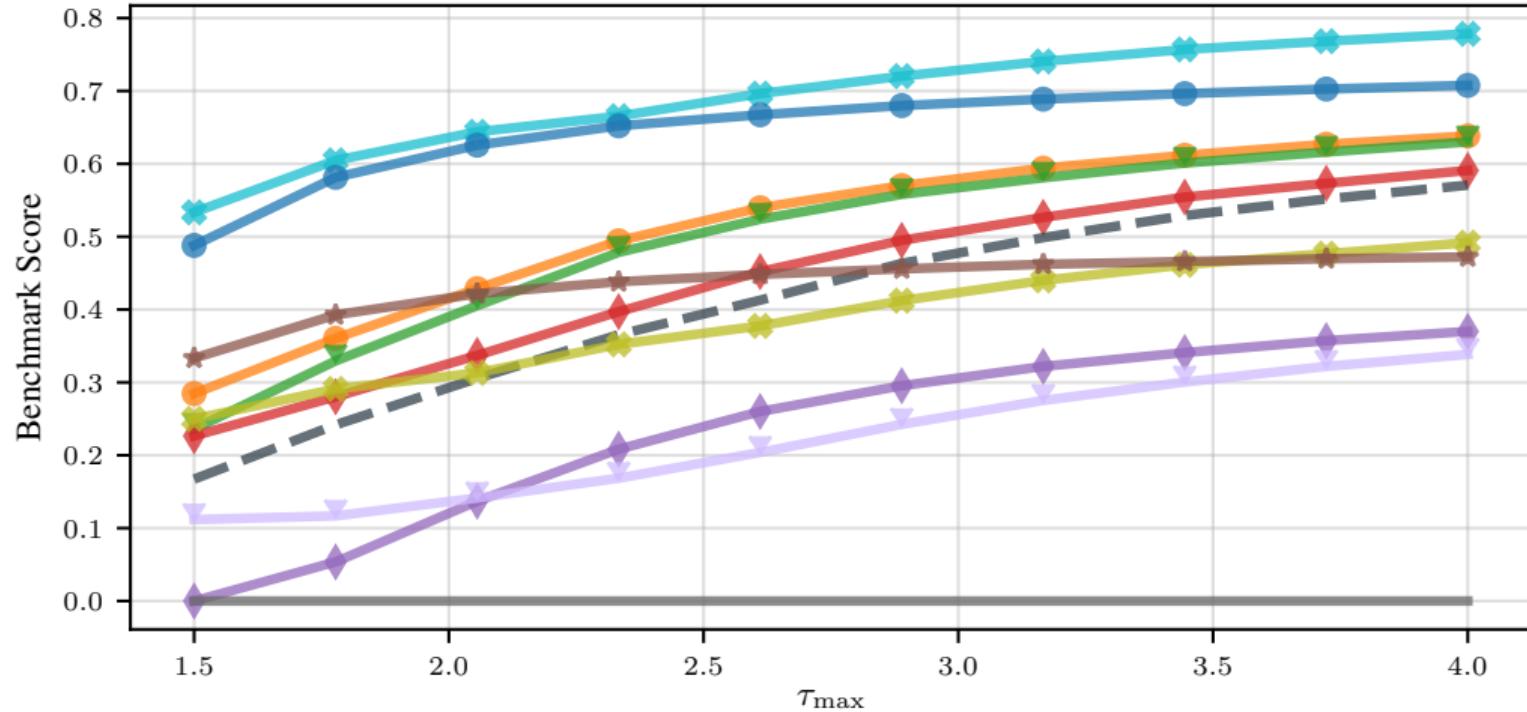
| Submission                  | Line | Score  |
|-----------------------------|------|--------|
| PYTORCH DISTRIBUTED SHAMPOO |      | 0.7784 |
| SCHEDULE FREE ADAMW         |      | 0.7077 |
| GENERALIZED ADAM            |      | 0.6383 |
| CYCLIC LR                   |      | 0.6301 |
| NADAMP                      |      | 0.5909 |
| BASELINE                    |      | 0.5707 |
| AMOS                        |      | 0.4918 |
| CASPR ADAPTIVE              |      | 0.4722 |
| LAWA QUEUE                  |      | 0.3699 |
| LAWA EMA                    |      | 0.3384 |
| SCHEDULE FREE PRODIGY       |      | 0      |

(a) External tuning leaderboard



(b) External tuning performance profiles

# AlgoPerf benchmark

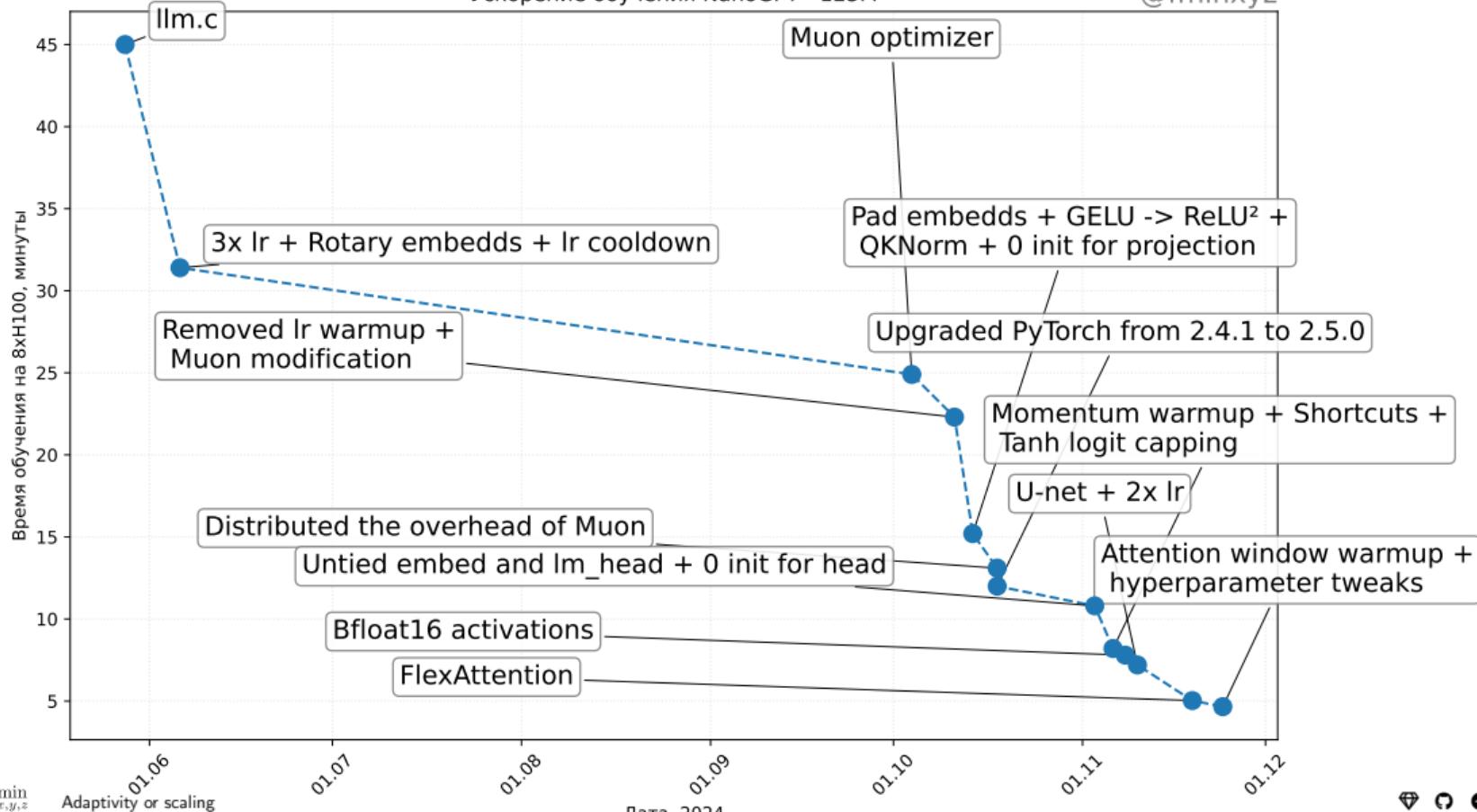


- PyTorch Distr. Shampoo
- Schedule Free AdamW
- Generalized Adam
- Cyclic LR
- NadamP
- Baseline
- Amos
- CASPR Adaptive
- Lawa Queue
- Lawa EMA
- Schedule Free Prodigy

# NanoGPT speedrun

Ускорение обучения NanoGPT - 125M

@fminxyz



## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .

**Notes:**

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .

**Notes:**

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)

**Notes:**

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

**Notes:**

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

**Notes:**

- Aims to capture curvature information more effectively than first-order methods.

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

**Notes:**

- Aims to capture curvature information more effectively than first-order methods.
- Computationally more expensive than Adam but can converge faster or to better solutions in terms of steps.

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

**Notes:**

- Aims to capture curvature information more effectively than first-order methods.
- Computationally more expensive than Adam but can converge faster or to better solutions in terms of steps.
- Requires careful implementation for efficiency (e.g., efficient computation of inverse matrix roots, handling large matrices).

## Shampoo (Gupta, Anil, et al., 2018; Anil et al., 2020)

Stands for **S**tochastic **H**essian-**A**pproximation **M**atrix **P**reconditioning for **O**ptimization **O**f deep networks. It's a method inspired by second-order optimization designed for large-scale deep learning.

**Core Idea:** Approximates the full-matrix AdaGrad pre conditioner using efficient matrix structures, specifically Kronecker products.

For a weight matrix  $W \in \mathbb{R}^{m \times n}$ , the update involves preconditioning using approximations of the statistics matrices  $L \approx \sum_k G_k G_k^T$  and  $R \approx \sum_k G_k^T G_k$ , where  $G_k$  are the gradients.

Simplified concept:

1. Compute gradient  $G_k$ .
2. Update statistics  $L_k = \beta L_{k-1} + (1 - \beta) G_k G_k^T$  and  $R_k = \beta R_{k-1} + (1 - \beta) G_k^T G_k$ .
3. Compute preconditioners  $P_L = L_k^{-1/4}$  and  $P_R = R_k^{-1/4}$ . (Inverse matrix root)
4. Update:  $W_{k+1} = W_k - \alpha P_L G_k P_R$ .

**Notes:**

- Aims to capture curvature information more effectively than first-order methods.
- Computationally more expensive than Adam but can converge faster or to better solutions in terms of steps.
- Requires careful implementation for efficiency (e.g., efficient computation of inverse matrix roots, handling large matrices).
- Variants exist for different tensor shapes (e.g., convolutional layers).

$$\begin{aligned}W_{t+1} &= W_t - \eta(G_t G_t^\top)^{-1/4} G_t (G_t^\top G_t)^{-1/4} \\&= W_t - \eta(US^2U^\top)^{-1/4}(USV^\top)(VS^2V^\top)^{-1/4} \\&= W_t - \eta(US^{-1/2}U^\top)(USV^\top)(VS^{-1/2}V^\top) \\&= W_t - \eta US^{-1/2} S S^{-1/2} V^\top \\&= W_t - \eta U V^\top\end{aligned}$$

---

<sup>5</sup>Deriving Muon

## Neural network training

## Optimization for Neural Network training

Neural network is a function, that takes an input  $x$  and current set of weights (parameters)  $w$  and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU ( $x$ ) or sigmoid):

## Optimization for Neural Network training

Neural network is a function, that takes an input  $x$  and current set of weights (parameters)  $\mathbf{w}$  and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU ( $x$ ) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \dots \circ \sigma_1 \circ w_1 \circ x \quad \mathbf{w} = (W_1, b_1, \dots W_L, b_L),$$

## Optimization for Neural Network training

Neural network is a function, that takes an input  $x$  and current set of weights (parameters)  $\mathbf{w}$  and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU ( $x$ ) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \dots \circ \sigma_1 \circ w_1 \circ x \quad \mathbf{w} = (W_1, b_1, \dots W_L, b_L),$$

where  $L$  is the number of layers,  $\sigma_i$  - non-linear activation function,  $w_i = W_i x + b_i$  - linear layer.

## Optimization for Neural Network training

Neural network is a function, that takes an input  $x$  and current set of weights (parameters)  $\mathbf{w}$  and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU ( $x$ ) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \dots \circ \sigma_1 \circ w_1 \circ x \quad \mathbf{w} = (W_1, b_1, \dots W_L, b_L),$$

where  $L$  is the number of layers,  $\sigma_i$  - non-linear activation function,  $w_i = W_i x + b_i$  - linear layer.

Typically, we aim to find  $\mathbf{w}$  in order to solve some problem (let say to be  $\mathcal{NN}(\mathbf{w}, x_i) \sim y_i$  for some training data  $x_i, y_i$ ). In order to do it, we solve the optimization problem:

## Optimization for Neural Network training

Neural network is a function, that takes an input  $x$  and current set of weights (parameters)  $\mathbf{w}$  and predicts some vector as an output. Note, that a variety of feed-forward neural networks could be represented as a series of linear transformations, followed by some nonlinear function (say, ReLU ( $x$ ) or sigmoid):

$$\mathcal{NN}(\mathbf{w}, x) = \sigma_L \circ w_L \circ \dots \circ \sigma_1 \circ w_1 \circ x \quad \mathbf{w} = (W_1, b_1, \dots W_L, b_L),$$

where  $L$  is the number of layers,  $\sigma_i$  - non-linear activation function,  $w_i = W_i x + b_i$  - linear layer.

Typically, we aim to find  $\mathbf{w}$  in order to solve some problem (let say to be  $\mathcal{NN}(\mathbf{w}, x_i) \sim y_i$  for some training data  $x_i, y_i$ ). In order to do it, we solve the optimization problem:

$$L(\mathbf{w}, X, y) \rightarrow \min_{\mathbf{w}} \quad \frac{1}{N} \sum_{i=1}^N l(\mathbf{w}, x_i, y_i) \rightarrow \min_{\mathbf{w}}$$

## Loss functions

In the context of training neural networks, the loss function, denoted by  $l(\mathbf{w}, x_i, y_i)$ , measures the discrepancy between the predicted output  $\mathcal{NN}(\mathbf{w}, x_i)$  and the true output  $y_i$ . The choice of the loss function can significantly influence the training process. Common loss functions include:

### Mean Squared Error (MSE)

Used primarily for regression tasks. It computes the square of the difference between predicted and true values, averaged over all samples.

$$\text{MSE}(\mathbf{w}, X, y) = \frac{1}{N} \sum_{i=1}^N (\mathcal{NN}(\mathbf{w}, x_i) - y_i)^2$$

### Cross-Entropy Loss

Typically used for classification tasks. It measures the dissimilarity between the true label distribution and the predictions, providing a probabilistic interpretation of classification.

$$\text{Cross-Entropy}(\mathbf{w}, X, y) = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log(\mathcal{NN}(\mathbf{w}, x_i)_c)$$

where  $y_{i,c}$  is a binary indicator (0 or 1) if class label  $c$  is the correct classification for observation  $i$ , and  $C$  is the number of classes.

# Simple example: Fashion MNIST classification problem

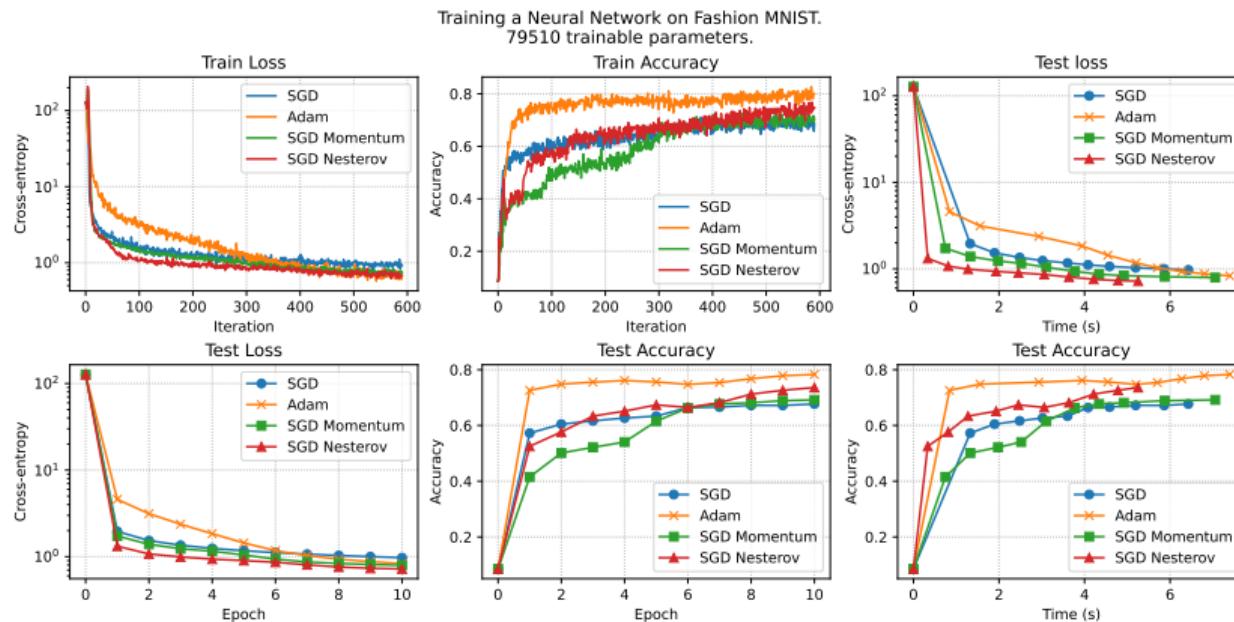
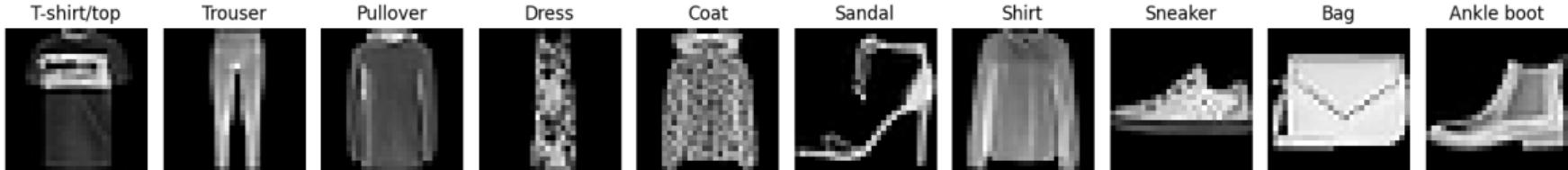
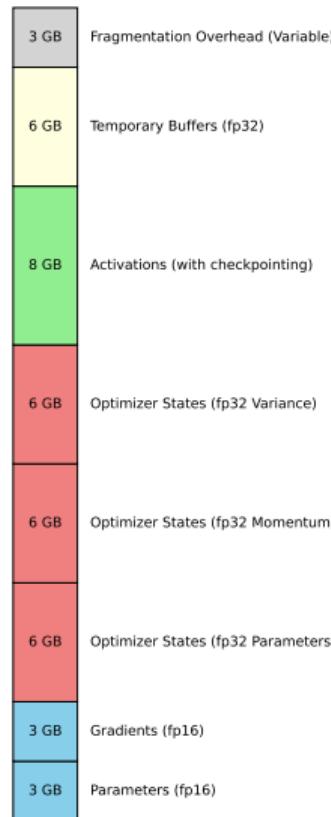


Рис. 2: [Open in colab](#)

## GPT-2 training Memory footprint

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

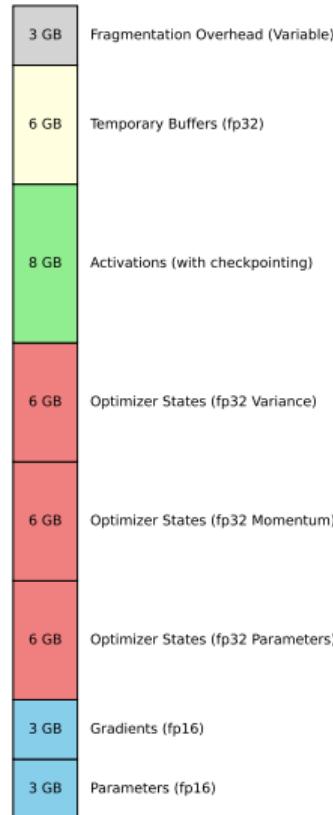
## Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.

## Memory Requirements Example:

## Residual Memory Consumption:

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

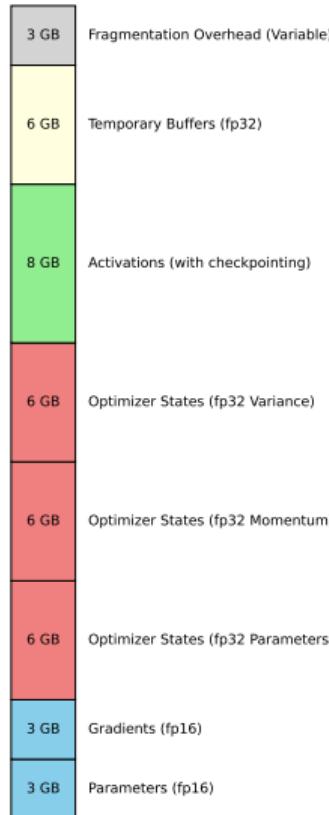
## Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

## Memory Requirements Example:

## Residual Memory Consumption:

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

## Model States:

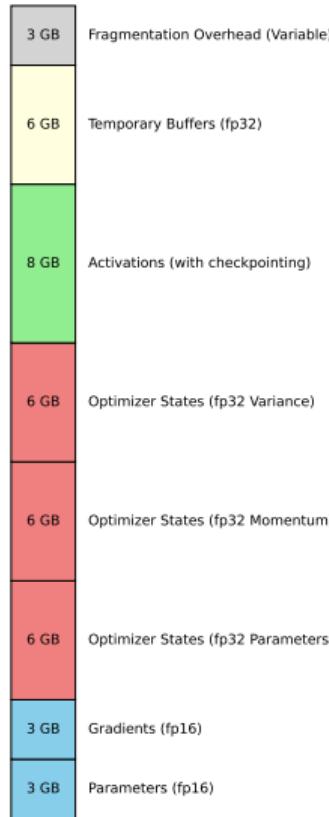
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

## Memory Requirements Example:

- Training with Adam in mixed precision for a model with  $\Psi$  parameters:  $2\Psi$  bytes for fp16 parameters and gradients,  $12\Psi$  bytes for optimizer states (parameters, momentum, variance).

## Residual Memory Consumption:

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

## Model States:

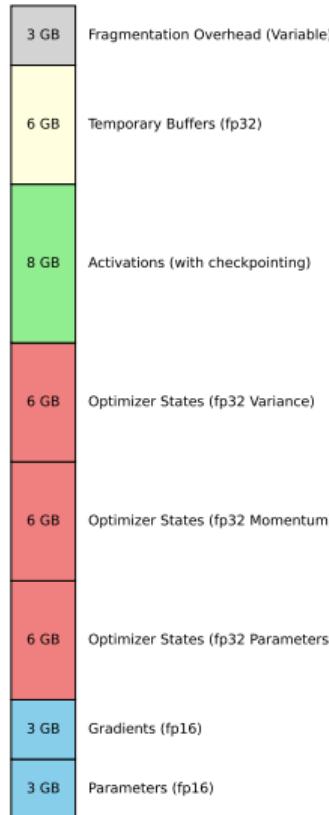
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

## Memory Requirements Example:

- Training with Adam in mixed precision for a model with  $\Psi$  parameters:  $2\Psi$  bytes for fp16 parameters and gradients,  $12\Psi$  bytes for optimizer states (parameters, momentum, variance).
- Total:  $16\Psi$  bytes; for GPT-2 with 1.5B parameters, this equals 24GB.

## Residual Memory Consumption:

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

## Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

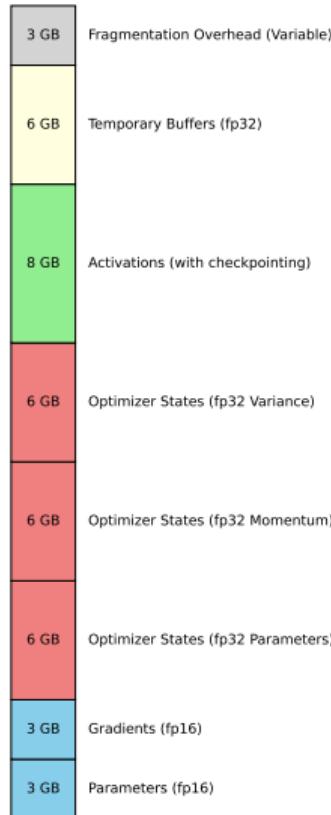
## Memory Requirements Example:

- Training with Adam in mixed precision for a model with  $\Psi$  parameters:  $2\Psi$  bytes for fp16 parameters and gradients,  $12\Psi$  bytes for optimizer states (parameters, momentum, variance).
- Total:  $16\Psi$  bytes; for GPT-2 with 1.5B parameters, this equals 24GB.

## Residual Memory Consumption:

- Activations: Significant memory usage, e.g., 1.5B parameter GPT-2 model with sequence length 1K and batch size 32 requires ~60GB.

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

## Model States:

- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

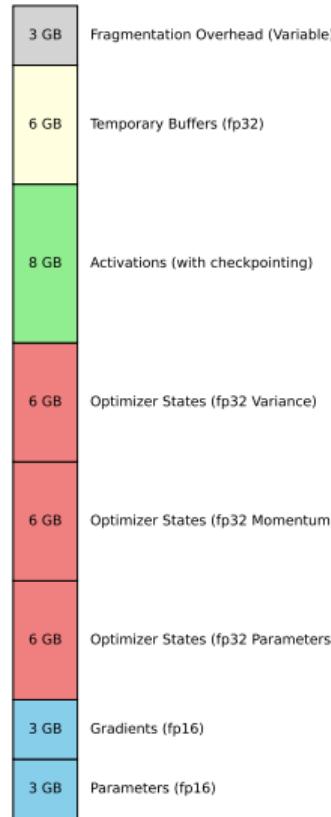
## Memory Requirements Example:

- Training with Adam in mixed precision for a model with  $\Psi$  parameters:  $2\Psi$  bytes for fp16 parameters and gradients,  $12\Psi$  bytes for optimizer states (parameters, momentum, variance).
- Total:  $16\Psi$  bytes; for GPT-2 with 1.5B parameters, this equals 24GB.

## Residual Memory Consumption:

- Activations: Significant memory usage, e.g., 1.5B parameter GPT-2 model with sequence length 1K and batch size 32 requires ~60GB.
- Activation checkpointing can reduce activation memory by about 50%, with a 33% recomputation overhead.

# GPT-2 training Memory footprint



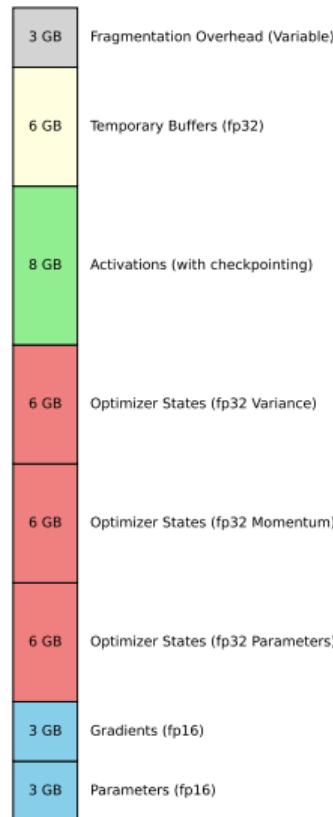
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

## Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.

## Memory Fragmentation:

# GPT-2 training Memory footprint



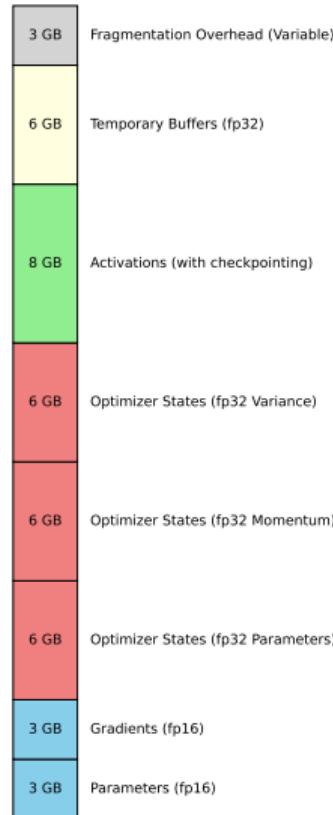
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

## Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.
- For large models, temporary buffers can consume substantial memory (e.g., 6GB for 1.5B parameter model with fp32 buffer).

## Memory Fragmentation:

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

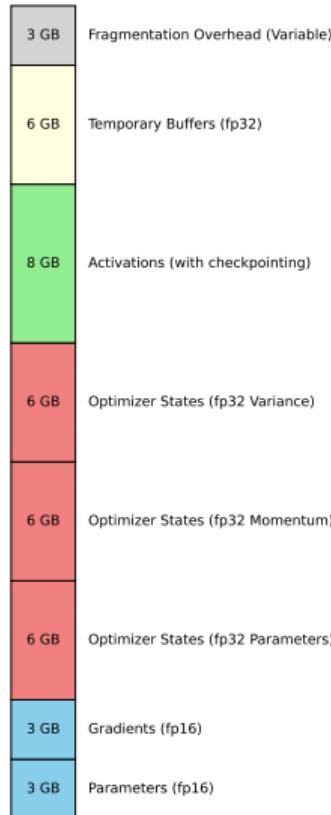
## Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.
- For large models, temporary buffers can consume substantial memory (e.g., 6GB for 1.5B parameter model with fp32 buffer).

## Memory Fragmentation:

- Memory fragmentation can cause out-of-memory issues despite available memory, as contiguous blocks are required.

# GPT-2 training Memory footprint



Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

## Temporary Buffers:

- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.
- For large models, temporary buffers can consume substantial memory (e.g., 6GB for 1.5B parameter model with fp32 buffer).

## Memory Fragmentation:

- Memory fragmentation can cause out-of-memory issues despite available memory, as contiguous blocks are required.
- In some cases, over 30% of memory remains unusable due to fragmentation.