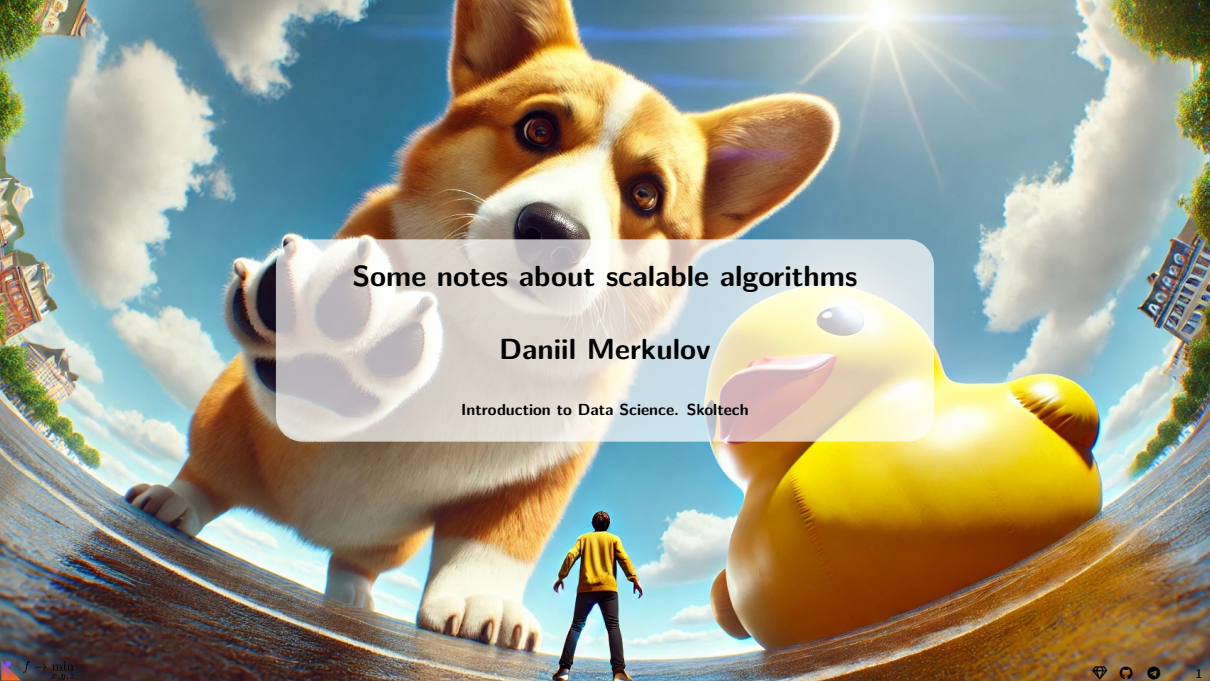


Some notes about scalable algorithms

Daniil Merkulov

Introduction to Data Science. Skoltech

Stochastic Gradient Descent

Example: multidimensional scaling problem

Suppose, we have a pairwise distance matrix for N d-dimensional objects $D \in \mathbb{R}^{N \times N}$. Given this matrix, our goal is to recover the initial coordinates $W_i \in \mathbb{R}^d,\ i = 1, \ldots, N$.

Example: multidimensional scaling problem

Suppose, we have a pairwise distance matrix for N d-dimensional objects $D \in \mathbb{R}^{N \times N}$. Given this matrix, our goal is to recover the initial coordinates $W_i \in \mathbb{R}^d,\ i = 1, \dots, N$.

$$L(W) = \sum_{i,j=1}^{N} \left(\|W_i - W_j\|_2^2 - D_{i,j} \right)^2 \to \min_{W \in \mathbb{R}^{N \times d}}$$

Example: multidimensional scaling problem

Suppose, we have a pairwise distance matrix for N d-dimensional objects $D \in \mathbb{R}^{N \times N}$. Given this matrix, our goal is to recover the initial coordinates $W_i \in \mathbb{R}^d$, $i = 1, \ldots, N$.

$$L(W) = \sum_{i,j=1}^{N} \left(\|W_i - W_j\|_2^2 - D_{i,j} \right)^2 \to \min_{W \in \mathbb{R}^{N \times d}}$$

Link to a nice visualization ♣, where one can see, that gradient-free methods handle this problem much slower, especially in higher dimensions.

> **Question**
>
> Is it somehow connected with PCA?

Example: multidimensional scaling problem

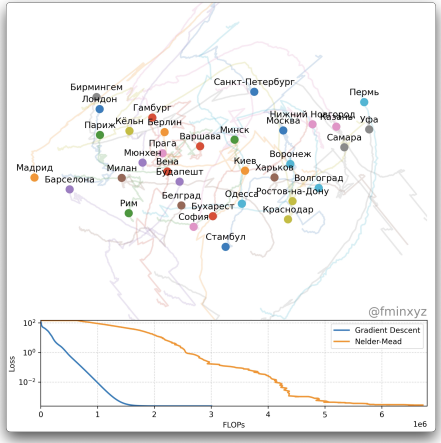


Figure 1: Link to the animation

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:

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

- Convergence with constant α 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:

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

- Convergence with constant α or line search.
- Iteration cost is linear in n. For ImageNet $n \approx 1.4 \cdot 10^7$, for WikiText $n \approx 10^8$.

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:

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

- Convergence with constant α or line search.
- Iteration cost is linear in n. For ImageNet $n \approx 1.4 \cdot 10^7$, for WikiText $n \approx 10^8$.

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:

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

- Convergence with constant α or line search.
- Iteration cost is linear in n. For ImageNet $n \approx 1.4 \cdot 10^7$, for WikiText $n \approx 10^8$.

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:

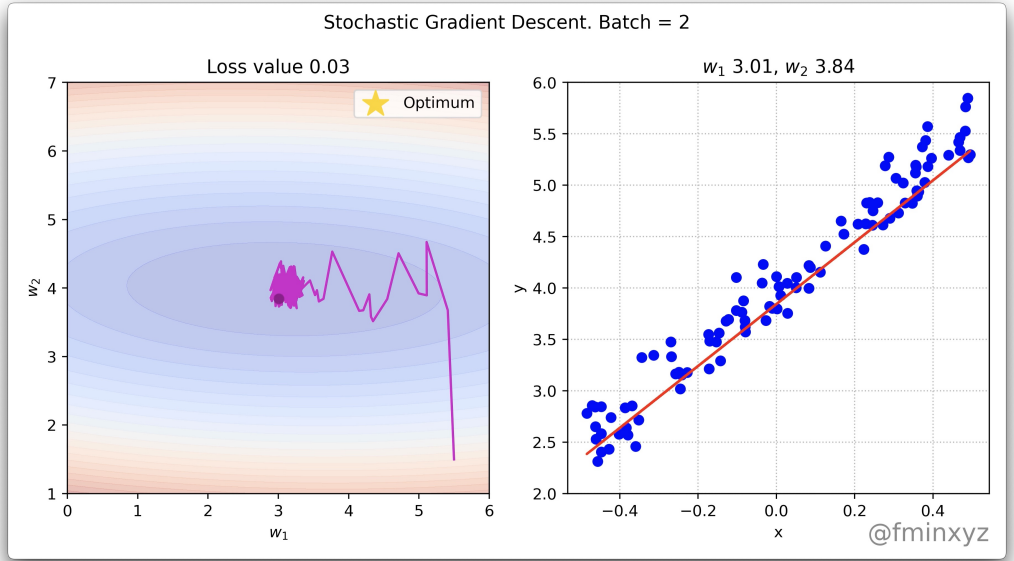
$$x_{k+1} = x_k - \alpha_k \nabla f_{i_k}(x_k) \tag{SGD}$$

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 $f(x)$.

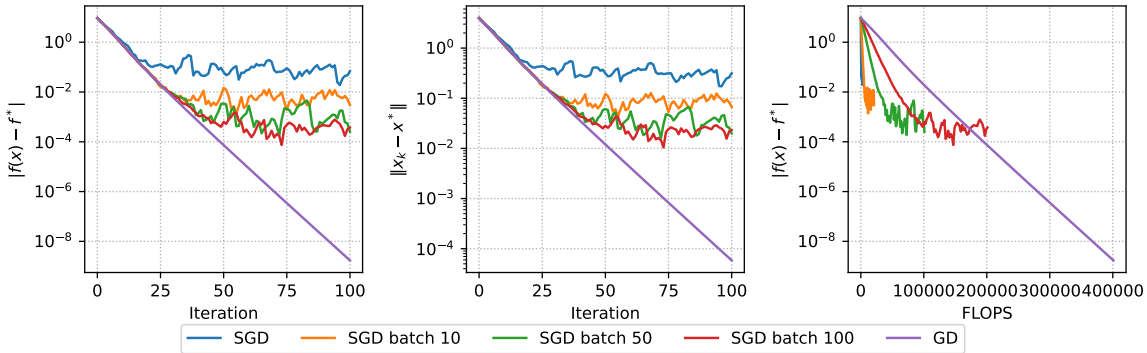
Typical behaviour



Main problem of SGD

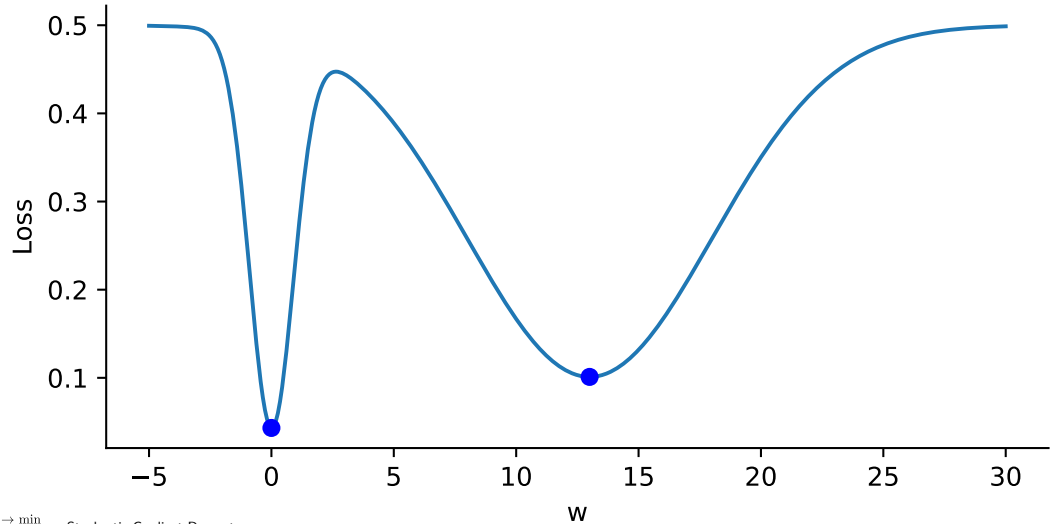
$$f(x) = \frac{\mu}{2}\|x\|_2^2 + \frac{1}{m}\sum_{i=1}^{m}\log(1 + \exp(-y_i\langle a_i, x\rangle)) \to \min_{x\in\mathbb{R}^n}$$

Strongly convex binary logistic regression. m=200, n=10, mu=1.



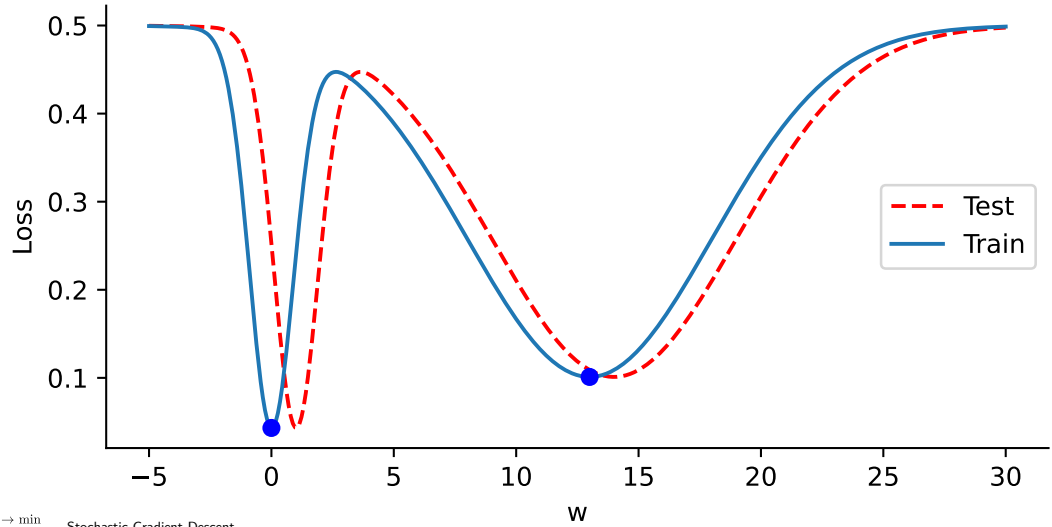
Wide vs narrow local minima

Узкие и широкие локальные минимумы

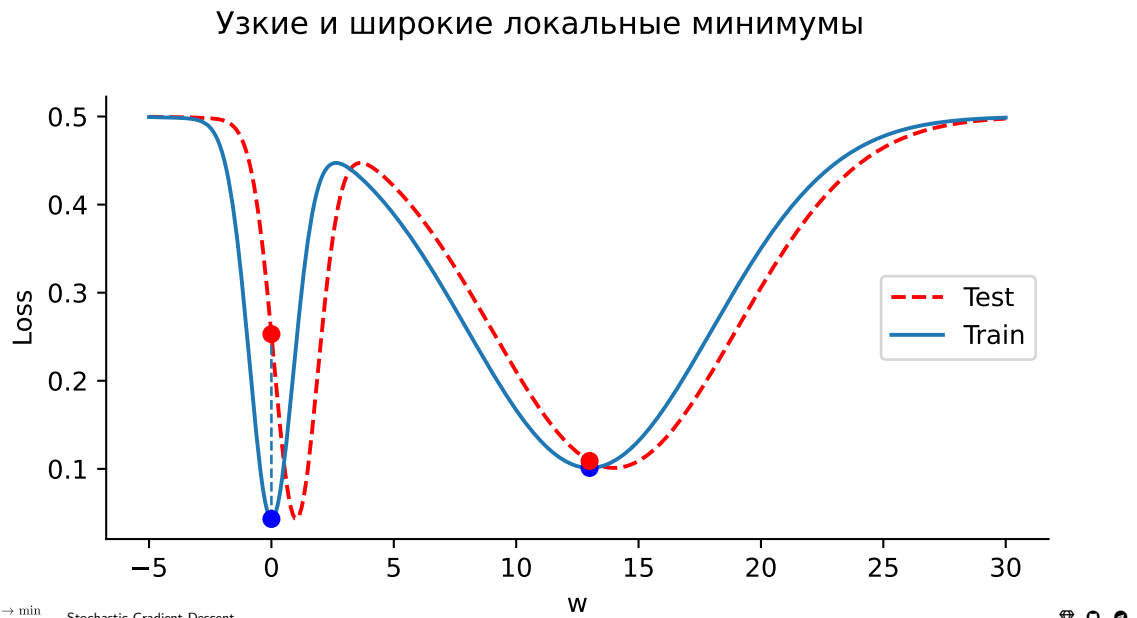


Wide vs narrow local minima

Узкие и широкие локальные минимумы

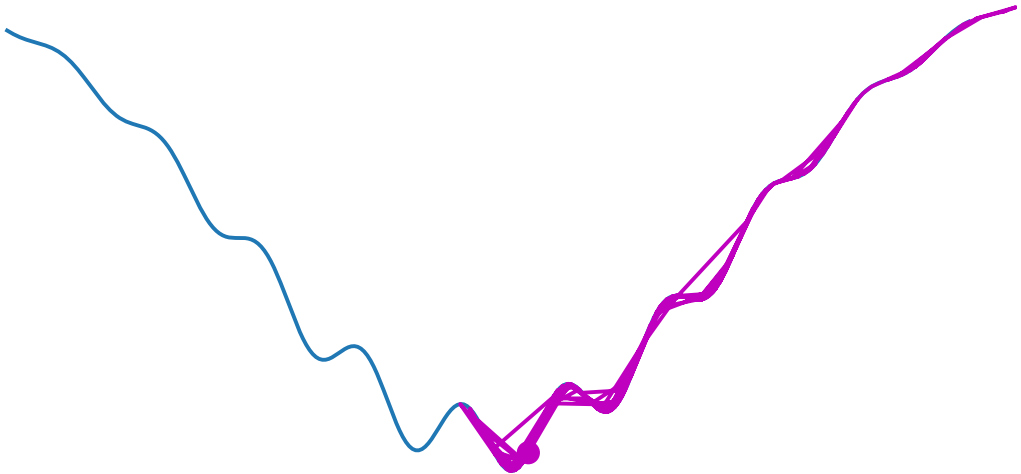


Wide vs narrow local minima



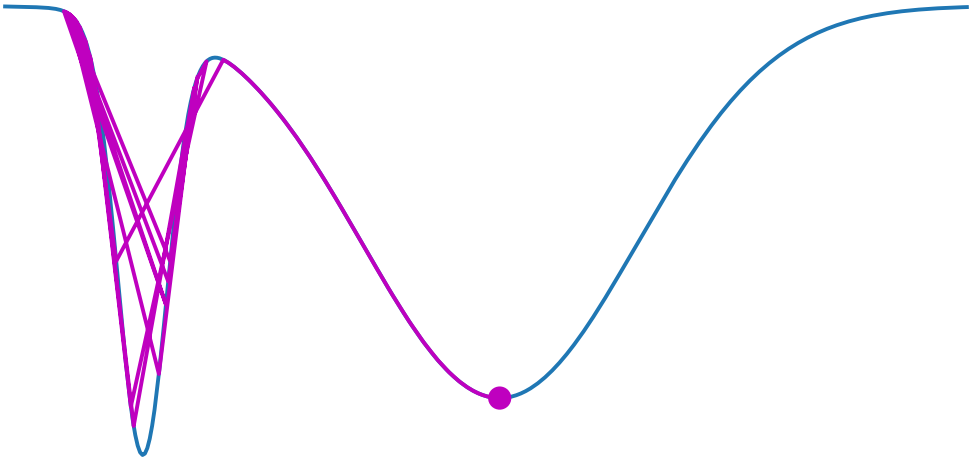
Stochasticity allows to escape local minima

Стохастический градиентный спуск
выпрыгивает из локальных минимумов



Local divergence can also be benefitial

Градиентный спуск с большим шагом
избегает узкого локального минимума



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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 ϵ 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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 $j = 1, \ldots, p$:

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

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

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 α:

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

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

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

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

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 α:

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

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

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

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

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 α:

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

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

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

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

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)

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

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

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

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

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

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.

Adam (Kingma and Ba, 2014)

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

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

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

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

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

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.
- 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)

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

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

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

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

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

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.
- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Highly popular in training deep learning models, owing to its efficiency and straightforward implementation.

Adam (Kingma and Ba, 2014)

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

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

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

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

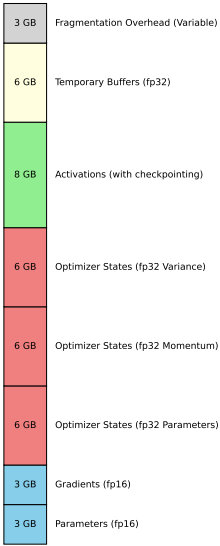
$$x_j^{(k)} = x_j^{(k-1)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$$

Notes:

- Adam is suitable for large datasets and high-dimensional optimization problems.
- It corrects the bias towards zero in the initial moments seen in other methods like RMSProp, making the estimates more accurate.
- Highly popular in training deep learning models, owing to its efficiency and straightforward implementation.
- However, the proposed algorithm in initial version does not converge even in convex setting (later fixes

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.

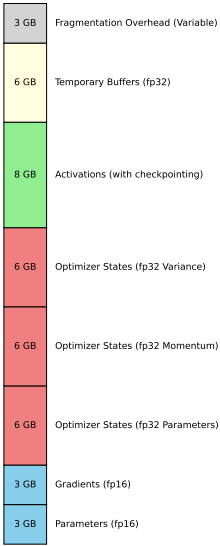
Model States:

- 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.

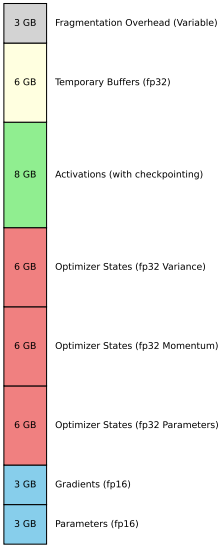
Model States:

- 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:

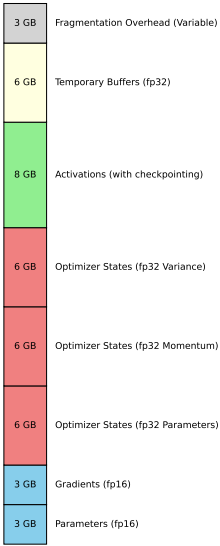
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ 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:

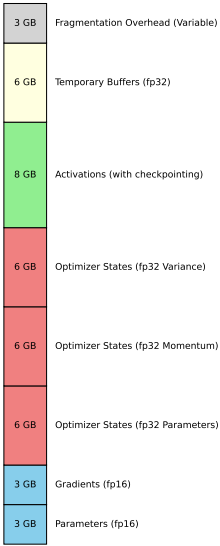
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ bytes for optimizer states (parameters, momentum, variance).
- Total: 16Ψ 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.

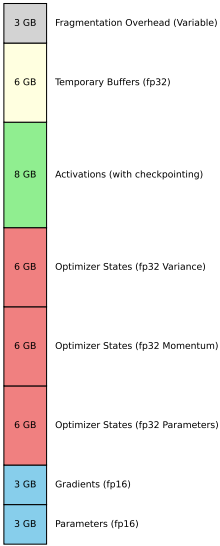
Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ bytes for optimizer states (parameters, momentum, variance).
- Total: 16Ψ 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.

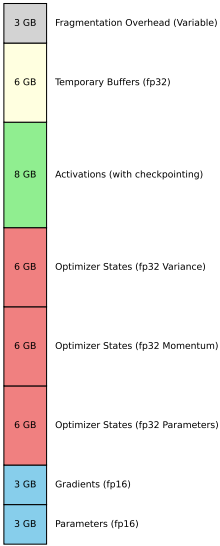
Memory Requirements Example:

- Training with Adam in mixed precision for a model with Ψ parameters: 2Ψ bytes for fp16 parameters and gradients, 12Ψ bytes for optimizer states (parameters, momentum, variance).
- Total: 16Ψ 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



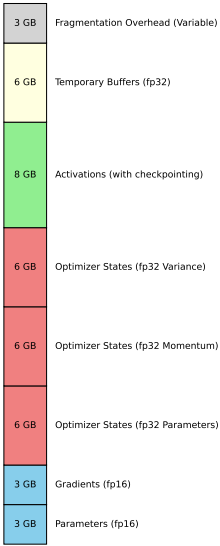
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Temporary Buffers:

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

Memory Fragmentation:

GPT-2 training Memory footprint



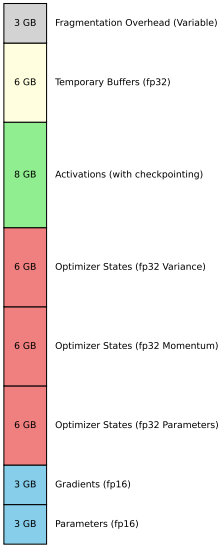
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

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.

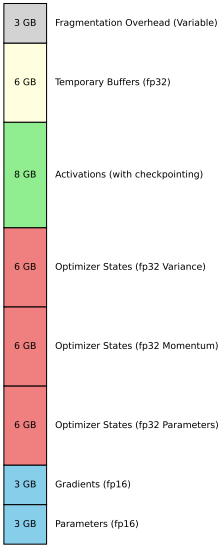
Temporary Buffers:

- 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.

Simple annotated MNIST exercise

A quick introduction to Optuna

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

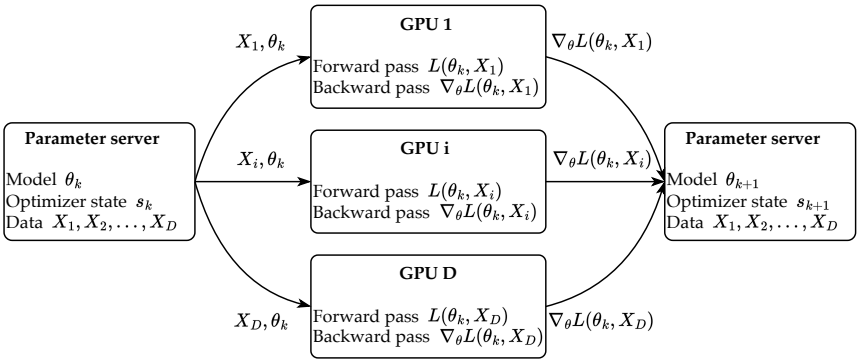
Data Parallel training

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) [1] 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	Model is replicated only once
Only supports single-node parallelism	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

[1] 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.

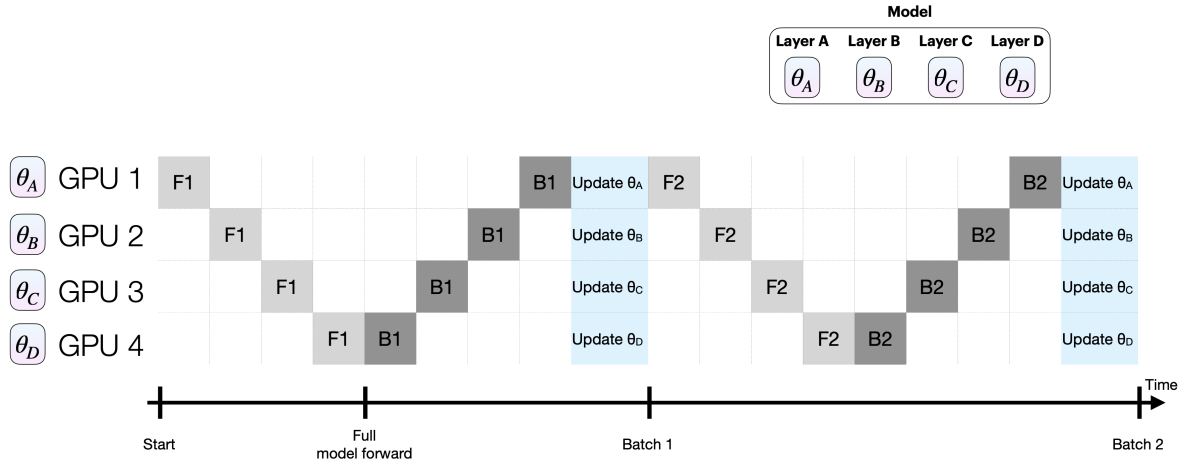
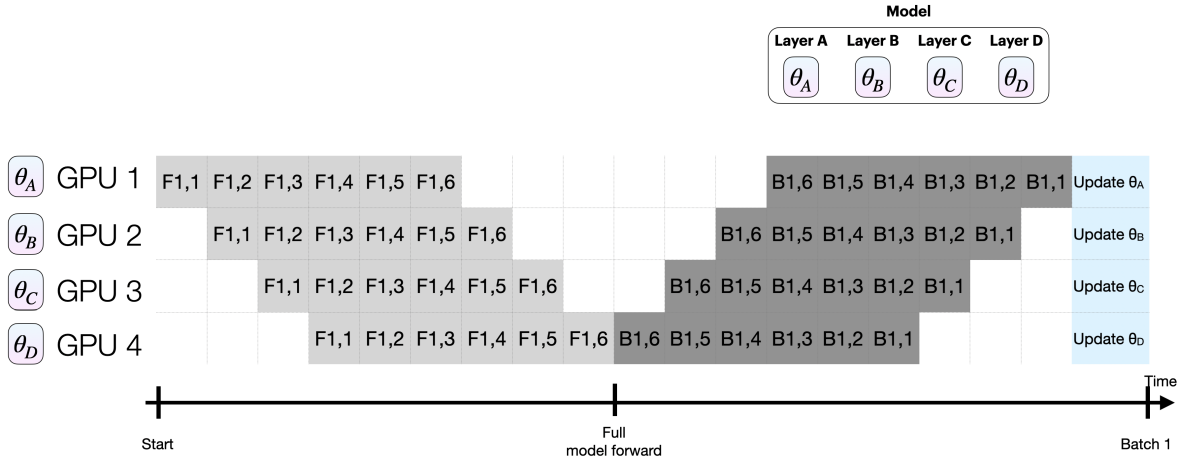


Figure 4: Model parallelism

Pipeline model parallelism (GPipe) [2]

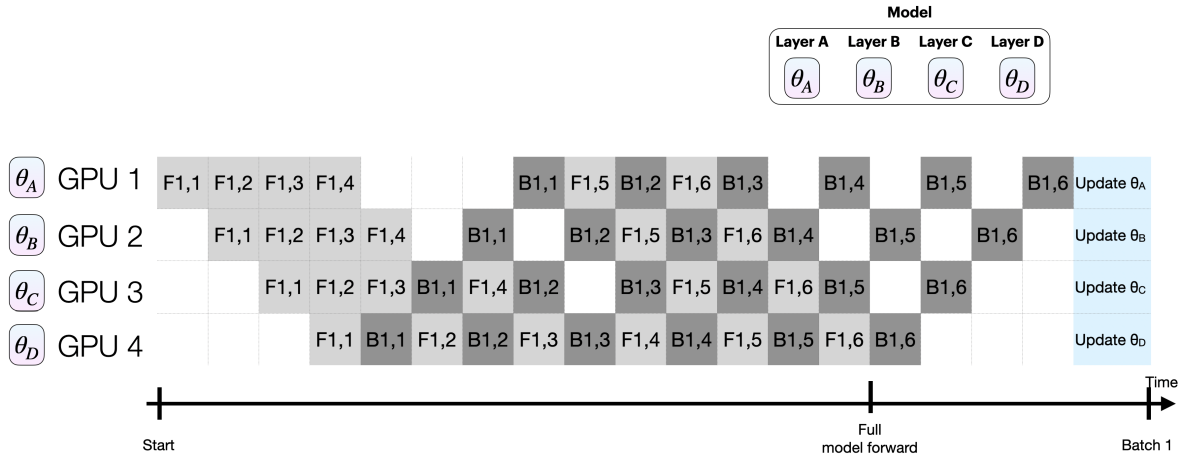
GPipe splits the model into stages, each processed sequentially. Micro-batches are passed through the pipeline, allowing for overlapping computation and communication:



[2] GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism

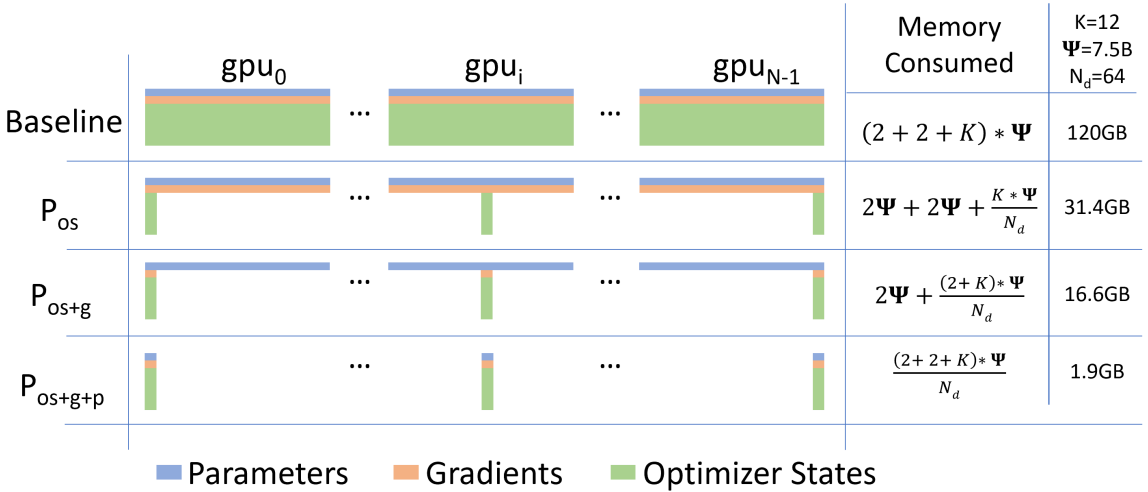
Pipeline model parallelism (PipeDream) [3]

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



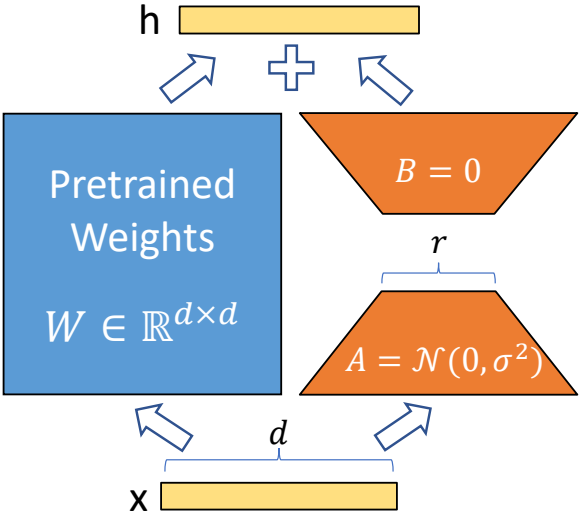
[3] PipeDream: Generalized Pipeline Parallelism for DNN Training

ZeRO [4]



[4] ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

LoRA [5]



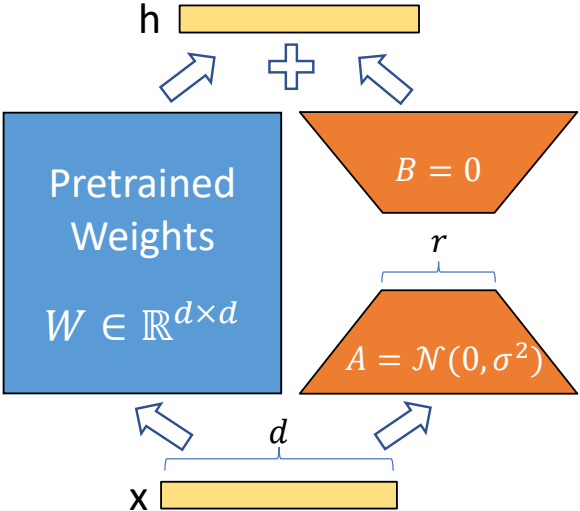
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

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

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 [5]



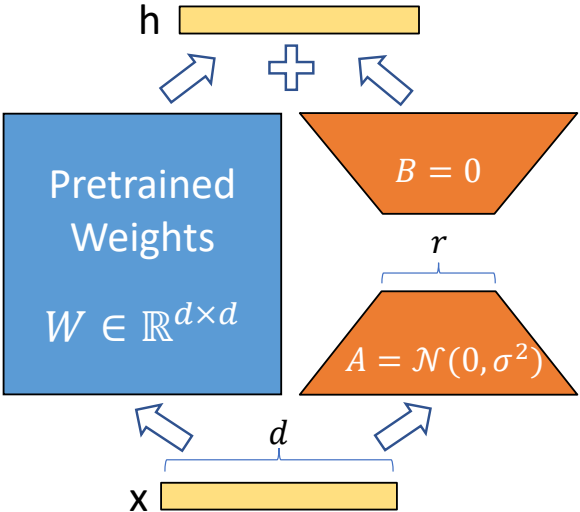
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

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

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 [5]



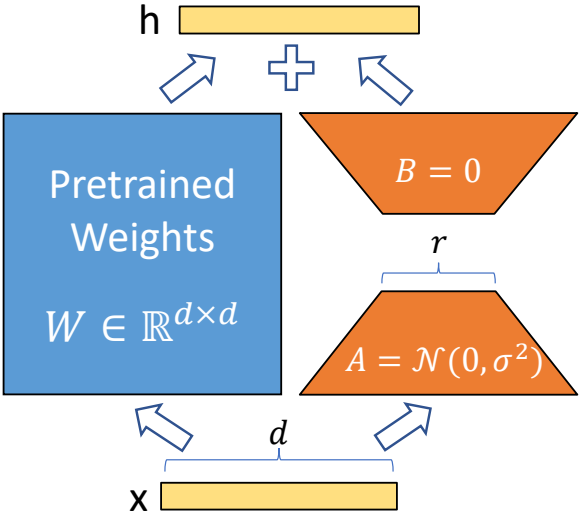
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

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

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 [5]



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

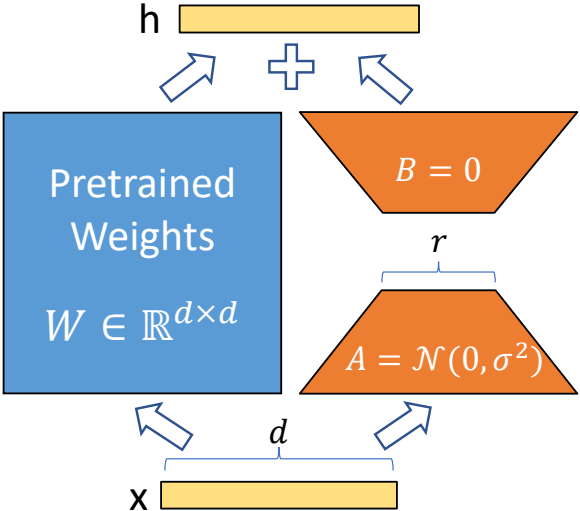
$$W_{\text{new}} = W + \Delta W$$

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

[5] LoRA: Low-Rank Adaptation of Large Language Models

LoRA [5]



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

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

[5] LoRA: Low-Rank Adaptation of Large Language Models

Feedforward Architecture

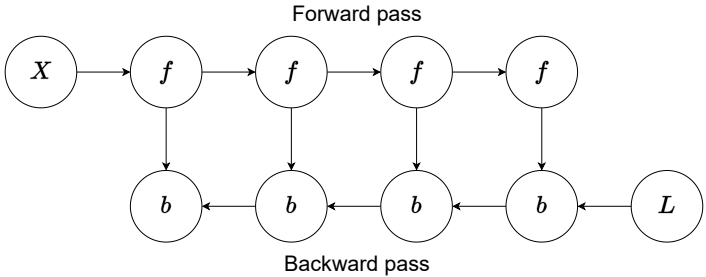


Figure 5: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

Feedforward Architecture

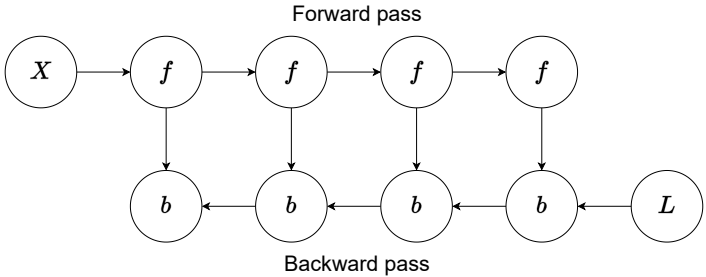


Figure 5: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

! Important

The results obtained for the f nodes are needed to compute the b nodes.

Vanilla backpropagation

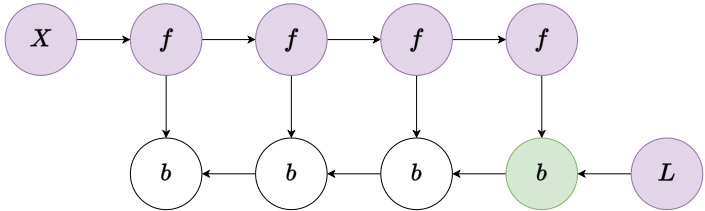


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Vanilla backpropagation

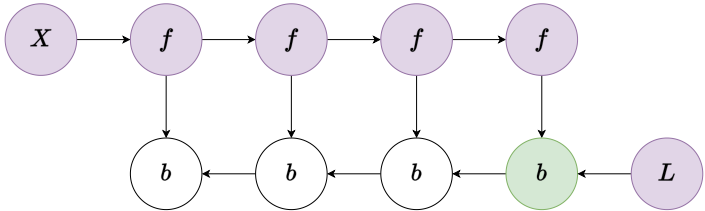


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

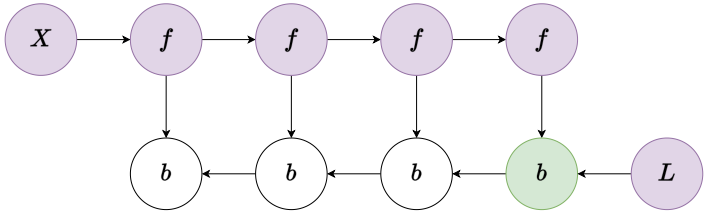


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

Vanilla backpropagation

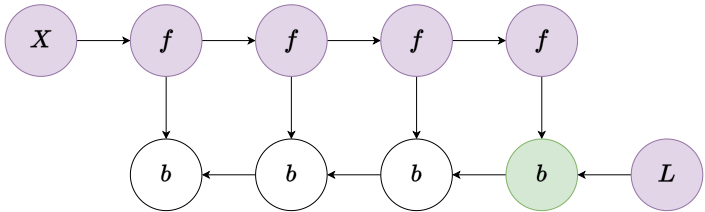


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

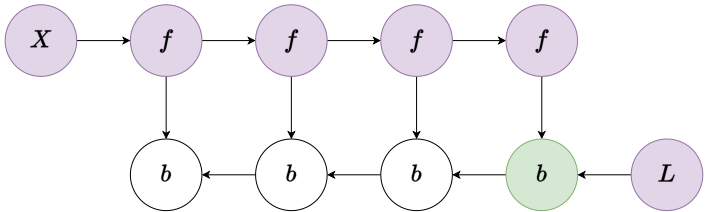


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

Vanilla backpropagation

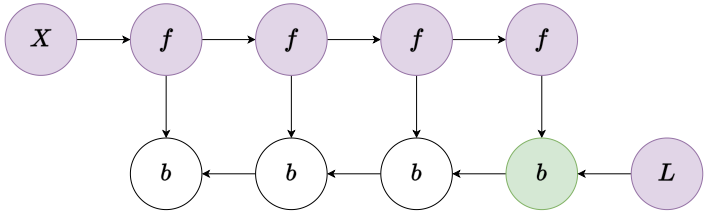


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- All activations f are kept in memory after the forward pass.

- Optimal in terms of computation: it only computes each node once.

- High memory usage. The memory usage grows linearly with the number of layers in the neural network.

Memory poor backpropagation

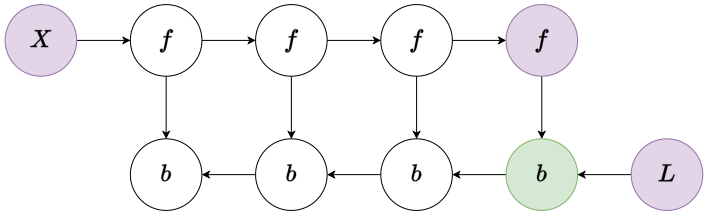


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Memory poor backpropagation

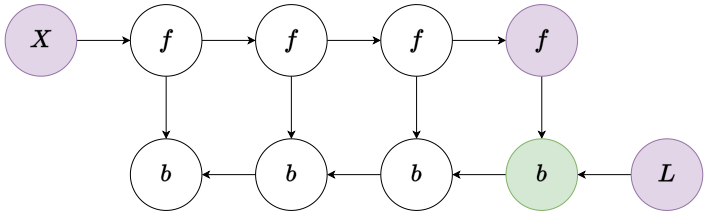


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

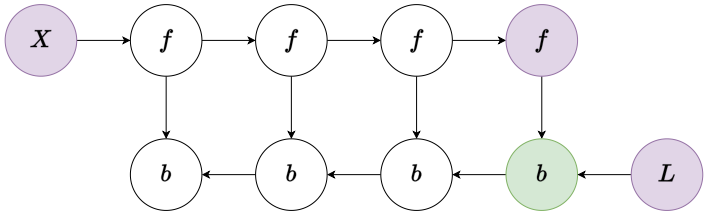


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

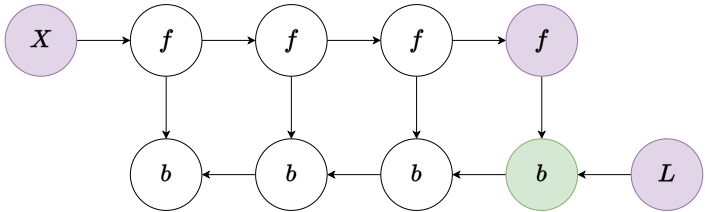


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

  - Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

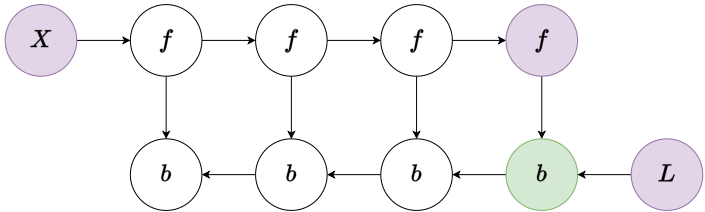


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

  - Optimal in terms of memory: there is no need to store all activations in memory.

Memory poor backpropagation

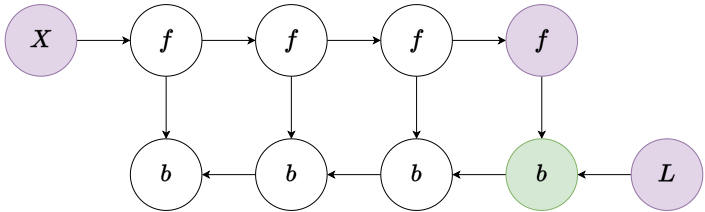


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

- Optimal in terms of memory: there is no need to store all activations in memory.

- Computationally inefficient. The number of node evaluations scales with n^2, whereas it vanilla backprop scaled as n: each of the n nodes is recomputed on the order of n times.

Checkpointed backpropagation

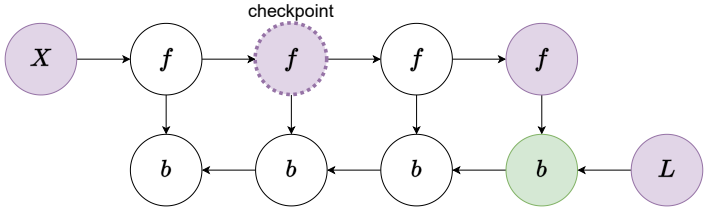


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Checkpointed backpropagation

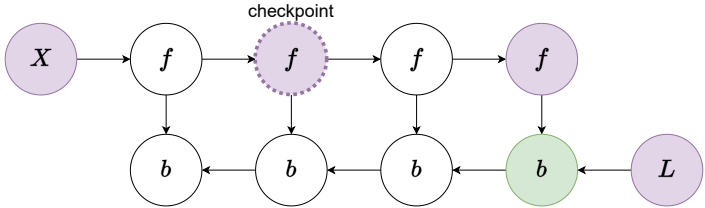


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

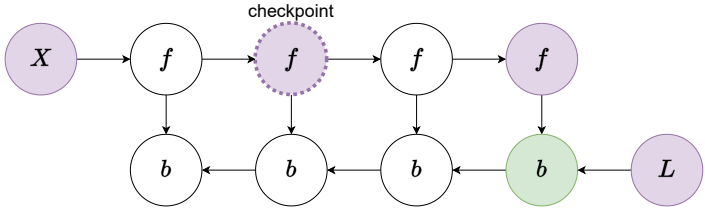


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

Checkpointed backpropagation

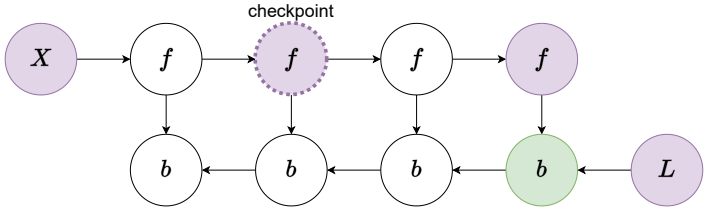


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

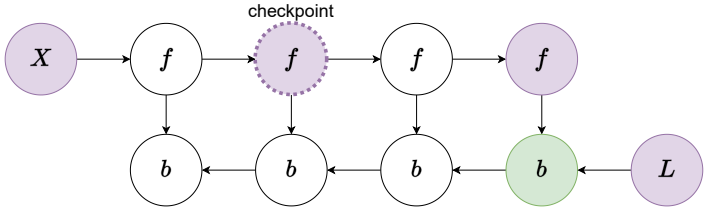


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

Checkpointed backpropagation

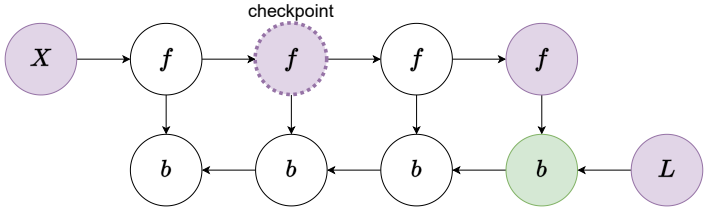


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Trade-off between the **vanilla** and **memory poor** approaches. The strategy is to mark a subset of the neural net activations as checkpoint nodes, that will be stored in memory.

- Faster recalculation of activations f. We only need to recompute the nodes between a b node and the last checkpoint preceding it when computing that b node during backprop.

- Memory consumption depends on the number of checkpoints. More effective then **vanilla** approach.

Gradient checkpointing visualization

The animated visualization of the above approaches

An example of using a gradient checkpointing

Quantization

Split the weight matrix into 2 well clustered factors [6]

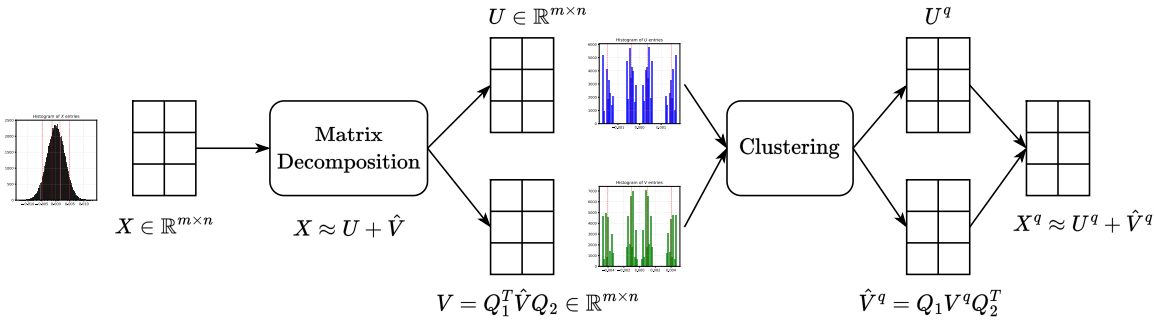


Figure 9: Scheme of post-training quantization approach.

[6] Quantization of Large Language Models with an Overdetermined Basis