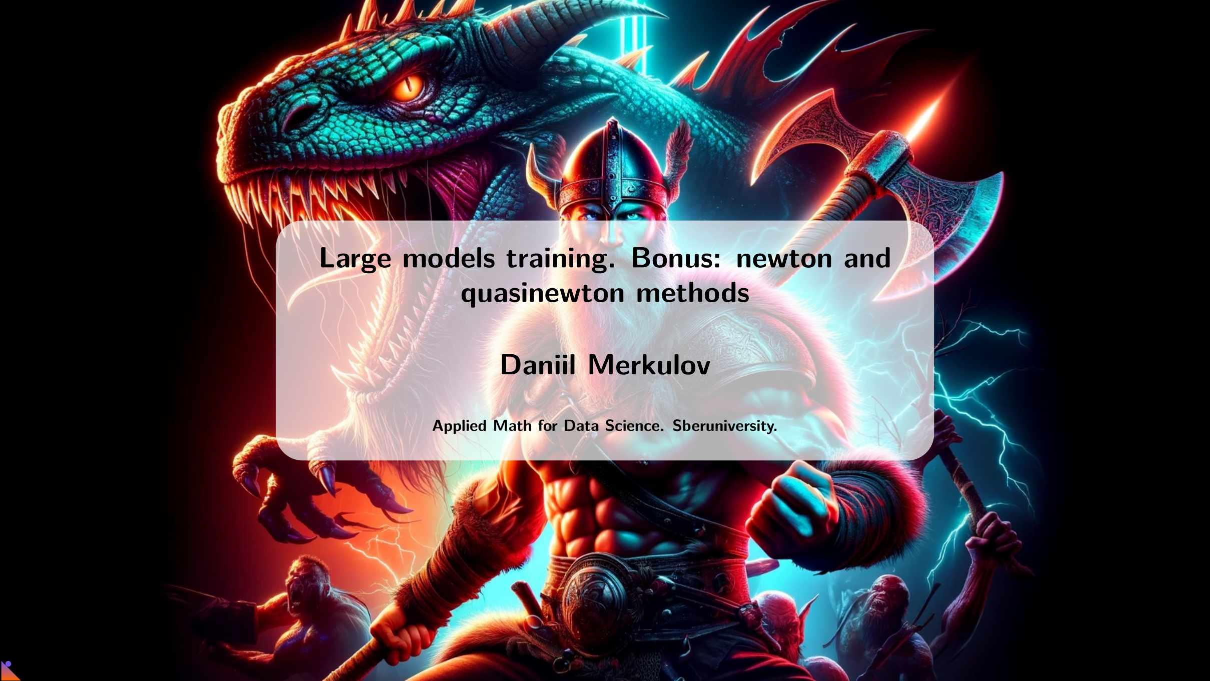


Large models training. Bonus: newton and quasinewton methods

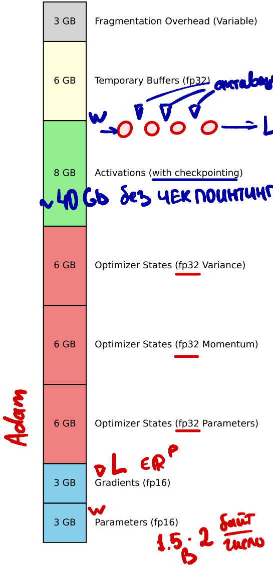
Daniil Merkulov

Applied Math for Data Science. Sberuniversity.

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.



3 GB Fragmentation Overhead (Variable)

6 GB Temporary Buffers (fp32)

активации

w → ∇ ∇ ∇ → L

8 GB Activations (with checkpointing)

~40 Gb без чекпоинтинга

6 GB Optimizer States (fp32 Variance)

6 GB Optimizer States (fp32 Momentum)

6 GB Optimizer States (fp32 Parameters)

Adam

$\nabla L \in \mathbb{R}^p$

3 GB Gradients (fp16)

w

3 GB Parameters (fp16)

$\underset{B}{1.5} \cdot 2 \frac{\text{байт}}{\text{число}}$

Model States:

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

$\min L(w)$

$w \in \mathbb{R}^p$

$p = 1.5B$

GPT-2

GPT-3 175 B

GPT-4 ~400 B (неточно)

Memory Requirements Example:

fp 64 - double precision

1 байт = = 8 бит

Residual Memory Consumption:

fp 32 - single precision

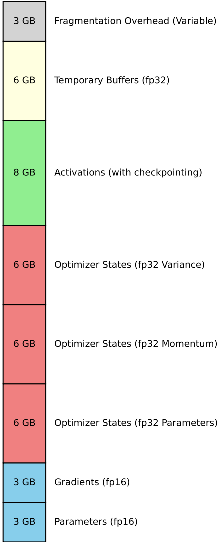
fp 16 - half precision

$32 \frac{\text{бит}}{\text{число}}$

$16 \frac{\text{бит}}{\text{число}}$

$2 \frac{\text{байта}}{\text{число}}$

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:

SGD+ Momentum, Rmsprop, NAG-GS, Lion.

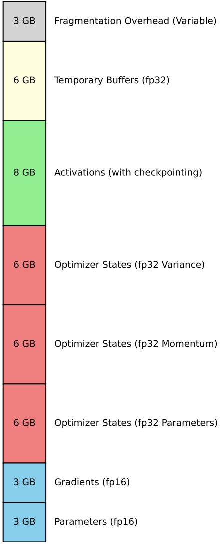
Adam 8bit

Bits and Bytes

Residual Memory Consumption:

1. CHECKpointing
2. Optimizer
3. квантизация

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:

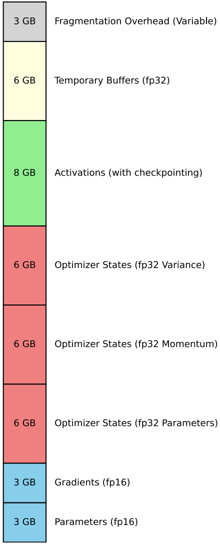
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

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:

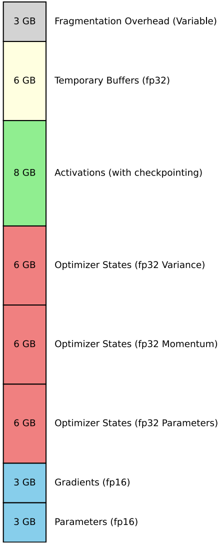
- Optimizer states (e.g., Adam) require memory for time-averaged momentum and gradient variance.
- Mixed-precision training (fp16/32) necessitates storing parameters and activations as fp16, but keeps fp32 copies for updates.

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.

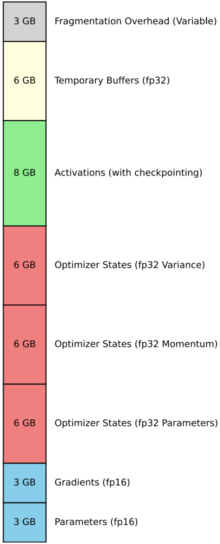
Memory Requirements Example:

- 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.

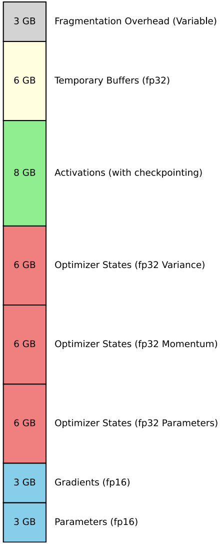
Memory Requirements Example:

- 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



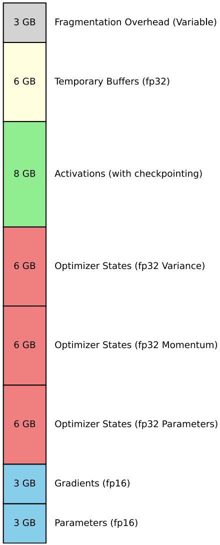
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

Temporary Buffers:

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

Memory Fragmentation:

GPT-2 training Memory footprint



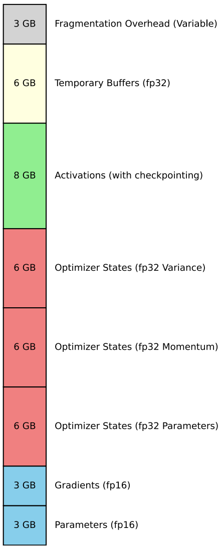
Example: 1.5B parameter GPT-2 model needs 3GB for weights in 16-bit precision but can't be trained on a 32GB GPU using Tensorflow or PyTorch. Major memory usage during training includes optimizer states, gradients, parameters, activations, temporary buffers, and fragmented memory.

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.

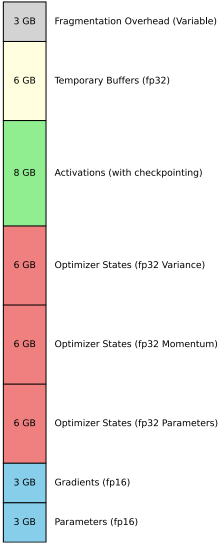
Temporary Buffers:

- 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:

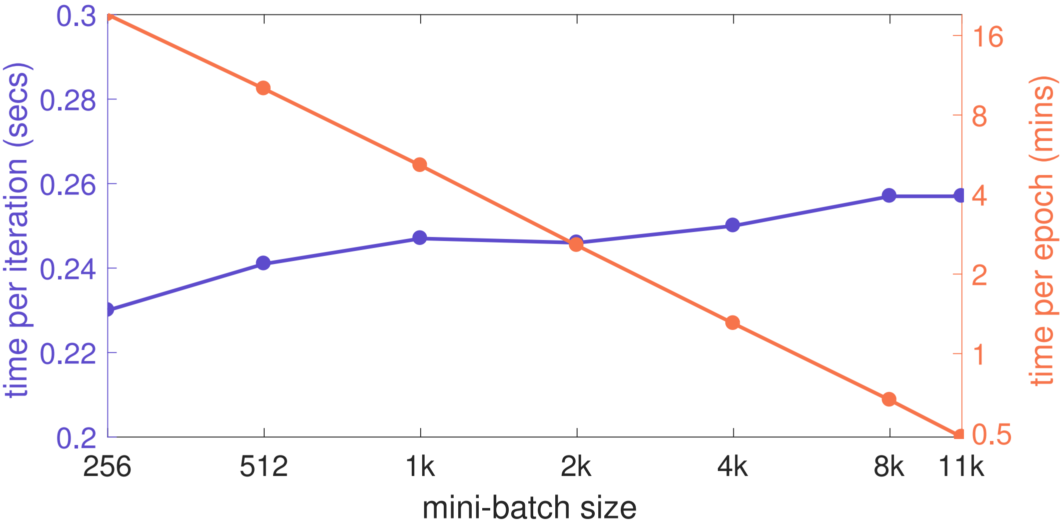
- Store intermediate results; e.g., gradient all-reduce operations fuse gradients into a single buffer.
- For large models, temporary buffers can consume substantial memory (e.g., 6GB for 1.5B parameter model with fp32 buffer).

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.

Large batch training

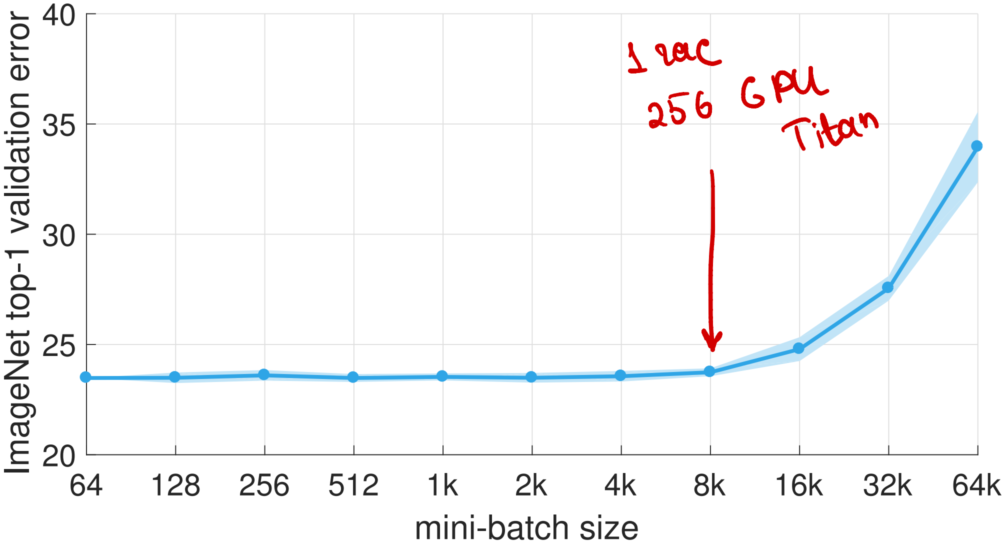
Large batch training [1]



[1] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Large batch training [2]

SGD → GD

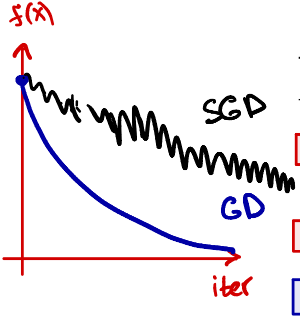


1 час
256 GPU
Titan

рост ошибки

[2] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Large batch training [3]



learning rate ↓ ↓ чем меньше, тем лучше

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

лучший lr на малом батче

увеличили lr в 32 раза

не было

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

[3] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Linear and square root scaling rules

When training with large batches, the learning rate must be adjusted to maintain convergence speed and stability. The **linear scaling rule**[4] suggests multiplying the learning rate by the same factor as the increase in batch size:

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

- SGD, SGD+Momentum, NAG

The **square root scaling rule**[5] proposes scaling the learning rate with the square root of the batch size increase:

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

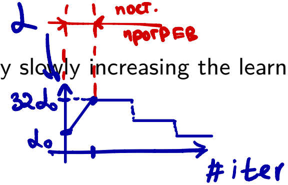
- Adam, Rmsprop, NAGGS, ...

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

[4] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

[5] Learning Rates as a Function of Batch Size: A Random Matrix Theory Approach to Neural Network Training

Gradual warmup [6] постепенный прогрев



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

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

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

в начале делаем несколько (5-10%) итер. с фикс. маленьким lr а затем резко увеличиваем

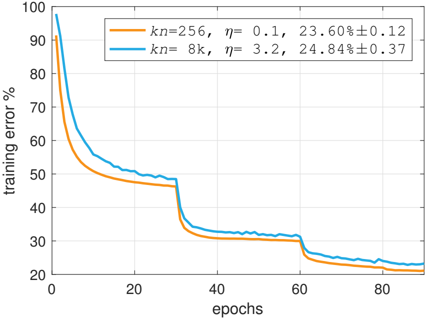


Figure 1: no warmup

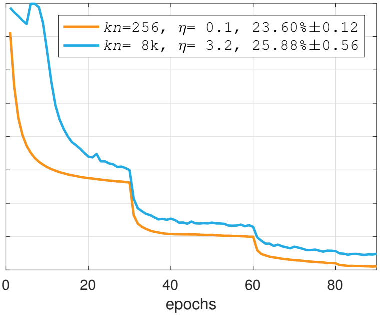


Figure 2: constant warmup

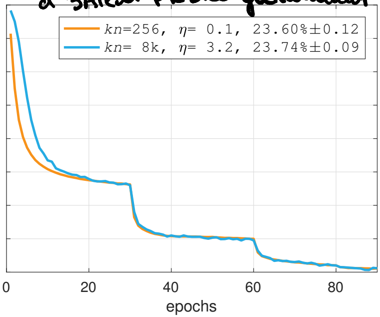


Figure 3: gradual warmup

[6] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour

Gradient accumulation

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

Without gradient accumulation

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

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

FORWARD

BACKWARD

Gradient accumulation

1. вычисляются $\frac{\partial L}{\partial W}$

2. в поле переменных $W.grad \mathrel{+}= \frac{\partial L}{\partial W}$

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

Without gradient accumulation

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

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

With gradient accumulation

```
for i, (inputs, targets) in enumerate(data):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    if (i+1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

гиперпар.

легко увеличить эффективный размер батча

GPU
b = 1

accumulation steps = 10

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

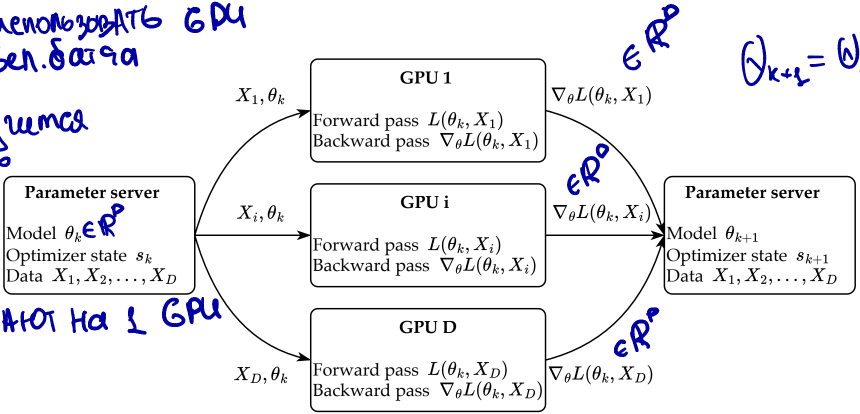
$$g_k = \text{mean}(\nabla_\theta L_1, \dots \nabla_\theta L_D)$$

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:

+ можно использовать GPU для увел. батча

$$\theta_{k+1} = \theta_k - \alpha \cdot g_k$$

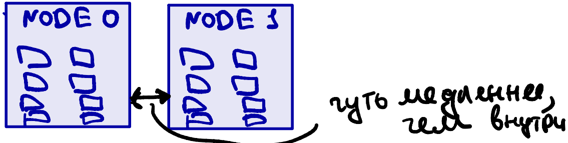
- не получится обучать модели, которые не влезают на 1 GPU



Parameter server
Model $\theta_k \in \mathbb{R}^p$
Optimizer state s_k
Data $X_1, X_2, \dots, X_D$
X_1, θ_k
X_i, θ_k
X_D, θ_k
GPU 1
Forward pass $L(\theta_k, X_1)$
Backward pass $\nabla_\theta L(\theta_k, X_1)$
GPU i
Forward pass $L(\theta_k, X_i)$
Backward pass $\nabla_\theta L(\theta_k, X_i)$
GPU D
Forward pass $L(\theta_k, X_D)$
Backward pass $\nabla_\theta L(\theta_k, X_D)$
$\in \mathbb{R}^p$
$\nabla_\theta L(\theta_k, X_1)$
$\in \mathbb{R}^p$
$\nabla_\theta L(\theta_k, X_i)$
$\in \mathbb{R}^p$
$\nabla_\theta L(\theta_k, X_D)$
Parameter server
Model θ_{k+1}
Optimizer state s_{k+1}
Data $X_1, X_2, \dots, X_D$

Distributed Data Parallel training

DDP



Distributed Data Parallel (DDP) [7] 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

[7] Getting Started with Distributed Data Parallel

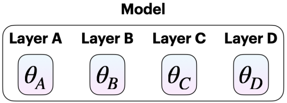
Naive model parallelism

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.

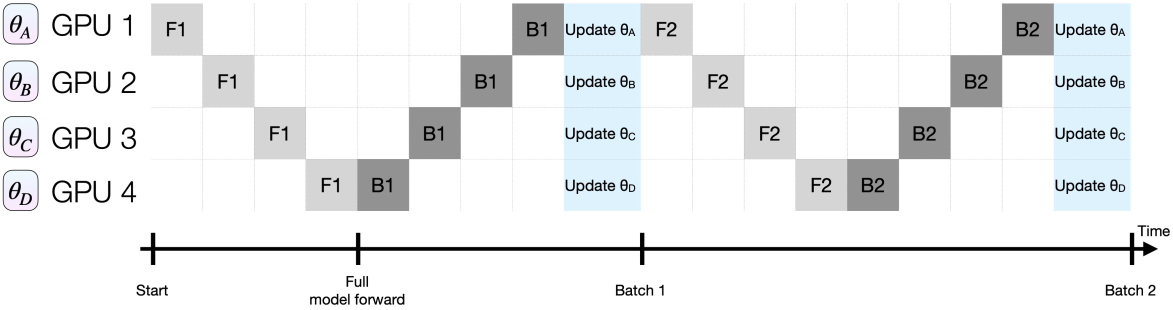
+ Наконец-то можно обучать модели, которые не помещаются на 1 GPU

— слишком много простоя GPU

Вход ⇒



Выход ⇒



θ_A GPU 1	F1							B1	Update θ_A	F2							B2	Update θ_A
θ_B GPU 2		F1					B1		Update θ_B		F2					B2		Update θ_B
θ_C GPU 3			F1			B1			Update θ_C			F2			B2			Update θ_C
θ_D GPU 4				F1	B1				Update θ_D				F2	B2				Update θ_D

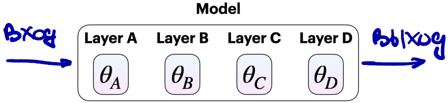
Figure 5: Model parallelism

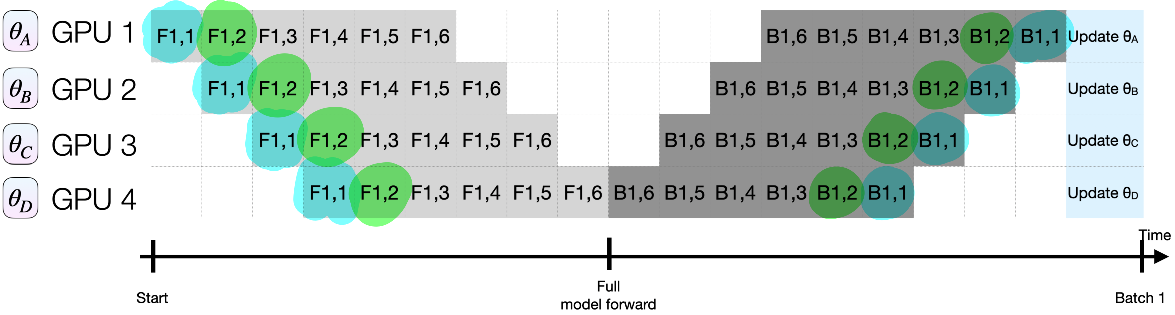
Pipeline model parallelism (GPipe) [8]

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

пусть $b = 120$

$b_{micro} = 20$



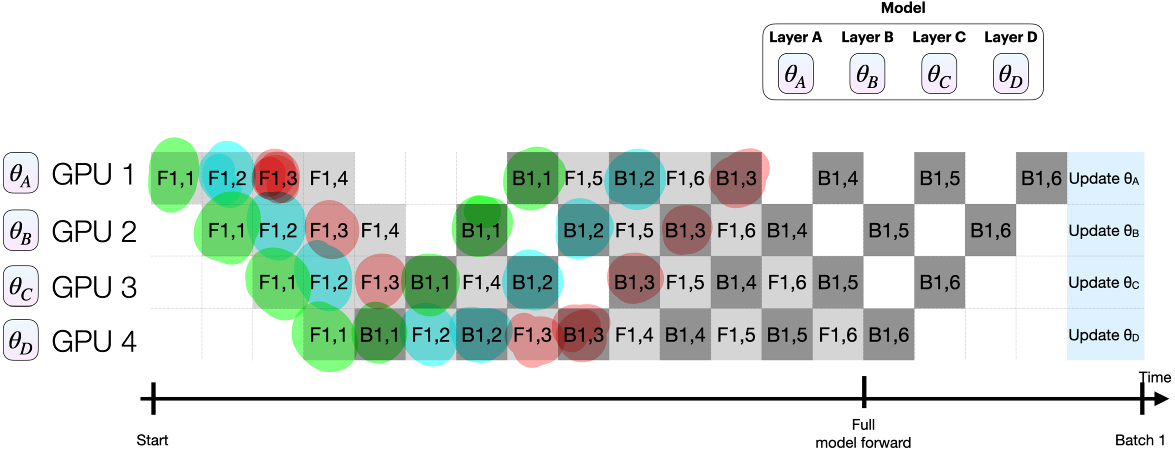


θ_A GPU 1	F1,1	F1,2	F1,3	F1,4	F1,5	F1,6							B1,6	B1,5	B1,4	B1,3	B1,2	B1,1	Update θA
θ_B GPU 2		F1,1	F1,2	F1,3	F1,4	F1,5	F1,6					B1,6	B1,5	B1,4	B1,3	B1,2	B1,1		Update θB
θ_C GPU 3			F1,1	F1,2	F1,3	F1,4	F1,5	F1,6			B1,6	B1,5	B1,4	B1,3	B1,2	B1,1			Update θC
θ_D GPU 4				F1,1	F1,2	F1,3	F1,4	F1,5	F1,6	B1,6	B1,5	B1,4	B1,3	B1,2	B1,1				Update θD

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

Pipeline model parallelism (PipeDream) [9]

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



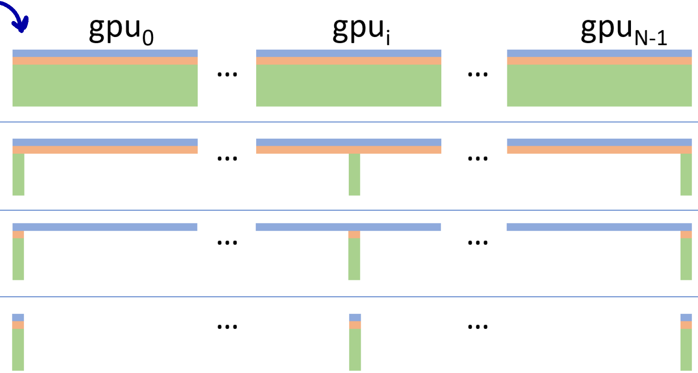
θ_A GPU 1	F1,1	F1,2	F1,3	F1,4			B1,1	F1,5	B1,2	F1,6	B1,3		B1,4		B1,5		B1,6	Update θ_A
θ_B GPU 2		F1,1	F1,2	F1,3	F1,4		B1,1		B1,2	F1,5	B1,3	F1,6	B1,4		B1,5		B1,6	Update θ_B
θ_C GPU 3			F1,1	F1,2	F1,3	B1,1	F1,4	B1,2		B1,3	F1,5	B1,4	F1,6	B1,5		B1,6		Update θ_C
θ_D GPU 4				F1,1	B1,1	F1,2	B1,2	F1,3	B1,3	F1,4	B1,4	F1,5	B1,5	F1,6	B1,6			Update θ_D

[9] PipeDream: Generalized Pipeline Parallelism for DNN Training

ZeRO [10]

Deep Speed

DP



	Memory Consumed	K=12
		Ψ=7.5B
		N$_d$=64

Baseline	$(2+2+K)*\Psi$	120GB
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

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

Automatic Mixed Precision training

AMP

fp32 vs fp8

разница в байтах в 4 раз

Two copies of the models needed to be stored - fp32 and fp16 (fp8). Rewrite the computational graph with respect to the following idea:

Numerically-Safe plus Performance Critical (always in fp16/fp8)

Convolution & Matmul

Numerically-Neutral (Context-Dependent)

Max, min

Numerically-Safe (Conditional, Context-Dependent)

Activations

Numerically-Dangerous (Always in fp32)

Exp, Log, Pow, Softmax, Reduction Sum, Mean

In `accelerate`:

```
torch.cuda.amp.autocast(dtype=torch.float16)(model_forward_func)
```

or

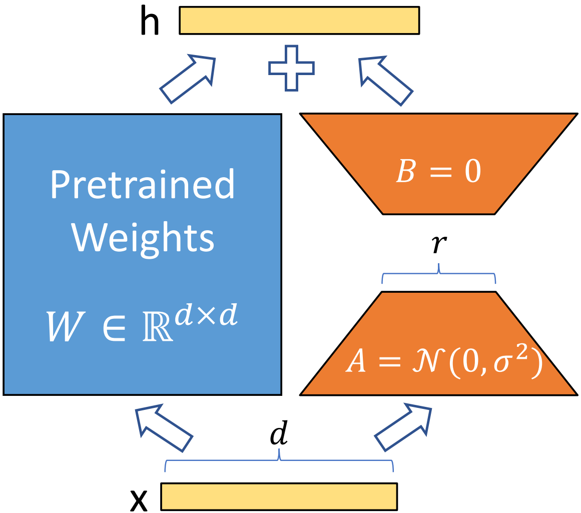
```
torch.autocast(device_type=self.device.type, dtype=torch.bfloat16)(model_forward_func)
```

увеличиваем потребление памяти во время обучения

экономим на инференсе

LoRA[11]

задача: дообучить LLM на конкретную задачу



h
Pretrained Weights
$W \in \mathbb{R}^{d\times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

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

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

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

$$\min_{w \in \mathbb{R}^P} L(w) =$$

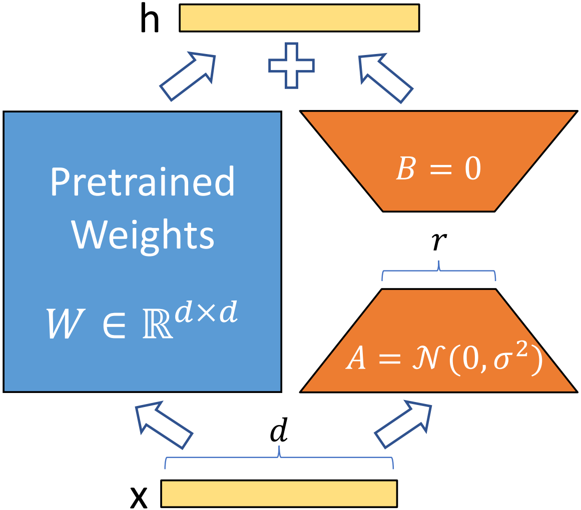
$$= \min_{\Delta W \in \mathbb{R}^K} L(W + \Delta W)$$

$$\underline{\underline{K << P}} \quad // \ 2rd$$

$$\underset{d\times d}{\Delta W} = \underset{d\times r}{A}\,\underset{r\times d}{B^T}$$

а $\to B_{dxr}$

LoRA [11]


h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

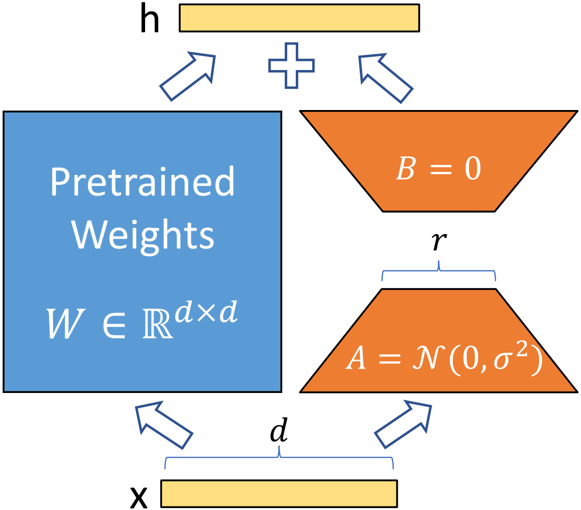
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

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

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 [11]


h
Pretrained Weights
$W \in \mathbb{R}^{d\times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

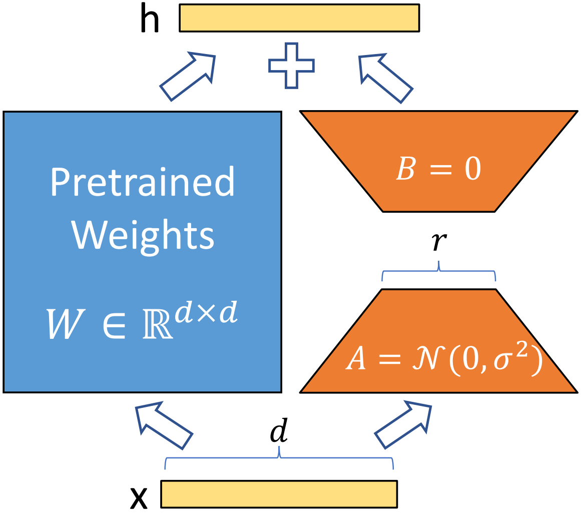
LoRA reduces the number of parameters by approximating weight matrices with low-rank factorization:

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

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 [11]


h
Pretrained
Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

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

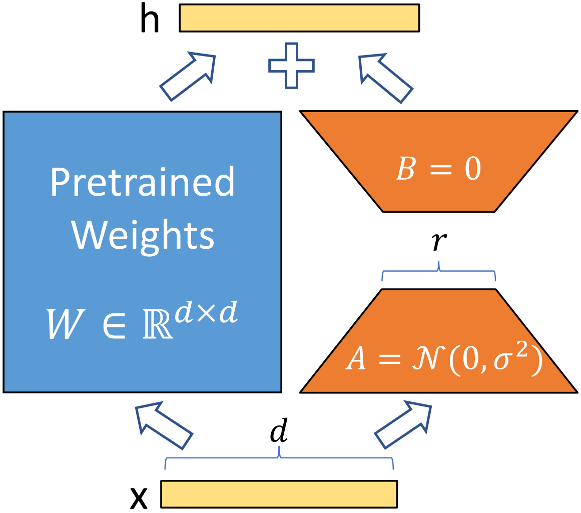
$$W_{\text{new}} = W + \Delta W$$

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

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

LoRA [11]


h
Pretrained Weights
$W \in \mathbb{R}^{d\times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

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

$$\boxed{h = W_{\text{new}}x} = \overset{O(d^2)}{\underset{d\times d \; d\times 1}{Wx}} + \Delta W x = Wx + \overset{dr}{\underset{d\times r \; r\times d \; d\times 1}{AB^T x}}$$

dr

O(2dr)

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

Feedforward Architecture

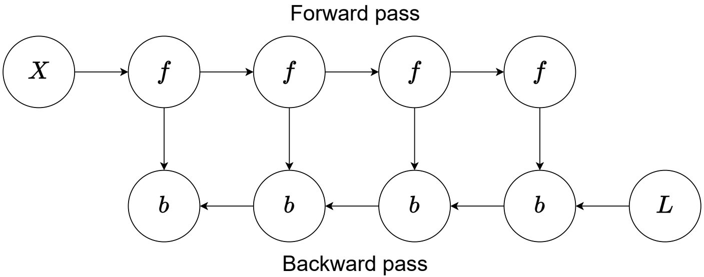


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

Feedforward Architecture

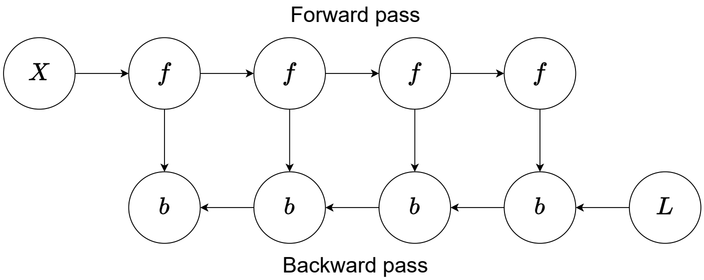


Figure 6: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The activations marked with an f. The gradient of the loss with respect to the activations and parameters marked with b.

Important

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

Vanilla backpropagation

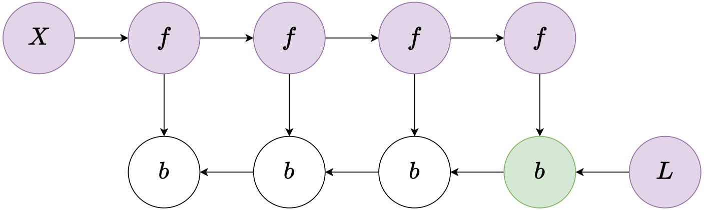


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Vanilla backpropagation

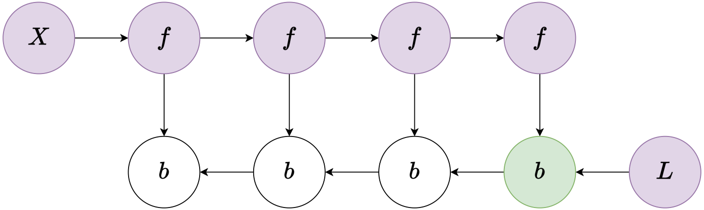


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

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

Vanilla backpropagation

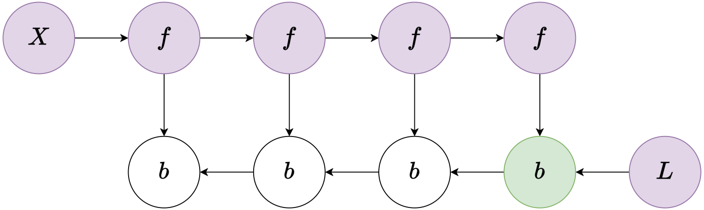


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

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

Vanilla backpropagation

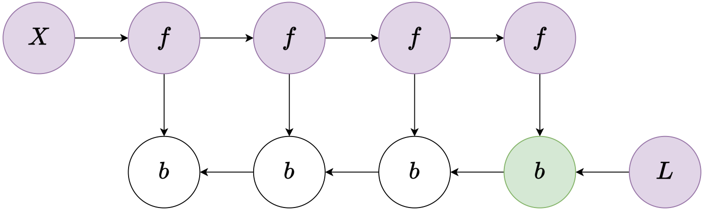


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

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

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

Vanilla backpropagation

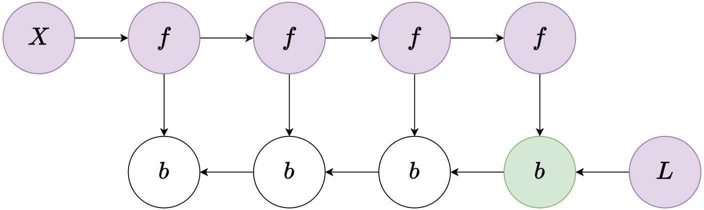


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

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

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

Vanilla backpropagation

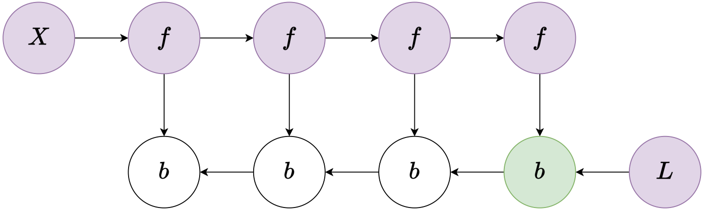


Figure 7: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

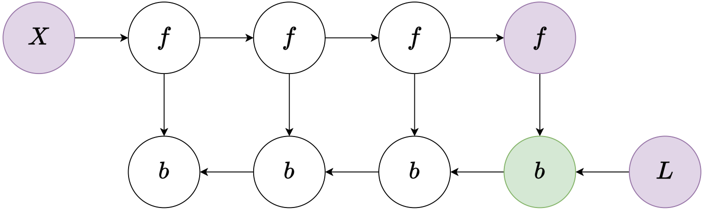


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Memory poor backpropagation

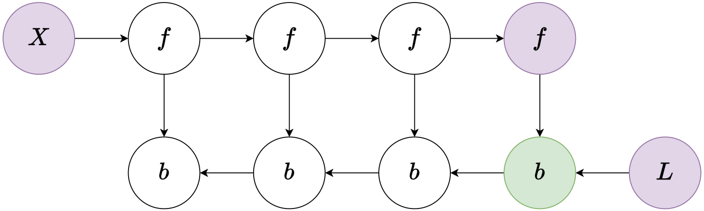


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

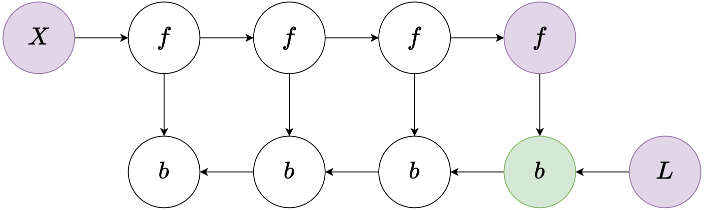


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- Each activation f is recalculated as needed.

Memory poor backpropagation

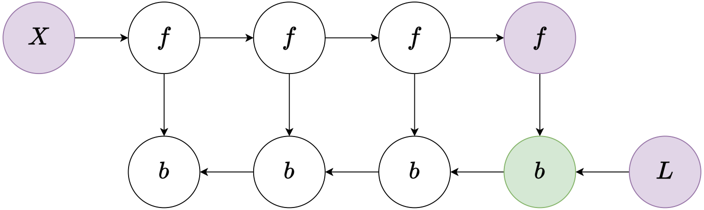


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

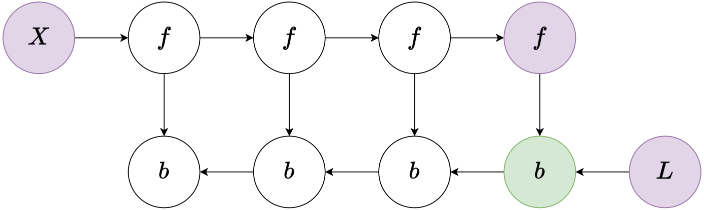


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

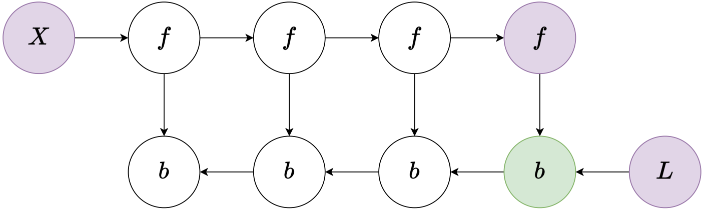


Figure 8: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

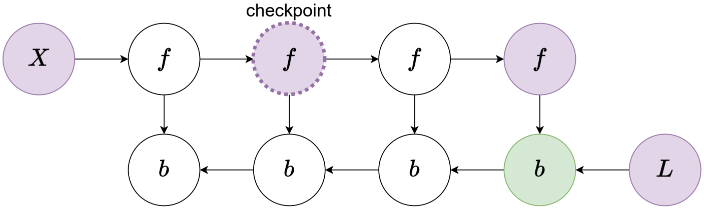


Figure 9: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

Checkpointed backpropagation

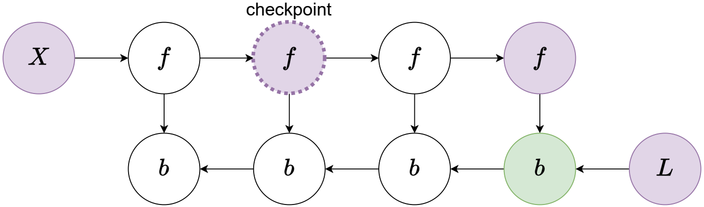


Figure 9: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

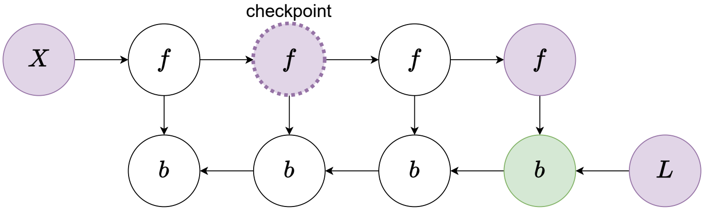


Figure 9: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

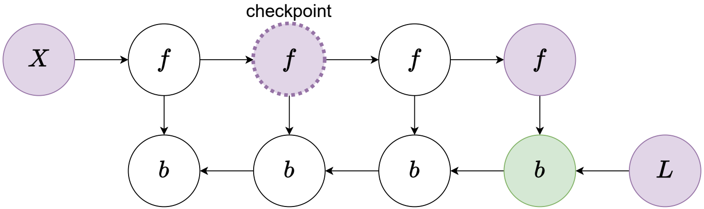


Figure 9: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

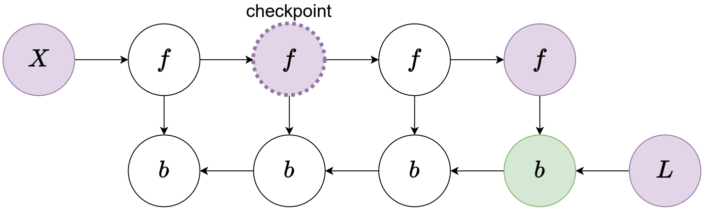


Figure 9: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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

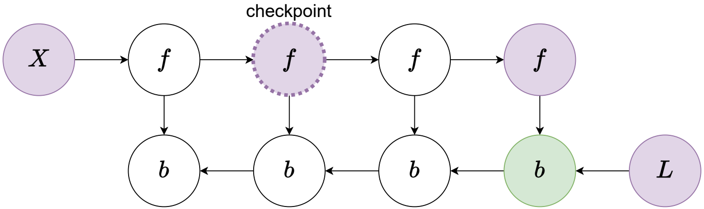


Figure 9: Computation graph for obtaining gradients for a simple feed-forward neural network with n layers. The purple color indicates nodes that are stored in memory.

- 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.

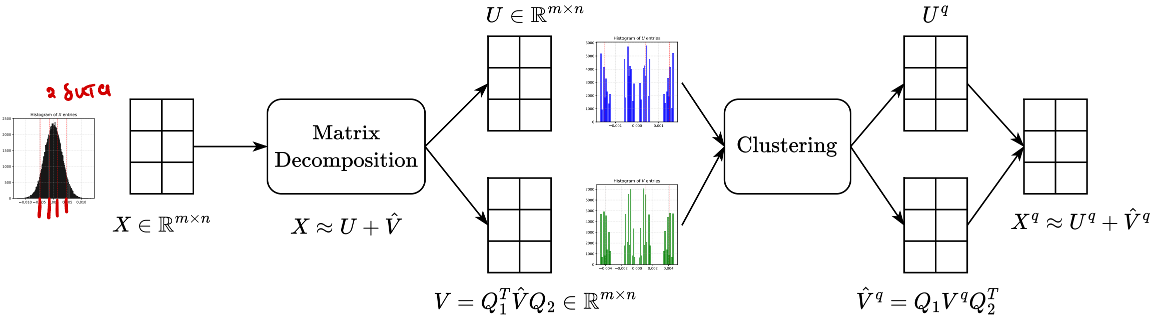
Gradient checkpointing visualization

The animated visualization of the above approaches

An example of using a gradient checkpointing

Quantization

Split the weight matrix into 2 well clustered factors [12]



2 бита

$X \in \mathbb{R}^{m \times n}$

Matrix Decomposition

$X \approx U + \hat{V}$

$U \in \mathbb{R}^{m \times n}$

$V = Q_1^T \hat{V} Q_2 \in \mathbb{R}^{m \times n}$

Clustering

U^q

$\hat{V}^q = Q_1 V^q Q_2^T$

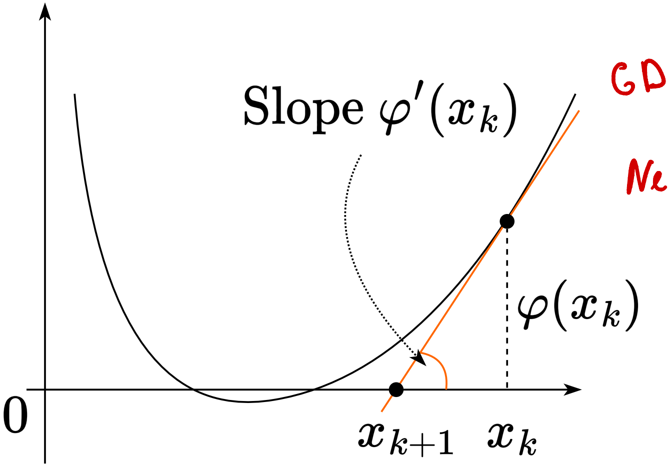
$X^q \approx U^q + \hat{V}^q$

Figure 10: Scheme of post-training quantization approach.

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

Newton method

Idea of Newton method of root finding

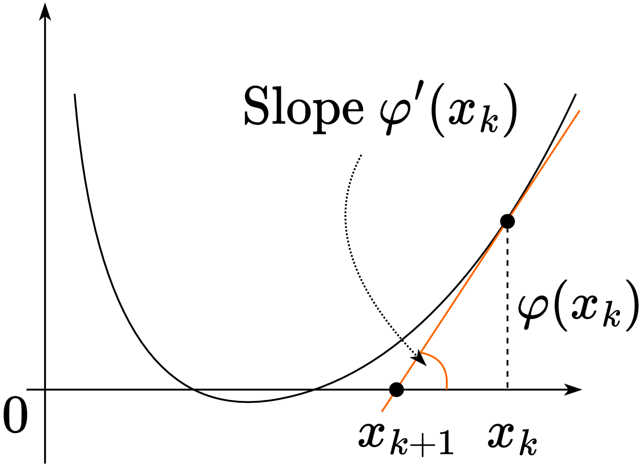


Consider the function $\varphi(x) : \mathbb{R} \to \mathbb{R}$.

GD: $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$

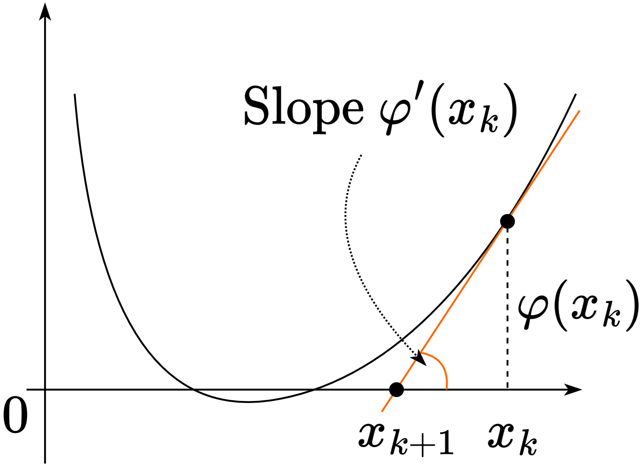
Newton: $x_k = x_k - \alpha_k [\nabla^2 f(x_k)]^{-1} \nabla f(x_k)$

Idea of Newton method of root finding



Consider the function $\varphi(x) : \mathbb{R} \to \mathbb{R}$.
The whole idea came from building a linear approximation at the point x_k and find its root, which will be the new iteration point:

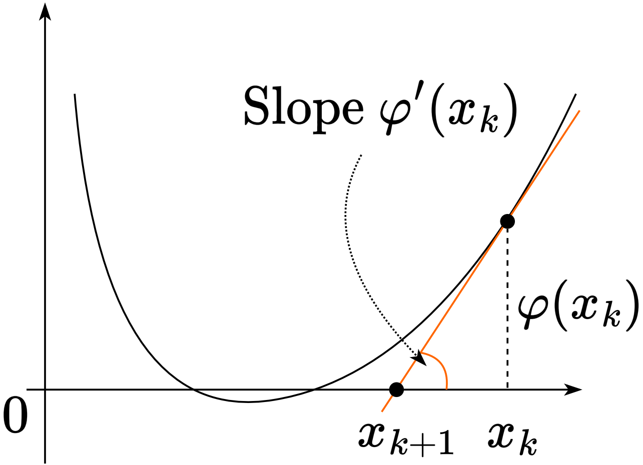
Idea of Newton method of root finding



Consider the function $\varphi(x) : \mathbb{R} \to \mathbb{R}$.
The whole idea came from building a linear approximation at the point x_k and find its root, which will be the new iteration point:

$$\varphi'(x_k) = \frac{\varphi(x_k)}{x_{k+1} - x_k}$$

Idea of Newton method of root finding

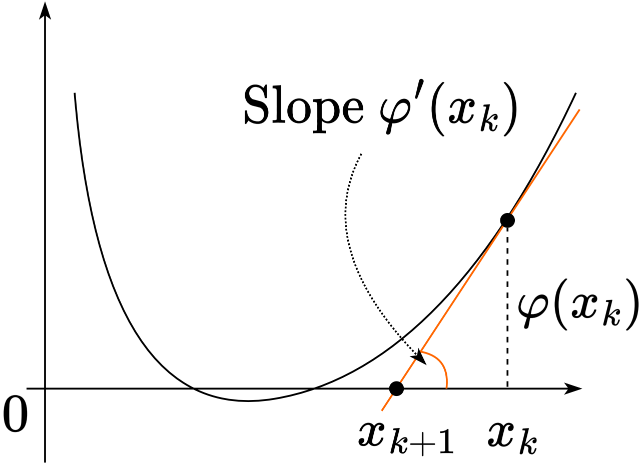


Consider the function $\varphi(x) : \mathbb{R} \to \mathbb{R}$.
The whole idea came from building a linear approximation at the point x_k and find its root, which will be the new iteration point:

$$\varphi'(x_k) = \frac{\varphi(x_k)}{x_{k+1} - x_k}$$

We get an iterative scheme:

Idea of Newton method of root finding



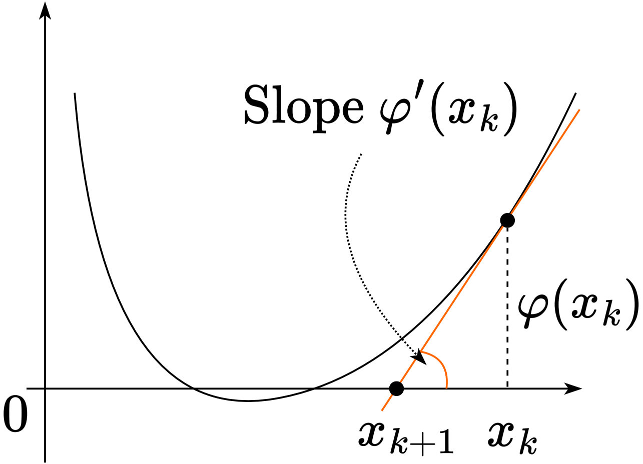
Consider the function $\varphi(x) : \mathbb{R} \to \mathbb{R}$.
The whole idea came from building a linear approximation at the point x_k and find its root, which will be the new iteration point:

$$\varphi'(x_k) = \frac{\varphi(x_k)}{x_{k+1} - x_k}$$

We get an iterative scheme:

$$x_{k+1} = x_k - \frac{\varphi(x_k)}{\varphi'(x_k)}.$$

Idea of Newton method of root finding



Consider the function $\varphi(x) : \mathbb{R} \to \mathbb{R}$.
The whole idea came from building a linear approximation at the point x_k and find its root, which will be the new iteration point:

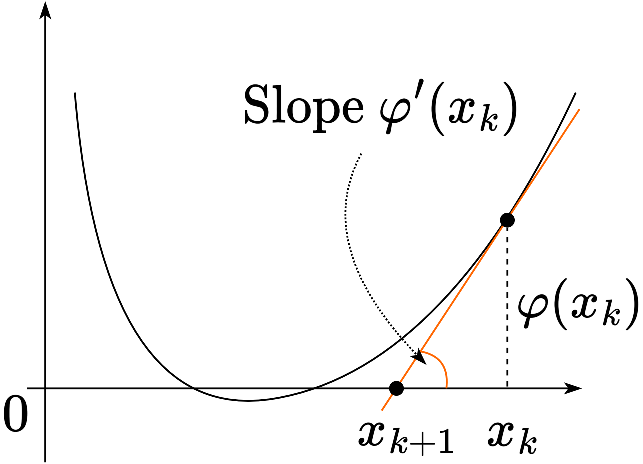
$$\varphi'(x_k) = \frac{\varphi(x_k)}{x_{k+1} - x_k}$$

We get an iterative scheme:

$$x_{k+1} = x_k - \frac{\varphi(x_k)}{\varphi'(x_k)}.$$

Which will become a Newton optimization method in case $f'(x) = \varphi(x)$[a]:

Idea of Newton method of root finding



Consider the function $\varphi(x) : \mathbb{R} \to \mathbb{R}$.
The whole idea came from building a linear approximation at the point x_k and find its root, which will be the new iteration point:

$$\varphi'(x_k) = \frac{\varphi(x_k)}{x_{k+1} - x_k}$$

We get an iterative scheme:

$$x_{k+1} = x_k - \frac{\varphi(x_k)}{\varphi'(x_k)}.$$

Which will become a Newton optimization method in case $f'(x) = \varphi(x)$[a]:

$$\boxed{x_{k+1} = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k)}$$

[a] Literally we aim to solve the problem of finding stationary points $\nabla f(x) = 0$

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x - x_k), x - x_k\rangle.$$

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x - x_k), x - x_k\rangle.$$

The idea of the method is to find the point x_{k+1}, that minimizes the function $f^{II}_{x_k}(x)$, i.e. $\nabla f^{II}_{x_k}(x_{k+1}) = 0$.

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x-x_k), x-x_k \rangle.$$

The idea of the method is to find the point x_{k+1}, that minimizes the function $f^{II}_{x_k}(x)$, i.e. $\nabla f^{II}_{x_k}(x_{k+1}) = 0$.

$$\nabla f^{II}_{x_k}(x_{k+1}) = \nabla f(x_k) + \nabla^2 f(x_k)(x_{k+1} - x_k) = 0$$

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x-x_k), x-x_k \rangle.$$

The idea of the method is to find the point x_{k+1}, that minimizes the function $f^{II}_{x_k}(x)$, i.e. $\nabla f^{II}_{x_k}(x_{k+1}) = 0$.

$$\begin{aligned} \nabla f^{II}_{x_k}(x_{k+1}) &= \nabla f(x_k) + \nabla^2 f(x_k)(x_{k+1} - x_k) = 0 \\ \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\nabla f(x_k) \end{aligned}$$

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x-x_k), x-x_k \rangle.$$

The idea of the method is to find the point x_{k+1}, that minimizes the function $f^{II}_{x_k}(x)$, i.e. $\nabla f^{II}_{x_k}(x_{k+1}) = 0$.

$$\begin{aligned} \nabla f^{II}_{x_k}(x_{k+1}) &= \nabla f(x_k) + \nabla^2 f(x_k)(x_{k+1} - x_k) = 0 \\ \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\nabla f(x_k) \\ \left[\nabla^2 f(x_k)\right]^{-1} \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \end{aligned}$$

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x-x_k), x-x_k \rangle.$$

The idea of the method is to find the point x_{k+1}, that minimizes the function $f^{II}_{x_k}(x)$, i.e. $\nabla f^{II}_{x_k}(x_{k+1}) = 0$.

$$\begin{aligned}
\nabla f^{II}_{x_k}(x_{k+1}) &= \nabla f(x_k) + \nabla^2 f(x_k)(x_{k+1} - x_k) = 0 \\
\nabla^2 f(x_k)(x_{k+1} - x_k) &= -\nabla f(x_k) \\
\left[\nabla^2 f(x_k)\right]^{-1} \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \\
x_{k+1} &= x_k -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k).
\end{aligned}$$

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x-x_k), x-x_k \rangle.$$

The idea of the method is to find the point x_{k+1}, that minimizes the function $f^{II}_{x_k}(x)$, i.e. $\nabla f^{II}_{x_k}(x_{k+1}) = 0$.

$$\begin{aligned}
\nabla f^{II}_{x_k}(x_{k+1}) &= \nabla f(x_k) + \nabla^2 f(x_k)(x_{k+1} - x_k) = 0 \\
\nabla^2 f(x_k)(x_{k+1} - x_k) &= -\nabla f(x_k) \\
\left[\nabla^2 f(x_k)\right]^{-1} \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \\
x_{k+1} &= x_k -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k).
\end{aligned}$$

Newton method as a local quadratic Taylor approximation minimizer

Let us now have the function $f(x)$ and a certain point x_k. Let us consider the quadratic approximation of this function near x_k:

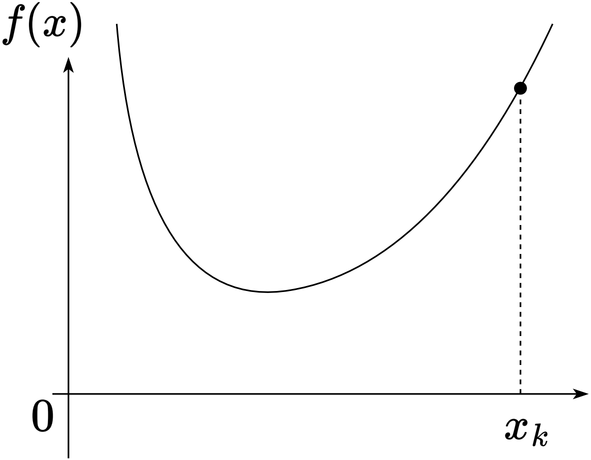
$$f^{II}_{x_k}(x) = f(x_k) + \langle \nabla f(x_k), x - x_k\rangle + \frac{1}{2} \langle \nabla^2 f(x_k)(x-x_k), x-x_k \rangle.$$

The idea of the method is to find the point x_{k+1}, that minimizes the function $f^{II}_{x_k}(x)$, i.e. $\nabla f^{II}_{x_k}(x_{k+1}) = 0$.

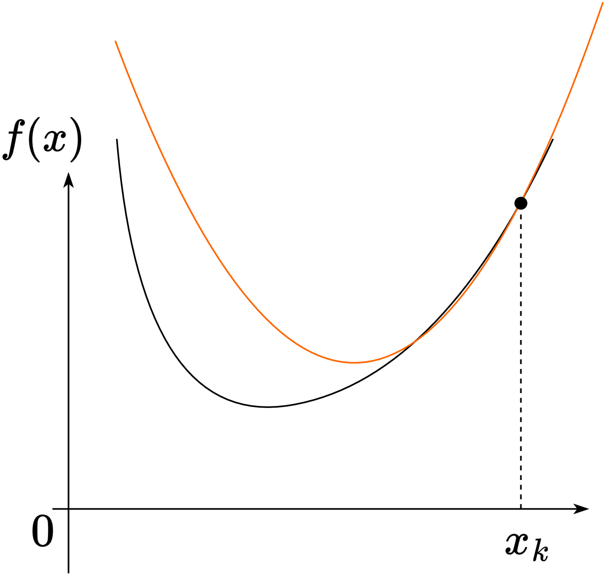
$$\begin{aligned} \nabla f^{II}_{x_k}(x_{k+1}) &= \nabla f(x_{k}) + \nabla^2 f(x_k)(x_{k+1} - x_k) = 0 \\ \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\nabla f(x_{k}) \\ \left[\nabla^2 f(x_k)\right]^{-1} \nabla^2 f(x_k)(x_{k+1} - x_k) &= -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_{k}) \\ x_{k+1} &= x_k -\left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_{k}). \end{aligned}$$

Let us immediately note the limitations related to the necessity of the Hessian's non-degeneracy (for the method to exist), as well as its positive definiteness (for the convergence guarantee).

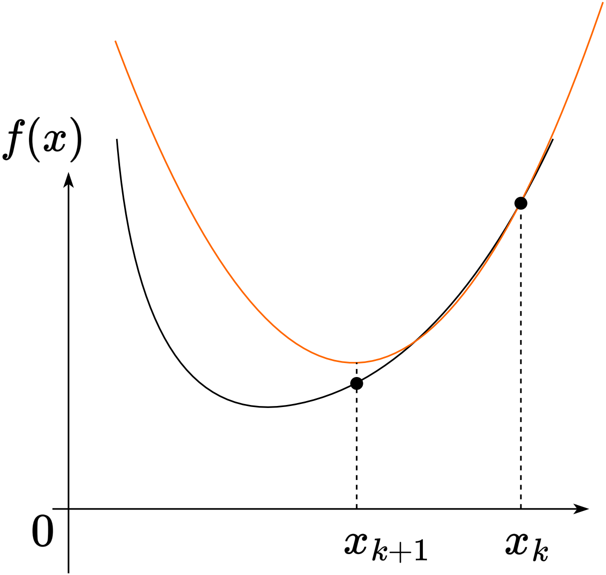
Newton method as a local quadratic Taylor approximation minimizer



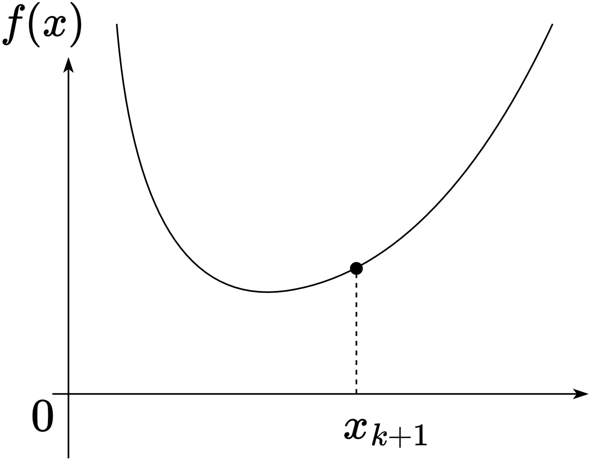
Newton method as a local quadratic Taylor approximation minimizer



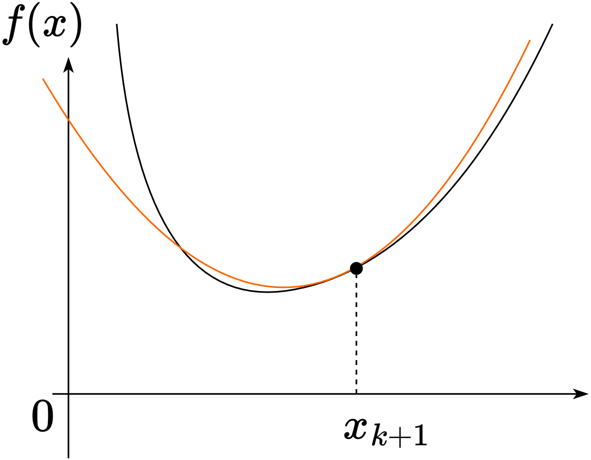
Newton method as a local quadratic Taylor approximation minimizer



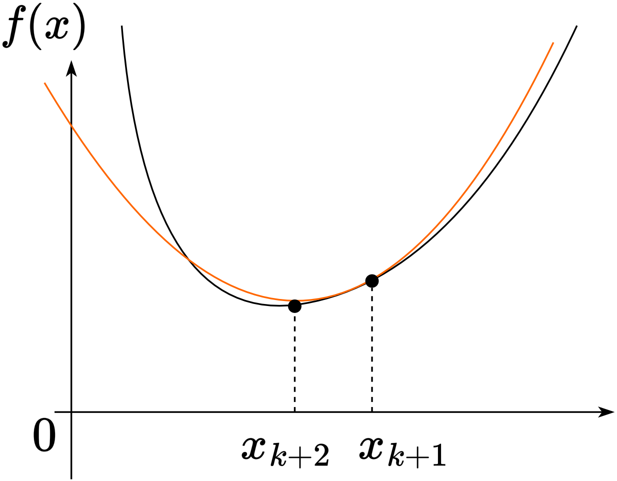
Newton method as a local quadratic Taylor approximation minimizer



Newton method as a local quadratic Taylor approximation minimizer



Newton method as a local quadratic Taylor approximation minimizer



Convergence

Theorem

Let $f(x)$ be a strongly convex twice continuously differentiable function at $\mathbb{R}^n$, for the second derivative of which inequalities are executed: $\mu I_n \preceq \nabla^2 f(x) \preceq L I_n$. Then Newton's method with a constant step locally converges to solving the problem with superlinear speed. If, in addition, Hessian is M-Lipschitz continuous, then this method converges locally to x^* at a quadratic rate.

Thus, we have an important result: Newton's method for the function with Lipschitz positive-definite Hessian converges **quadratically** near ($\|x_0 - x^*\| < \frac{2\mu}{3M}$) to the solution.

Affine Invariance of Newton's Method

An important property of Newton's method is **affine invariance**. Given a function f and a nonsingular matrix $A \in \mathbb{R}^{n \times n}$, let $x = Ay$, and define $g(y) = f(Ay)$. Note, that $\nabla g(y) = A^T \nabla f(x)$ and $\nabla^2 g(y) = A^T \nabla^2 f(x) A$. The Newton steps on g are expressed as:

$$y_{k+1} = y_k - \left(\nabla^2 g(y_k)\right)^{-1} \nabla g(y_k)$$

Affine Invariance of Newton's Method

An important property of Newton's method is **affine invariance**. Given a function f and a nonsingular matrix $A \in \mathbb{R}^{n \times n}$, let $x = Ay$, and define $g(y) = f(Ay)$. Note, that $\nabla g(y) = A^T \nabla f(x)$ and $\nabla^2 g(y) = A^T \nabla^2 f(x) A$. The Newton steps on g are expressed as:

$$y_{k+1} = y_k - \left(\nabla^2 g(y_k)\right)^{-1} \nabla g(y_k)$$

Expanding this, we get:

$$y_{k+1} = y_k - \left(A^T \nabla^2 f(Ay_k) A\right)^{-1} A^T \nabla f(Ay_k)$$

Affine Invariance of Newton's Method

An important property of Newton's method is **affine invariance**. Given a function f and a nonsingular matrix $A \in \mathbb{R}^{n \times n}$, let $x = Ay$, and define $g(y) = f(Ay)$. Note, that $\nabla g(y) = A^T \nabla f(x)$ and $\nabla^2 g(y) = A^T \nabla^2 f(x) A$. The Newton steps on g are expressed as:

$$y_{k+1} = y_k - \left(\nabla^2 g(y_k)\right)^{-1} \nabla g(y_k)$$

Expanding this, we get:

$$y_{k+1} = y_k - \left(A^T \nabla^2 f(Ay_k) A\right)^{-1} A^T \nabla f(Ay_k)$$

Using the property of matrix inverse $(AB)^{-1} = B^{-1}A^{-1}$, this simplifies to:

$$\begin{aligned} y_{k+1} &= y_k - A^{-1} \left(\nabla^2 f(Ay_k)\right)^{-1} \nabla f(Ay_k) \\ Ay_{k+1} &= Ay_k - \left(\nabla^2 f(Ay_k)\right)^{-1} \nabla f(Ay_k) \end{aligned}$$

Affine Invariance of Newton's Method

An important property of Newton's method is **affine invariance**. Given a function f and a nonsingular matrix $A \in \mathbb{R}^{n \times n}$, let $x = Ay$, and define $g(y) = f(Ay)$. Note, that $\nabla g(y) = A^T \nabla f(x)$ and $\nabla^2 g(y) = A^T \nabla^2 f(x) A$. The Newton steps on g are expressed as:

$$y_{k+1} = y_k - \left(\nabla^2 g(y_k)\right)^{-1} \nabla g(y_k)$$

Expanding this, we get:

$$y_{k+1} = y_k - \left(A^T \nabla^2 f(Ay_k) A\right)^{-1} A^T \nabla f(Ay_k)$$

Using the property of matrix inverse $(AB)^{-1} = B^{-1}A^{-1}$, this simplifies to:

$$\begin{aligned} y_{k+1} &= y_k - A^{-1} \left(\nabla^2 f(Ay_k)\right)^{-1} \nabla f(Ay_k) \\ Ay_{k+1} &= Ay_k - \left(\nabla^2 f(Ay_k)\right)^{-1} \nabla f(Ay_k) \end{aligned}$$

Thus, the update rule for x is:

$$x_{k+1} = x_k - \left(\nabla^2 f(x_k)\right)^{-1} \nabla f(x_k)$$

Affine Invariance of Newton's Method

An important property of Newton's method is **affine invariance**. Given a function f and a nonsingular matrix $A \in \mathbb{R}^{n \times n}$, let $x = Ay$, and define $g(y) = f(Ay)$. Note, that $\nabla g(y) = A^T \nabla f(x)$ and $\nabla^2 g(y) = A^T \nabla^2 f(x) A$. The Newton steps on g are expressed as:

$$y_{k+1} = y_k - \left(\nabla^2 g(y_k)\right)^{-1} \nabla g(y_k)$$

Expanding this, we get:

$$y_{k+1} = y_k - \left(A^T \nabla^2 f(Ay_k) A\right)^{-1} A^T \nabla f(Ay_k)$$

Using the property of matrix inverse $(AB)^{-1} = B^{-1}A^{-1}$, this simplifies to:

$$\begin{aligned} y_{k+1} &= y_k - A^{-1} \left(\nabla^2 f(Ay_k)\right)^{-1} \nabla f(Ay_k) \\ Ay_{k+1} &= Ay_k - \left(\nabla^2 f(Ay_k)\right)^{-1} \nabla f(Ay_k) \end{aligned}$$

Thus, the update rule for x is:

$$x_{k+1} = x_k - \left(\nabla^2 f(x_k)\right)^{-1} \nabla f(x_k)$$

This shows that the progress made by Newton's method is independent of problem scaling. This property is not shared by the gradient descent method!

Summary

What's nice:

- quadratic convergence near the solution x^*

Summary

What's nice:

- quadratic convergence near the solution x^*
- affine invariance

Summary

What's nice:

- quadratic convergence near the solution x^*
- affine invariance
- the parameters have little effect on the convergence rate

Summary

What's nice:

- quadratic convergence near the solution x^*
- affine invariance
- the parameters have little effect on the convergence rate

Summary

What's nice:

- quadratic convergence near the solution x^*
- affine invariance
- the parameters have little effect on the convergence rate

What's not nice:

- it is necessary to store the (inverse) hessian on each iteration: $\mathcal{O}(n^2)$ memory

Summary

What's nice:

- quadratic convergence near the solution x^*
- affine invariance
- the parameters have little effect on the convergence rate

What's not nice:

- it is necessary to store the (inverse) hessian on each iteration: $\mathcal{O}(n^2)$ memory
- it is necessary to solve linear systems: $\mathcal{O}(n^3)$ operations

Summary

What's nice:

- quadratic convergence near the solution x^*
- affine invariance
- the parameters have little effect on the convergence rate

What's not nice:

- it is necessary to store the (inverse) hessian on each iteration: $\mathcal{O}(n^2)$ memory
- it is necessary to solve linear systems: $\mathcal{O}(n^3)$ operations
- the Hessian can be degenerate at x^*

Summary

What's nice:

- quadratic convergence near the solution x^*
- affine invariance
- the parameters have little effect on the convergence rate

What's not nice:

- it is necessary to store the (inverse) hessian on each iteration: $\mathcal{O}(n^2)$ memory
- it is necessary to solve linear systems: $\mathcal{O}(n^3)$ operations
- the Hessian can be degenerate at x^*
- the hessian may not be positively determined → direction $-(f''(x))^{-1}f'(x)$ may not be a descending direction

Newton method problems

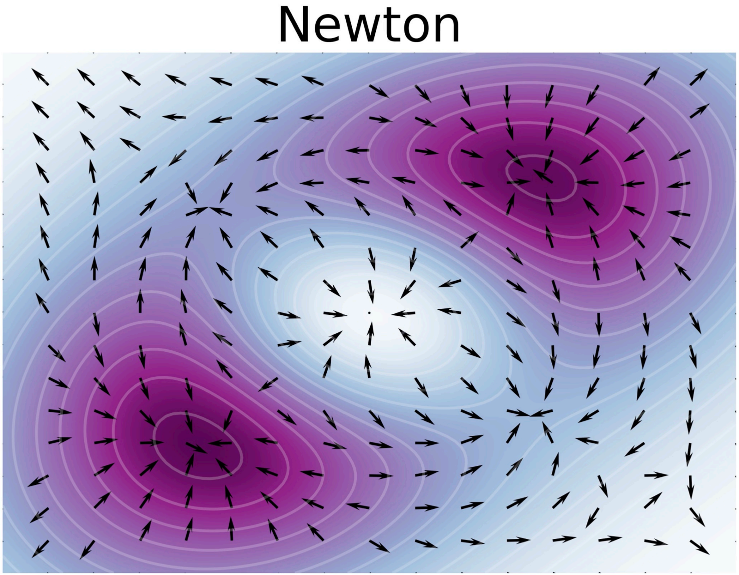


Figure 17: Animation

Newton method problems

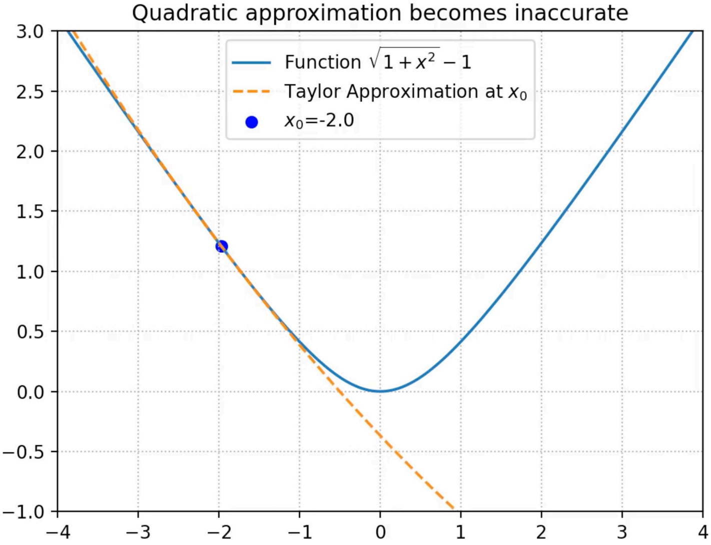


Figure 18: Animation

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

Then, we can define another *steepest descent* direction in terms of minimizer of function on a sphere:

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

Then, we can define another *steepest descent* direction in terms of minimizer of function on a sphere:

$$s = \lim_{\varepsilon \to 0} \frac{x^* - x_0}{\varepsilon}$$

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

Then, we can define another *steepest descent* direction in terms of minimizer of function on a sphere:

$$s = \lim_{\varepsilon \to 0} \frac{x^* - x_0}{\varepsilon}$$

Let us assume that the distance is defined locally by some metric A:

$$d(x, x_0) = (x - x_0)^\top A(x - x_0)$$

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

Then, we can define another *steepest descent* direction in terms of minimizer of function on a sphere:

$$s = \lim_{\varepsilon \to 0} \frac{x^* - x_0}{\varepsilon}$$

Let us assume that the distance is defined locally by some metric A:

$$d(x, x_0) = (x - x_0)^\top A(x - x_0)$$

Let us also consider first order Taylor approximation of a function $f(x)$ near the point x_0:

$$f(x_0 + \delta x) \approx f(x_0) + \nabla f(x_0)^\top \delta x \tag{1}$$

Now we can explicitly pose a problem of finding s, as it was stated above.

$$\min_{\delta x \in \mathbb{R}^{\ltimes}} f(x_0 + \delta x)$$

$$\text{s.t. } \delta x^\top A \delta x = \varepsilon^2$$

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

Then, we can define another *steepest descent* direction in terms of minimizer of function on a sphere:

$$s = \lim_{\varepsilon \to 0} \frac{x^* - x_0}{\varepsilon}$$

Let us assume that the distance is defined locally by some metric A:

$$d(x, x_0) = (x - x_0)^\top A(x - x_0)$$

Let us also consider first order Taylor approximation of a function $f(x)$ near the point x_0:

$$f(x_0 + \delta x) \approx f(x_0) + \nabla f(x_0)^\top \delta x \tag{1}$$

Now we can explicitly pose a problem of finding s, as it was stated above.

$$\min_{\delta x \in \mathbb{R}^{\ltimes}} f(x_0 + \delta x)$$

$$\text{s.t. } \delta x^\top A \delta x = \varepsilon^2$$

Using equation (1 it can be written as:

$$\min_{\delta x \in \mathbb{R}^{\ltimes}} \nabla f(x_0)^\top \delta x$$

$$\text{s.t. } \delta x^\top A \delta x = \varepsilon^2$$

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

Then, we can define another *steepest descent* direction in terms of minimizer of function on a sphere:

$$s = \lim_{\varepsilon \to 0} \frac{x^* - x_0}{\varepsilon}$$

Let us assume that the distance is defined locally by some metric A:

$$d(x, x_0) = (x - x_0)^\top A(x - x_0)$$

Let us also consider first order Taylor approximation of a function $f(x)$ near the point x_0:

$$f(x_0 + \delta x) \approx f(x_0) + \nabla f(x_0)^\top \delta x \tag{1}$$

Now we can explicitly pose a problem of finding s, as it was stated above.

$$\min_{\delta x \in \mathbb{R}^{\ltimes}} f(x_0 + \delta x)$$

$$\text{s.t. } \delta x^\top A \delta x = \varepsilon^2$$

Using equation (1 it can be written as:

$$\min_{\delta x \in \mathbb{R}^{\ltimes}} \nabla f(x_0)^\top \delta x$$

$$\text{s.t. } \delta x^\top A \delta x = \varepsilon^2$$

Using Lagrange multipliers method, we can easily conclude, that the answer is:

$$\delta x = -\frac{2\varepsilon^2}{\nabla f(x_0)^\top A^{-1} \nabla f(x_0)} A^{-1} \nabla f$$

The idea of adapive metrics

Given $f(x)$ and a point x_0. Define $B_\varepsilon(x_0) = \{x \in \mathbb{R}^n : d(x, x_0) = \varepsilon^2\}$ as the set of points with distance ε to x_0. Here we presume the existence of a distance function $d(x, x_0)$.

$$x^* = \text{arg} \min_{x \in B_\varepsilon(x_0)} f(x)$$

Then, we can define another *steepest descent* direction in terms of minimizer of function on a sphere:

$$s = \lim_{\varepsilon \to 0} \frac{x^* - x_0}{\varepsilon}$$

Let us assume that the distance is defined locally by some metric A:

$$d(x, x_0) = (x - x_0)^\top A (x - x_0)$$

Let us also consider first order Taylor approximation of a function $f(x)$ near the point x_0:

$$f(x_0 + \delta x) \approx f(x_0) + \nabla f(x_0)^\top \delta x \qquad (1)$$

Now we can explicitly pose a problem of finding s, as it was stated above.

$$\min_{\delta x \in \mathbb{R}^{\ltimes}} f(x_0 + \delta x)$$
$$\text{s.t. } \delta x^\top A \delta x = \varepsilon^2$$

Using equation (1 it can be written as:

$$\min_{\delta x \in \mathbb{R}^{\ltimes}} \nabla f(x_0)^\top \delta x$$
$$\text{s.t. } \delta x^\top A \delta x = \varepsilon^2$$

Using Lagrange multipliers method, we can easily conclude, that the answer is:

$$\delta x = -\frac{2\varepsilon^2}{\nabla f(x_0)^\top A^{-1} \nabla f(x_0)} A^{-1} \nabla f$$

Which means, that new direction of steepest descent is nothing else, but $A^{-1}\nabla f(x_0)$.

. . . Indeed, if the space is isotropic and $A = I$, we immediately have gradient descent formula, while Newton method uses local Hessian as a metric matrix.

Quasi-Newton methods

Quasi-Newton methods intuition

For the classic task of unconditional optimization $f(x) \to \min\limits_{x \in \mathbb{R}^n}$ the general scheme of iteration method is written as:

$$x_{k+1} = x_k + \alpha_k d_k$$

Quasi-Newton methods intuition

For the classic task of unconditional optimization $f(x) \to \min\limits_{x \in \mathbb{R}^n}$ the general scheme of iteration method is written as:

$$x_{k+1} = x_k + \alpha_k d_k$$

In the Newton method, the d_k direction (Newton's direction) is set by the linear system solution at each step:

$$B_k d_k = -\nabla f(x_k), \quad B_k = \nabla^2 f(x_k)$$

Quasi-Newton methods intuition

For the classic task of unconditional optimization $f(x) \to \min\limits_{x \in \mathbb{R}^n}$ the general scheme of iteration method is written as:

$$x_{k+1} = x_k + \alpha_k d_k$$

In the Newton method, the d_k direction (Newton's direction) is set by the linear system solution at each step:

$$B_k d_k = -\nabla f(x_k), \quad B_k = \nabla^2 f(x_k)$$

i.e. at each iteration it is necessary to **compute** hessian and gradient and **solve** linear system.

Quasi-Newton methods intuition

For the classic task of unconditional optimization $f(x) \to \min\limits_{x \in \mathbb{R}^n}$ the general scheme of iteration method is written as:

$$x_{k+1} = x_k + \alpha_k d_k$$

In the Newton method, the d_k direction (Newton's direction) is set by the linear system solution at each step:

$$B_k d_k = -\nabla f(x_k), \quad B_k = \nabla^2 f(x_k)$$

i.e. at each iteration it is necessary to **compute** hessian and gradient and **solve** linear system.

Note here that if we take a single matrix of $B_k = I_n$ as B_k at each step, we will exactly get the gradient descent method.

The general scheme of quasi-Newton methods is based on the selection of the B_k matrix so that it tends in some sense at $k \to \infty$ to the truth value of the Hessian $\nabla^2 f(x_k)$.

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \dots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Different quasi-Newton methods implement Step 3 differently. As we will see, commonly we can compute $(B_{k+1})^{-1}$ from $(B_k)^{-1}$.

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Different quasi-Newton methods implement Step 3 differently. As we will see, commonly we can compute $(B_{k+1})^{-1}$ from $(B_k)^{-1}$.

Basic Idea: As B_k already contains information about the Hessian, use a suitable matrix update to form B_{k+1}.

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Different quasi-Newton methods implement Step 3 differently. As we will see, commonly we can compute $(B_{k+1})^{-1}$ from $(B_k)^{-1}$.

Basic Idea: As B_k already contains information about the Hessian, use a suitable matrix update to form B_{k+1}.

Reasonable Requirement for B_{k+1} (motivated by the secant method):

$$\nabla f(x_{k+1}) - \nabla f(x_k) = B_{k+1}(x_{k+1} - x_k) = B_{k+1} d_k$$

$$\Delta y_k = B_{k+1} \Delta x_k$$

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \dots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Different quasi-Newton methods implement Step 3 differently. As we will see, commonly we can compute $(B_{k+1})^{-1}$ from $(B_k)^{-1}$.

Basic Idea: As B_k already contains information about the Hessian, use a suitable matrix update to form B_{k+1}.

Reasonable Requirement for B_{k+1} (motivated by the secant method):

$$\nabla f(x_{k+1}) - \nabla f(x_k) = B_{k+1}(x_{k+1} - x_k) = B_{k+1} d_k$$
$$\Delta y_k = B_{k+1} \Delta x_k$$

In addition to the secant equation, we want:

- B_{k+1} to be symmetric

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Different quasi-Newton methods implement Step 3 differently. As we will see, commonly we can compute $(B_{k+1})^{-1}$ from $(B_k)^{-1}$.

Basic Idea: As B_k already contains information about the Hessian, use a suitable matrix update to form B_{k+1}.

Reasonable Requirement for B_{k+1} (motivated by the secant method):

$$\nabla f(x_{k+1}) - \nabla f(x_k) = B_{k+1}(x_{k+1} - x_k) = B_{k+1} d_k$$
$$\Delta y_k = B_{k+1} \Delta x_k$$

In addition to the secant equation, we want:

- B_{k+1} to be symmetric
- B_{k+1} to be "close" to B_k

Quasi-Newton Method Template

Let $x_0 \in \mathbb{R}^n$, $B_0 \succ 0$. For $k = 1, 2, 3, \ldots$, repeat:

1. Solve $B_k d_k = -\nabla f(x_k)$
2. Update $x_{k+1} = x_k + \alpha_k d_k$
3. Compute B_{k+1} from B_k

Different quasi-Newton methods implement Step 3 differently. As we will see, commonly we can compute $(B_{k+1})^{-1}$ from $(B_k)^{-1}$.

Basic Idea: As B_k already contains information about the Hessian, use a suitable matrix update to form B_{k+1}.

Reasonable Requirement for B_{k+1} (motivated by the secant method):

$$\nabla f(x_{k+1}) - \nabla f(x_k) = B_{k+1}(x_{k+1} - x_k) = B_{k+1} d_k$$
$$\Delta y_k = B_{k+1} \Delta x_k$$

In addition to the secant equation, we want:

- B_{k+1} to be symmetric
- B_{k+1} to be “close” to B_k
- $B_k \succ 0 \Rightarrow B_{k+1} \succ 0$

Symmetric Rank-One Update

Let's try an update of the form:

$$B_{k+1} = B_k + auu^T$$

Symmetric Rank-One Update

Let's try an update of the form:

$$B_{k+1} = B_k + auu^T$$

The secant equation $B_{k+1}d_k = \Delta y_k$ yields:

$$(au^T d_k)u = \Delta y_k - B_k d_k$$

Symmetric Rank-One Update

Let's try an update of the form:

$$B_{k+1} = B_k + auu^T$$

The secant equation $B_{k+1}d_k = \Delta y_k$ yields:

$$(au^T d_k)u = \Delta y_k - B_k d_k$$

This only holds if u is a multiple of $\Delta y_k - B_k d_k$. Putting $u = \Delta y_k - B_k d_k$, we solve the above,

$$a = \frac{1}{(\Delta y_k - B_k d_k)^T d_k},$$

Symmetric Rank-One Update

Let's try an update of the form:

$$B_{k+1} = B_k + auu^T$$

The secant equation $B_{k+1}d_k = \Delta y_k$ yields:

$$(au^T d_k)u = \Delta y_k - B_k d_k$$

This only holds if u is a multiple of $\Delta y_k - B_k d_k$. Putting $u = \Delta y_k - B_k d_k$, we solve the above,

$$a = \frac{1}{(\Delta y_k - B_k d_k)^T d_k},$$

which leads to

$$B_{k+1} = B_k + \frac{(\Delta y_k - B_k d_k)(\Delta y_k - B_k d_k)^T}{(\Delta y_k - B_k d_k)^T d_k}$$

called the symmetric rank-one (SR1) update or Broyden method.

Symmetric Rank-One Update with inverse

How can we solve

$$B_{k+1}d_{k+1} = -\nabla f(x_{k+1}),$$

in order to take the next step? In addition to propagating B_k to B_{k+1}, let's propagate inverses, i.e., $C_k = B_k^{-1}$ to $C_{k+1} = (B_{k+1})^{-1}$.

Sherman-Morrison Formula:

The Sherman-Morrison formula states:

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

Thus, for the SR1 update, the inverse is also easily updated:

$$C_{k+1} = C_k + \frac{(d_k - C_k\Delta y_k)(d_k - C_k\Delta y_k)^T}{(d_k - C_k\Delta y_k)^T\Delta y_k}$$

In general, SR1 is simple and cheap, but it has a key shortcoming: it does not preserve positive definiteness.

Davidon-Fletcher-Powell Update

We could have pursued the same idea to update the inverse C:

$$C_{k+1} = C_k + auu^T + bvv^T.$$

Davidon-Fletcher-Powell Update

We could have pursued the same idea to update the inverse C:

$$C_{k+1} = C_k + auu^T + bvv^T.$$

Multiplying by Δy_k, using the secant equation $d_k = C_k \Delta y_k$, and solving for a, b, yields:

$$C_{k+1} = C_k - \frac{C_k \Delta y_k \Delta y_k^T C_k}{\Delta y_k^T C_k \Delta y_k} + \frac{d_k d_k^T}{\Delta y_k^T d_k}$$

Woodbury Formula Application

Woodbury then shows:

$$B_{k+1} = \left(I - \frac{\Delta y_k d_k^T}{\Delta y_k^T d_k} \right) B_k \left(I - \frac{d_k \Delta y_k^T}{\Delta y_k^T d_k} \right) + \frac{\Delta y_k \Delta y_k^T}{\Delta y_k^T d_k}$$

This is the Davidon-Fletcher-Powell (DFP) update. Also cheap: $O(n^2)$, preserves positive definiteness. Not as popular as BFGS.

Broyden-Fletcher-Goldfarb-Shanno update

Let's now try a rank-two update:

$$B_{k+1} = B_k + auu^T + bvv^T.$$

Broyden-Fletcher-Goldfarb-Shanno update

Let's now try a rank-two update:

$$B_{k+1} = B_k + auu^T + bvv^T.$$

The secant equation $\Delta y_k = B_{k+1} d_k$ yields:

$$\Delta y_k - B_k d_k = (au^T d_k)u + (bv^T d_k)v$$

Broyden-Fletcher-Goldfarb-Shanno update

Let's now try a rank-two update:

$$B_{k+1} = B_k + auu^T + bvv^T.$$

The secant equation $\Delta y_k = B_{k+1} d_k$ yields:

$$\Delta y_k - B_k d_k = (au^T d_k)u + (bv^T d_k)v$$

Putting $u = \Delta y_k$, $v = B_k d_k$, and solving for a, b we get:

$$B_{k+1} = B_k - \frac{B_k d_k d_k^T B_k}{d_k^T B_k d_k} + \frac{\Delta y_k \Delta y_k^T}{d_k^T \Delta y_k}$$

called the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update.

Broyden-Fletcher-Goldfarb-Shanno update with inverse

Woodbury Formula

The Woodbury formula, a generalization of the Sherman-Morrison formula, is given by:

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$$

Broyden-Fletcher-Goldfarb-Shanno update with inverse

Woodbury Formula

The Woodbury formula, a generalization of the Sherman-Morrison formula, is given by:

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$$

Applied to our case, we get a rank-two update on the inverse C:

$$C_{k+1} = C_k + \frac{(d_k - C_k \Delta y_k) d_k^T}{\Delta y_k^T d_k} + \frac{d_k (d_k - C_k \Delta y_k)^T}{\Delta y_k^T d_k} - \frac{(d_k - C_k \Delta y_k)^T \Delta y_k}{(\Delta y_k^T d_k)^2} d_k d_k^T$$

$$C_{k+1} = \left(I - \frac{d_k \Delta y_k^T}{\Delta y_k^T d_k}\right) C_k \left(I - \frac{\Delta y_k d_k^T}{\Delta y_k^T d_k}\right) + \frac{d_k d_k^T}{\Delta y_k^T d_k}$$

This formulation ensures that the BFGS update, while comprehensive, remains computationally efficient, requiring $O(n^2)$ operations. Importantly, BFGS update preserves positive definiteness. Recall this means $B_k \succ 0 \Rightarrow B_{k+1} \succ 0.$ Equivalently, $C_k \succ 0 \Rightarrow C_{k+1} \succ 0$

Code

- Open In Colab

Code

- Open In Colab
- Comparison of quasi Newton methods

Code

- Open In Colab
- Comparison of quasi Newton methods
- Some practical notes about Newton method