GraSS: Scalable Influence Function with Sparse Gradient Compression

A Foray to **Efficient** Data Attribution and Influence Function

Pingbang Hu[1] Joseph Melkonian[2] Weijing Tang[3] Han Zhao[1] Jiaqi W. Ma[1]

[1]University of Illinois Urbana-Champaign [2]Womp Labs [3]Carnegie Mellon University

September 23, 2025

Table of Content

Table of Content

Overview

Most of the popular data attribution methods are *gradient-based*:

Overview

Most of the popular data attribution methods are *gradient-based*:

- **Influence Function**: Influence Function [KL17], TRAK [Par+23], etc.
- **Training Dynamic**: SGD-influence [HNM19], Data-Value Embedding [Wan+25b], etc.

Overview

Most of the popular data attribution methods are *gradient-based*:

- **Influence Function**: Influence Function [KL17], TRAK [Par+23], etc.
- **Training Dynamic**: SGD-influence [HNM19], Data-Value Embedding [Wan+25b], etc.

Most of the methods are expensive, both *computation*-wise and *memory*-wise...

Overview

Most of the popular data attribution methods are *gradient-based*:

- **Influence Function**: Influence Function [KL17], TRAK [Par+23], etc.
- **Training Dynamic**: SGD-influence [HNM19], Data-Value Embedding [Wan+25b], etc.

Most of the methods are expensive, both *computation*-wise and *memory*-wise...

Goal

Introduce all common tricks for speeding up gradient-based data attribution methods.

Overview

Most of the popular data attribution methods are *gradient-based*:

- **Influence Function**: Influence Function [KL17], TRAK [Par+23], etc.
- **Training Dynamic**: SGD-influence [HNM19], Data-Value Embedding [Wan+25b], etc.

Most of the methods are expensive, both *computation*-wise and *memory*-wise...

Goal

Introduce all common tricks for speeding up gradient-based data attribution methods.

- *FIM block-diagonal approximation of Hessian*

Overview

Most of the popular data attribution methods are *gradient-based*:

- **Influence Function**: Influence Function [KL17], TRAK [Par+23], etc.
- **Training Dynamic**: SGD-influence [HNM19], Data-Value Embedding [Wan+25b], etc.

Most of the methods are expensive, both *computation*-wise and *memory*-wise...

Goal

Introduce all common tricks for speeding up gradient-based data attribution methods.

- *FIM block-diagonal approximation of Hessian*
- *Gradient compression:* Random *[Woj+16],* LoGra *[Cho+24], and* GraSS *[Hu+25]*

Overview

Most of the popular data attribution methods are *gradient-based*:

- **Influence Function**: Influence Function [KL17], TRAK [Par+23], etc.
- **Training Dynamic**: SGD-influence [HNM19], Data-Value Embedding [Wan+25b], etc.

Most of the methods are expensive, both *computation*-wise and *memory*-wise...

Goal

Introduce all common tricks for speeding up gradient-based data attribution methods.

- *FIM block-diagonal approximation of Hessian*
- *Gradient compression:* RANDOM *[Woj+16],* LOGRA *[Cho+24], and* GRASS *[Hu+25]*

Example (Running example)

We will consider the classical Influence Function [KL17] throughout the talk.

Table of Content

Data Attribution

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

- Say we have a model $\hat{\theta}_D$ trained on D, with $p = |\hat{\theta}_D|$ and $n = |D|$

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

- Say we have a model $\hat{\theta}_D$ trained on D, with $p = |\hat{\theta}_D|$ and $n = |D|$
- Given a quantity of interest—a *target* function $f(D)$ of $\hat{\theta}_D$, e.g., validation loss

Data Attribution

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

- Say we have a model $\hat{\theta}_D$ trained on D, with $p = |\hat{\theta}_D|$ and $n = |D|$
- Given a quantity of interest—a *target* function $f(D)$ of $\hat{\theta}_D$, e.g., validation loss
- Predict how f will change, if the dataset D is *counterfactually* perturbed to D':

$$\Delta f = f(D') - f(D).$$

Data Attribution

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

- Say we have a model $\hat{\theta}_D$ trained on D, with $p = |\hat{\theta}_D|$ and $n = |D|$
- Given a quantity of interest—a *target* function $f(D)$ of $\hat{\theta}_D$, e.g., validation loss
- Predict how f will change, if the dataset D is *counterfactually* perturbed to D':

$$\Delta f = f(D') - f(D).$$

Popular methods study this from a fine-grained, localized viewpoint:

Data Attribution

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

- Say we have a model $\hat{\theta}_D$ trained on D, with $p = |\hat{\theta}_D|$ and $n = |D|$
- Given a quantity of interest—a *target* function $f(D)$ of $\hat{\theta}_D$, e.g., validation loss
- Predict how f will change, if the dataset D is *counterfactually* perturbed to D':

$$\Delta f = f(D') - f(D).$$

Popular methods study this from a fine-grained, localized viewpoint:

1. Consider D' of the form $D' = D \setminus B$ for a small batch of samples B (or $D' = D \cup B$)

Data Attribution

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

- Say we have a model $\hat{\theta}_D$ trained on D, with $p = |\hat{\theta}_D|$ and $n = |D|$
- Given a quantity of interest—a *target* function $f(D)$ of $\hat{\theta}_D$, e.g., validation loss
- Predict how f will change, if the dataset D is *counterfactually* perturbed to D':

$$\Delta f = f(D') - f(D).$$

Popular methods study this from a fine-grained, localized viewpoint:

1. Consider D' of the form $D' = D \setminus B$ for a small batch of samples B (or $D' = D \cup B$)
2. For each possible B, we predict $\tau_f(B) := f(D \setminus B) - f(D)$ (or $f(D \cup B) - f(D)$)

Data Attribution

Data attribution algorithms quantify *counterfactual effect* for **dataset perturbation**:

- Say we have a model $\hat{\theta}_D$ trained on D, with $p = |\hat{\theta}_D|$ and $n = |D|$
- Given a quantity of interest—a *target* function $f(D)$ of $\hat{\theta}_D$, e.g., validation loss
- Predict how f will change, if the dataset D is *counterfactually* perturbed to D':

$$\Delta f = f(D') - f(D).$$

Popular methods study this from a fine-grained, localized viewpoint:

1. Consider D' of the form $D' = D \setminus B$ for a small batch of samples B (or $D' = D \cup B$)
2. For each possible B, we predict $\tau_f(B) := f(D \setminus B) - f(D)$ (or $f(D \cup B) - f(D)$)

Popular choice of B: $B_i = \{z_i\}$ for $z_i \in D$, i.e., $\tau_f(B_i)$ provides the *point-wise* effect.

Intuition (Estimating τ_f)

Parametrize D by a default weight vector $w = \mathbb{1}/n \in \mathbb{R}^n$ for the data points z_i's.

[1] For notational simplicity, we write $\ell_i := \ell(z_i; \theta)$ hereafter.

Intuition (Estimating τ_f)

Parametrize D by a default weight vector $w = \mathbb{1}/n \in \mathbb{R}^n$ for the data points z_i's.

$\Rightarrow$ *Model trained on (weighted) D is a function of w:* $\hat{\theta}_w = \arg\min_\theta \sum_{z_i \in D} w_i \ell_i$[1]

[1] For notational simplicity, we write $\ell_i := \ell(z_i; \theta)$ hereafter.

Intuition (Estimating τ_f)

Parametrize D by a default weight vector $w = \mathbb{1}/n \in \mathbb{R}^n$ for the data points z_i's.

$\Rightarrow$ *Model trained on (weighted) D is a function of w:* $\hat{\theta}_w = \arg\min_\theta \sum_{z_i \in D} w_i \ell_i$[1]

$\Rightarrow$ *Taylor-expand $\hat{\theta}_w$ around $w = \mathbb{1}/n \Leftrightarrow$ estimating perturbation effects ($D \to D'$)*

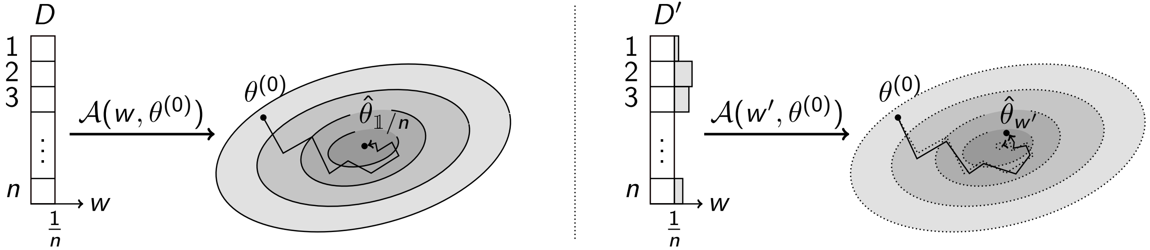
[1] For notational simplicity, we write $\ell_i \coloneqq \ell(z_i; \theta)$ hereafter.

Intuition (Estimating τ_f)

Parametrize D by a default weight vector $w = \mathbb{1}/n \in \mathbb{R}^n$ for the data points z_i's.

$\Rightarrow$ *Model trained on (weighted) D is a function of w:* $\hat{\theta}_w = \arg\min_\theta \sum_{z_i \in D} w_i \ell_i$[1]

$\Rightarrow$ *Taylor-expand $\hat{\theta}_w$ around $w = \mathbb{1}/n \Leftrightarrow$ estimating perturbation effects ($D \to D'$)*



[1] For notational simplicity, we write $\ell_i := \ell(z_i; \theta)$ hereafter.

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i$

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

1. From **first-order approximation** (i.e., Taylor expansion):

$$\Delta f$$

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

1. From **first-order approximation** (i.e., Taylor expansion):

$$\Delta f = \tau_f(\{z_i\})$$

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

1. From **first-order approximation** (i.e., Taylor expansion):

$$\Delta f = \tau_f(\{z_i\}) = [f(D + \epsilon z_i) - f(D)]\big|_{\epsilon = -\frac{1}{n}}$$

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

1. From **first-order approximation** (i.e., Taylor expansion):

$$\Delta f = \tau_f(\{z_i\}) = [f(D + \epsilon z_i) - f(D)]\big|_{\epsilon = -\frac{1}{n}} \approx \epsilon\big|_{\epsilon = -\frac{1}{n}} \cdot \left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon = 0}.$$

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

1. From **first-order approximation** (i.e., Taylor expansion):

$$\Delta f = \tau_f(\{z_i\}) = [f(D + \epsilon z_i) - f(D)]\big|_{\epsilon=-\frac{1}{n}} \approx \epsilon\big|_{\epsilon=-\frac{1}{n}} \cdot \left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon=0}.$$

2. From **chain rule**:

$$\left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon=0}$$

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

1. From **first-order approximation** (i.e., Taylor expansion):

$$\Delta f = \tau_f(\{z_i\}) = [f(D + \epsilon z_i) - f(D)]\big|_{\epsilon = -\frac{1}{n}} \approx \epsilon\big|_{\epsilon = -\frac{1}{n}} \cdot \left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon = 0}.$$

2. From **chain rule**:

$$\left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon = 0} = \nabla_\theta f(\hat{\theta}_{+\epsilon z_i})^\top\big|_{\epsilon = 0} \cdot \left.\frac{\mathrm{d}\hat{\theta}_{+\epsilon z_i}}{\mathrm{d}\epsilon}\right|_{\epsilon = 0}$$

Counterfactual Prediction from Freshman Calculus

To estimate $\tau_f(\{z_i\}) = f(D \setminus \{z_i\}) - f(D)$:

- Write $D \setminus \{z_i\}$ as $D - \frac{1}{n} z_i \Rightarrow \tau_f(\{z_i\}) = f(D + \epsilon z_i) - f(D)$ with $\epsilon = -1/n$!

Since $\hat{\theta}_w$ is a function of w, so is $f(w)$:

1. From **first-order approximation** (i.e., Taylor expansion):

$$\Delta f = \tau_f(\{z_i\}) = [f(D + \epsilon z_i) - f(D)]\big|_{\epsilon=-\frac{1}{n}} \approx \epsilon\big|_{\epsilon=-\frac{1}{n}} \cdot \left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon=0}.$$

2. From **chain rule**:

$$\left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon=0} = \nabla_\theta f(\hat{\theta}_{+\epsilon z_i})^\top\big|_{\epsilon=0} \cdot \left.\frac{\mathrm{d}\hat{\theta}_{+\epsilon z_i}}{\mathrm{d}\epsilon}\right|_{\epsilon=0} = \nabla_\theta f(\hat{\theta}_{\mathbb{1}/n})^\top \cdot \left.\frac{\mathrm{d}\hat{\theta}_{+\epsilon z_i}}{\mathrm{d}\epsilon}\right|_{\epsilon=0}.$$

Influence Function

Theorem (Influence function [KL17; Gro+23])

Let $\hat{\theta} = \hat{\theta}_{\mathbb{1}/n}$ *be the ERM trained on* D *and* $H_{\hat{\theta}} = \frac{1}{n}\sum_{z_i \in D} \nabla_{\theta}^2 \ell_i$ *be the empirical Hessian.*

Theorem (Influence function [KL17; Gro+23])

Let $\hat{\theta} = \hat{\theta}_{\mathbb{1}/n}$ be the ERM trained on D and $H_{\hat{\theta}} = \frac{1}{n}\sum_{z_i \in D} \nabla_{\theta}^2 \ell_i$ be the empirical Hessian. The influence function of upweighting $z_i \in D$ on the target function f is:

$$\mathcal{I}(z_i, f) := \left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon=0} = \nabla_{\theta} f(\hat{\theta})^{\top} \left.\frac{\mathrm{d}\hat{\theta}_{+\epsilon z_i}}{\mathrm{d}\epsilon}\right|_{\epsilon=0} = -\nabla_{\theta} f(\hat{\theta})^{\top} H_{\hat{\theta}}^{-1} \nabla_{\theta} \ell_i.$$

Theorem (Influence function [KL17; Gro+23])

Let $\hat{\theta} = \hat{\theta}_{\mathbb{1}/n}$ be the ERM trained on D and $H_{\hat{\theta}} = \frac{1}{n}\sum_{z_i \in D} \nabla_\theta^2 \ell_i$ be the empirical Hessian. The influence function of upweighting $z_i \in D$ on the target function f is:

$$\mathcal{I}(z_i, f) := \left.\frac{\mathrm{d} f(\hat{\theta}_{+\epsilon z_i})}{\mathrm{d}\epsilon}\right|_{\epsilon=0} = \nabla_\theta f(\hat{\theta})^\top \left.\frac{\mathrm{d}\hat{\theta}_{+\epsilon z_i}}{\mathrm{d}\epsilon}\right|_{\epsilon=0} = -\nabla_\theta f(\hat{\theta})^\top H_{\hat{\theta}}^{-1} \nabla_\theta \ell_i .$$

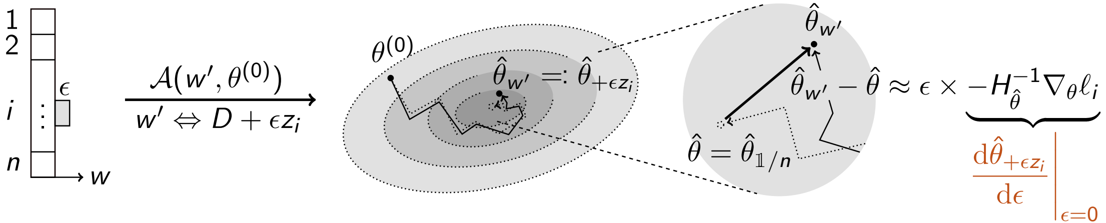


Table of Content

Computing Influence Function

As previously seen (Influence function)

Counterfactual prediction of removing z_i is $\Delta f = \tau_f(\{z_i\}) \approx \epsilon \cdot \mathcal{I}(z_i, f)$ with $\epsilon = -1/n$, where

$$\mathcal{I}(z_i, f) = -\nabla_\theta f(\hat{\theta})^\top H_{\hat{\theta}}^{-1} \nabla_\theta \ell_i, \quad H_{\hat{\theta}} = \frac{1}{n} \sum_{z_i \in D} \nabla_\theta^2 \ell_i$$

As previously seen (Influence function)

Counterfactual prediction of removing z_i is $\Delta f = \tau_f(\{z_i\}) \approx \epsilon \cdot \mathcal{I}(z_i, f)$ with $\epsilon = -1/n$, where

$$\mathcal{I}(z_i, f) = -\nabla_\theta f(\hat{\theta})^\top H_{\hat{\theta}}^{-1} \nabla_\theta \ell_i, \quad H_{\hat{\theta}} = \frac{1}{n} \sum_{z_i \in D} \nabla_\theta^2 \ell_i$$

The main computation is the *inverse-Hessian-vector-product* $H_{\hat{\theta}}^{-1} \times \nabla_\theta \ell_i$, or iHVP:

Computing Influence Function

As previously seen (Influence function)

Counterfactual prediction of removing z_i is $\Delta f = \tau_f(\{z_i\}) \approx \epsilon \cdot \mathcal{I}(z_i, f)$ with $\epsilon = -1/n$, where

$$\mathcal{I}(z_i, f) = -\nabla_\theta f(\hat{\theta})^\top H_{\hat{\theta}}^{-1} \nabla_\theta \ell_i, \quad H_{\hat{\theta}} = \frac{1}{n} \sum_{z_i \in D} \nabla_\theta^2 \ell_i$$

The main computation is the *inverse-Hessian-vector-product* $H_{\hat{\theta}}^{-1} \times \nabla_\theta \ell_i$, or iHVP:

Remark

Once iHVP is solved, $\tau_f(\{z_i\})$ can be computed by efficient inner-product with $\nabla_\theta f$.

Computing Influence Function

As previously seen (Influence function)

Counterfactual prediction of removing z_i is $\Delta f = \tau_f(\{z_i\}) \approx \epsilon \cdot \mathcal{I}(z_i, f)$ with $\epsilon = -1/n$, where

$$\mathcal{I}(z_i, f) = -\nabla_\theta f(\hat{\theta})^\top H_{\hat{\theta}}^{-1} \nabla_\theta \ell_i, \quad H_{\hat{\theta}} = \frac{1}{n} \sum_{z_i \in D} \nabla_\theta^2 \ell_i$$

The main computation is the *inverse-Hessian-vector-product* $H_{\hat{\theta}}^{-1} \times \nabla_\theta \ell_i$, or iHVP:

Remark

Once iHVP is solved, $\tau_f(\{z_i\})$ can be computed by efficient inner-product with $\nabla_\theta f$.

- Vector $\nabla_\theta \ell_i \in \mathbb{R}^p$: first-order gradient for all $z_i \in D$

Computing Influence Function

As previously seen (Influence function)

Counterfactual prediction of removing z_i is $\Delta f = \tau_f(\{z_i\}) \approx \epsilon \cdot \mathcal{I}(z_i, f)$ with $\epsilon = -1/n$, where

$$\mathcal{I}(z_i, f) = -\nabla_\theta f(\hat{\theta})^\top H_{\hat{\theta}}^{-1} \nabla_\theta \ell_i, \quad H_{\hat{\theta}} = \frac{1}{n} \sum_{z_i \in D} \nabla_\theta^2 \ell_i$$

The main computation is the *inverse-Hessian-vector-product* $H_{\hat{\theta}}^{-1} \times \nabla_\theta \ell_i$, or iHVP:

Remark

Once iHVP is solved, $\tau_f(\{z_i\})$ can be computed by efficient inner-product with $\nabla_\theta f$.

- Vector $\nabla_\theta \ell_i \in \mathbb{R}^p$: first-order gradient for all $z_i \in D$
- Inverse-Hessian $H_{\hat{\theta}}^{-1} \in \mathbb{R}^{p \times p}$: inverting a $p \times p$ second-order Hessian

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$
- Computing product requires $O(np^2)$

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$
- Computing product requires $O(np^2)$

Next, the issue of ***storage***:

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$
- Computing product requires $O(np^2)$

Next, the issue of ***storage***:

- Storing all vectors $\{\nabla_\theta \ell_i \in \mathbb{R}^p\}_{i=1}^n$ requires $O(np)$.

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$
- Computing product requires $O(np^2)$

Next, the issue of ***storage***:

- Storing all vectors $\{\nabla_\theta \ell_i \in \mathbb{R}^p\}_{i=1}^n$ requires $O(np)$.
- Storing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2)$

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$
- Computing product requires $O(np^2)$

Next, the issue of ***storage***:

- Storing all vectors $\{\nabla_\theta \ell_i \in \mathbb{R}^p\}_{i=1}^n$ requires $O(np)$.
- Storing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2)$

Remark (Main bottleneck)

Respectively, the main bottlenecks are:

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$
- Computing product requires $O(np^2)$

Next, the issue of ***storage***:

- Storing all vectors $\{\nabla_\theta \ell_i \in \mathbb{R}^p\}_{i=1}^n$ requires $O(np)$.
- Storing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2)$

Remark (Main bottleneck)

Respectively, the main bottlenecks are:

- ***Computation****: inverse-Hessian* $O(p^3)$

Bottleneck of Naive iHVP

There are several bottlenecks for iHVP. First, the ***computation***:

- Computing all vectors $\{\nabla_\theta \ell_i\}_{i=1}^n$ requires $O(np)$
- Computing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2 + p^3) = O(p^3)$
- Computing product requires $O(np^2)$

Next, the issue of ***storage***:

- Storing all vectors $\{\nabla_\theta \ell_i \in \mathbb{R}^p\}_{i=1}^n$ requires $O(np)$.
- Storing inverse-Hessian $H_{\hat{\theta}}^{-1}$ requires $O(p^2)$

Remark (Main bottleneck)

Respectively, the main bottlenecks are:

- ***Computation***: *inverse-Hessian* $O(p^3)$
- ***Storage***: *vectors* + *inverse-Hessian* $O(np + p^2)$

Table of Content

Classical iHVP

iHVP is actually a general problem:

Classical iHVP

iHVP is actually a general problem:

- E.g., it appears in stochastic optimization (read: conditioned gradient)

Classical iHVP

iHVP is actually a general problem:

- E.g., it appears in stochastic optimization (read: conditioned gradient)
- Techniques to accelerate iHVP computation has been developed

Classical iHVP

iHVP is actually a general problem:

- E.g., it appears in stochastic optimization (read: conditioned gradient)
- Techniques to accelerate iHVP computation has been developed

Notably, these techniques aims to directly compute iHVP:

Classical iHVP

iHVP is actually a general problem:

- E.g., it appears in stochastic optimization (read: conditioned gradient)
- Techniques to accelerate iHVP computation has been developed

Notably, these techniques aims to directly compute iHVP:

- They require using the result of iHVP *literally*

Classical iHVP

iHVP is actually a general problem:

- E.g., it appears in stochastic optimization (read: conditioned gradient)
- Techniques to accelerate iHVP computation has been developed

Notably, these techniques aims to directly compute iHVP:

- They require using the result of iHVP *literally*
- LiSSA [ABH17], DataInf [Kwo+24]: avoiding performing large matrix inverse

Classical iHVP

iHVP is actually a general problem:

- E.g., it appears in stochastic optimization (read: conditioned gradient)
- Techniques to accelerate iHVP computation has been developed

Notably, these techniques aims to directly compute iHVP:

- They require using the result of iHVP *literally*
- LiSSA [ABH17], DataInf [Kwo+24]: avoiding performing large matrix inverse

However, they tend to be slow and can't be scaled up.

Classical iHVP

iHVP is actually a general problem:

- E.g., it appears in stochastic optimization (read: conditioned gradient)
- Techniques to accelerate iHVP computation has been developed

Notably, these techniques aims to directly compute iHVP:

- They require using the result of iHVP *literally*
- LiSSA [ABH17], DataInf [Kwo+24]: avoiding performing large matrix inverse

However, they tend to be slow and can't be scaled up.

Remark

iHVP in influence function specifically is different and orthogonal to above.

Table of Content

Scalable Approximation: FIM

To mitigate the bottleneck of inverse-Hessian:

Scalable Approximation: FIM

To mitigate the bottleneck of inverse-Hessian:

Theorem (Fisher information matrix)

For cross-entropy loss, in expectation, empirical fisher information matrix (FIM) $F_{\hat{\theta}}$ equals $H_{\hat{\theta}}$:

Scalable Approximation: FIM

To mitigate the bottleneck of inverse-Hessian:

Theorem (Fisher information matrix)

For cross-entropy loss, in expectation, empirical fisher information matrix (FIM) $F_{\hat{\theta}}$ equals $H_{\hat{\theta}}$:

$$F_{\hat{\theta}} := \frac{1}{n} \sum_{z_i \in D} \nabla_\theta \ell_i \nabla_\theta \ell_i^\top .$$

Scalable Approximation: FIM

To mitigate the bottleneck of inverse-Hessian:

Theorem (Fisher information matrix)

For cross-entropy loss, in expectation, empirical fisher information matrix (FIM) $F_{\hat{\theta}}$ equals $H_{\hat{\theta}}$:

$$F_{\hat{\theta}} := \frac{1}{n} \sum_{z_i \in D} \nabla_\theta \ell_i \nabla_\theta \ell_i^\top .$$

We see that using FIM approximation:

Scalable Approximation: FIM

To mitigate the bottleneck of inverse-Hessian:

Theorem (Fisher information matrix)

For cross-entropy loss, in expectation, empirical fisher information matrix (FIM) $F_{\hat{\theta}}$ equals $H_{\hat{\theta}}$:

$$F_{\hat{\theta}} := \frac{1}{n} \sum_{z_i \in D} \nabla_\theta \ell_i \nabla_\theta \ell_i^\top .$$

We see that using FIM approximation:

- Although no higher-order differentiation, computation changes from $O(p^2)$ to $O(np^2)$

Scalable Approximation: FIM

To mitigate the bottleneck of inverse-Hessian:

Theorem (Fisher information matrix)

For cross-entropy loss, in expectation, empirical fisher information matrix (FIM) $F_{\hat{\theta}}$ equals $H_{\hat{\theta}}$:

$$F_{\hat{\theta}} := \frac{1}{n} \sum_{z_i \in D} \nabla_\theta \ell_i \nabla_\theta \ell_i^\top .$$

We see that using FIM approximation:

- Although no higher-order differentiation, computation changes from $O(p^2)$ to $O(np^2)$
- Inverting still requires $O(p^3)$, as well as storage $O(p^2)$

Scalable Approximation: FIM

To mitigate the bottleneck of inverse-Hessian:

Theorem (Fisher information matrix)

For cross-entropy loss, in expectation, empirical fisher information matrix (FIM) $F_{\hat{\theta}}$ equals $H_{\hat{\theta}}$:

$$F_{\hat{\theta}} := \frac{1}{n} \sum_{z_i \in D} \nabla_\theta \ell_i \nabla_\theta \ell_i^\top .$$

We see that using FIM approximation:

- Although no higher-order differentiation, computation changes from $O(p^2)$ to $O(np^2)$
- Inverting still requires $O(p^3)$, as well as storage $O(p^2)$

Problem

Why is this helpful?

Scalable Approximation: Block-Diagonal FIM

To actually speed up inverse-Hessian, we *break* $F_{\hat{\theta}}$:

Scalable Approximation: Block-Diagonal FIM

To actually speed up inverse-Hessian, we *break* $F_{\hat{\theta}}$:

- **Structural assumption**: layers are *independent* $\Rightarrow$ $F_{\hat{\theta}}$ is *block-diagonal* (and hence $F_{\hat{\theta}}^{-1}$)

Scalable Approximation: Block-Diagonal FIM

To actually speed up inverse-Hessian, we *break* $F_{\hat{\theta}}$:

- **Structural assumption**: layers are *independent* $\Rightarrow$ $F_{\hat{\theta}}$ is *block-diagonal* (and hence $F_{\hat{\theta}}^{-1}$)
- Inverse and product can now be done *layer-wise*!

Scalable Approximation: Block-Diagonal FIM

To actually speed up inverse-Hessian, we *break* $F_{\hat{\theta}}$:

- **Structural assumption**: layers are *independent* $\Rightarrow$ $F_{\hat{\theta}}$ is *block-diagonal* (and hence $F_{\hat{\theta}}^{-1}$)
- Inverse and product can now be done *layer-wise*!

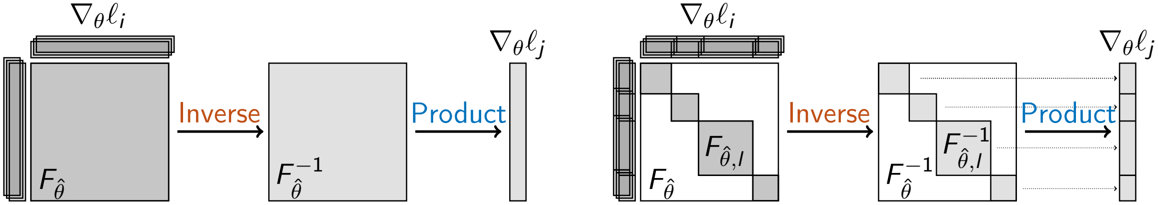
If you enjoy figures...

Scalable Approximation: Block-Diagonal FIM

To actually speed up inverse-Hessian, we *break* $F_{\hat{\theta}}$:

- **Structural assumption**: layers are *independent* $\Rightarrow$ $F_{\hat{\theta}}$ is *block-diagonal* (and hence $F_{\hat{\theta}}^{-1}$)
- Inverse and product can now be done *layer-wise*!

If you enjoy figures...



Remaining Bottlenecks

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

Remaining Bottlenecks

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors* + *inverse-FIM* + *product* $O(np + p^3/L^2 + np^2/L + np^2/L)$

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors* + *inverse-FIM* + *product* $O(np + p^3/L^2 + np^2/L + np^2/L)$
- ***Storage***: *vectors* + *inverse-FIM* $O(np + p^2/L)$

Remaining Bottlenecks

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors* + *inverse-FIM* + *product* $O(np + p^3/L^2 + np^2/L + np^2/L)$
- ***Storage***: *vectors* + *inverse-FIM* $O(np + p^2/L)$

Is this enough? Probably not since p is typically large:

Remaining Bottlenecks

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors* + *inverse-FIM* + *product* $O(np + p^3/L^2 + np^2/L + np^2/L)$
- ***Storage***: *vectors* + *inverse-FIM* $O(np + p^2/L)$

Is this enough? Probably not since p is typically large:

- Computation-wise, inverse-FIM takes $O(p^3/L^2)$.

Remaining Bottlenecks

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors* + *inverse-FIM* + *product* $O(np + p^3/L^2 + np^2/L + np^2/L)$
- ***Storage***: *vectors* + *inverse-FIM* $O(np + p^2/L)$

Is this enough? Probably not since p is typically large:

- Computation-wise, inverse-FIM takes $O(p^3/L^2)$.
- Storing vectors is challenging: $O(np)$ for 1B model with 1B dataset $\approx$ 4EB

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors + inverse-FIM + product* $O(np + p^3/L^2 + np^2/L + np^2/L)$
- ***Storage***: *vectors + inverse-FIM* $O(np + p^2/L)$

Is this enough? Probably not since p is typically large:

- Computation-wise, inverse-FIM takes $O(p^3/L^2)$.
- Storing vectors is challenging: $O(np)$ for 1B model with 1B dataset $\approx$ 4EB

The main bottleneck now becomes the large p for $\nabla_\theta \ell_i$:

Remaining Bottlenecks

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors + inverse-FIM + product* $O(np + p^3/L^2 + np^2/L + np^2/L)$
- ***Storage***: *vectors + inverse-FIM* $O(np + p^2/L)$

Is this enough? Probably not since p is typically large:

- Computation-wise, inverse-FIM takes $O(p^3/L^2)$.
- Storing vectors is challenging: $O(np)$ for 1B model with 1B dataset $\approx$ 4EB

The main bottleneck now becomes the large p for $\nabla_\theta \ell_i$:

- If we can operate with vectors of dimension $k \ll p$

Remark (Main bottleneck for block-diagonal FIM)

Say we have L layers. Respectively, the main bottlenecks are:

- ***Computation***: *vectors* + *inverse-FIM* + *product* $O(np + p^3/L^2 + np^2/L + np^2/L)$
- ***Storage***: *vectors* + *inverse-FIM* $O(np + p^2/L)$

Is this enough? Probably not since p is typically large:

- Computation-wise, inverse-FIM takes $O(p^3/L^2)$.
- Storing vectors is challenging: $O(np)$ for 1B model with 1B dataset $\approx$ 4EB

The main bottleneck now becomes the large p for $\nabla_\theta \ell_i$:

- If we can operate with vectors of dimension $k \ll p$

$\Rightarrow$ Replacing p with k everywhere (with some **computation** overhead...)

Table of Content

Intuition (Gradient Compression)

We can compress $g_i := \nabla_\theta \ell_i \in \mathbb{R}^p$ *down to* $\widetilde{g}_i \in \mathbb{R}^k$ *for some* $k \ll p$.

[2] In our case, we're considering more complicated operations. See discussion in [Sch+22].

Intuition (Gradient Compression)

We can compress $g_i := \nabla_\theta \ell_i \in \mathbb{R}^p$ *down to* $\widetilde{g}_i \in \mathbb{R}^k$ *for some* $k \ll p$.

The possibility of compression is motivated by the following:

[2] In our case, we're considering more complicated operations. See discussion in [Sch+22].

Intuition (Gradient Compression)

We can compress $g_i := \nabla_\theta \ell_i \in \mathbb{R}^p$ *down to* $\widetilde{g}_i \in \mathbb{R}^k$ *for some* $k \ll p$.

The possibility of compression is motivated by the following:

Theorem ((Informal) Johnson-Lindenstrauss Lemma)

Given n *vectors in* $\mathbb{R}^d$, *they can be projected to* $\mathbb{R}^k$ *with* $k = O(\frac{\log n}{\epsilon^2})$ *while approximately preserving pairwise distances and geometric structure.*

[2] In our case, we're considering more complicated operations. See discussion in [Sch+22].

Intuition (Gradient Compression)

We can compress $g_i := \nabla_\theta \ell_i \in \mathbb{R}^p$ *down to* $\widetilde{g}_i \in \mathbb{R}^k$ *for some* $k \ll p$.

The possibility of compression is motivated by the following:

Theorem ((Informal) Johnson-Lindenstrauss Lemma)

Given n *vectors in* $\mathbb{R}^d$, *they can be projected to* $\mathbb{R}^k$ *with* $k = O(\frac{\log n}{\epsilon^2})$ *while approximately preserving pairwise distances and geometric structure.*

This tells us that for simple operations (e.g., inner products):[2]

[2] In our case, we're considering more complicated operations. See discussion in [Sch+22].

Intuition (Gradient Compression)

We can compress $g_i := \nabla_\theta \ell_i \in \mathbb{R}^p$ *down to* $\widetilde{g}_i \in \mathbb{R}^k$ *for some* $k \ll p$.

The possibility of compression is motivated by the following:

Theorem ((Informal) Johnson-Lindenstrauss Lemma)

Given n *vectors in* $\mathbb{R}^d$, *they can be projected to* $\mathbb{R}^k$ *with* $k = O(\frac{\log n}{\epsilon^2})$ *while approximately preserving pairwise distances and geometric structure.*

This tells us that for simple operations (e.g., inner products):[2]

- Compression algorithms that admit JL guarantee can be integrated.

[2] In our case, we're considering more complicated operations. See discussion in [Sch+22].

Small Detour: Why Compression is New?

A natural question you now should have is:

Small Detour: Why Compression is New?

A natural question you now should have is:

Why can't we also apply gradient compression in, say, LiSSA?

Small Detour: Why Compression is New?

A natural question you now should have is:

Why can't we also apply gradient compression in, say, LiSSA?

The reason is the following:

Small Detour: Why Compression is New?

A natural question you now should have is:

Why can't we also apply gradient compression in, say, LiSSA?

The reason is the following:

- Previously, the application they consider *requires* iHVP (read: update parameters with conditioned gradient)

Small Detour: Why Compression is New?

A natural question you now should have is:

Why can't we also apply gradient compression in, say, LiSSA?

The reason is the following:

- Previously, the application they consider *requires* iHVP (read: update parameters with conditioned gradient)
- Now, in influence function computation, we take inner product between iHVP and ∇f

A natural question you now should have is:

Why can't we also apply gradient compression in, say, LiSSA?

The reason is the following:

- Previously, the application they consider *requires* iHVP (read: update parameters with conditioned gradient)
- Now, in influence function computation, we take inner product between iHVP and ∇f

Overall,

Small Detour: Why Compression is New?

A natural question you now should have is:

Why can't we also apply gradient compression in, say, LiSSA?

The reason is the following:

- Previously, the application they consider *requires* iHVP (read: update parameters with conditioned gradient)
- Now, in influence function computation, we take inner product between iHVP and ∇f

Overall,

- operating on smaller vectors makes no sense to optimization-related application;

Small Detour: Why Compression is New?

A natural question you now should have is:

Why can't we also apply gradient compression in, say, LiSSA?

The reason is the following:

- Previously, the application they consider *requires* iHVP (read: update parameters with conditioned gradient)
- Now, in influence function computation, we take inner product between iHVP and ∇f

Overall,

- operating on smaller vectors makes no sense to optimization-related application;
- but for us, we can also compress ∇f and take inner product without problems!

Example (Gaussian/Rademacher Projection (Random [Woj+16]))

Linear map induced by $P \in \mathbb{R}^{k \times p}$ with $P_{ij} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$ or $\mathcal{U}(\{\pm 1\})$ satisfies the JL lemma.

Example (Gaussian/Rademacher Projection (RANDOM [Woj+16]))

Linear map induced by $P \in \mathbb{R}^{k \times p}$ with $P_{ij} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 1)$ or $\mathcal{U}(\{\pm 1\})$ satisfies the JL lemma.

RANDOM states that to compress $g_{i,l}$, consider

$$\widetilde{g}_{i,l} = P^{(l)} \times g_{i,l}$$

for some projection matrix $P^{(l)} \in \mathbb{R}^{k/L \times p/L}$ that satisfies JL guarantee.

Random and its Computational Complexity

Example (Gaussian/Rademacher Projection (Random [Woj+16]))

Linear map induced by $P \in \mathbb{R}^{k \times p}$ with $P_{ij} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$ or $\mathcal{U}(\{\pm 1\})$ satisfies the JL lemma.

Random states that to compress $g_{i,l}$, consider

$$\widetilde{g}_{i,l} = P^{(l)} \times g_{i,l}$$

for some projection matrix $P^{(l)} \in \mathbb{R}^{k/L \times p/L}$ that satisfies JL guarantee.

- Projection time per $g_{i,l}$ is $O(kp/L^2)$.

Example (Gaussian/Rademacher Projection (Random [Woj+16]))

Linear map induced by $P \in \mathbb{R}^{k \times p}$ with $P_{ij} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0,1)$ or $\mathcal{U}(\{\pm 1\})$ satisfies the JL lemma.

Random states that to compress $g_{i,l}$, consider

$$\widetilde{g}_{i,l} = P^{(l)} \times g_{i,l}$$

for some projection matrix $P^{(l)} \in \mathbb{R}^{k/L \times p/L}$ that satisfies JL guarantee.

- Projection time per $g_{i,l}$ is $O(kp/L^2)$.

In total, for all data points and all layers, Random takes $O(npk/L)$.

To put everything together:

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

To put everything together:

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Putting Everything Together: Random

To put everything together:

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

Putting Everything Together: RANDOM

To put everything together:

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

- **Computation**: RANDOM with matrix multiplication implementation $O(npk/L)$
- **Storage**: compressed vectors $O(nk)$

Putting Everything Together: RANDOM

To put everything together:

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

- **Computation**: RANDOM with matrix multiplication implementation $O(npk/L)$
- **Storage**: compressed vectors $O(nk)$

Stage 2: Compute iFVP using $\widetilde{g}_i$:

Putting Everything Together: Random

To put everything together:

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

- **Computation**: Random with matrix multiplication implementation $O(npk/L)$
- **Storage**: compressed vectors $O(nk)$

Stage 2: Compute iFVP using $\widetilde{g}_i$:

- **Computation**: inverse-FIM + product $O(k^3/L^2 + nk^2/L)$

Putting Everything Together: RANDOM

To put everything together:

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

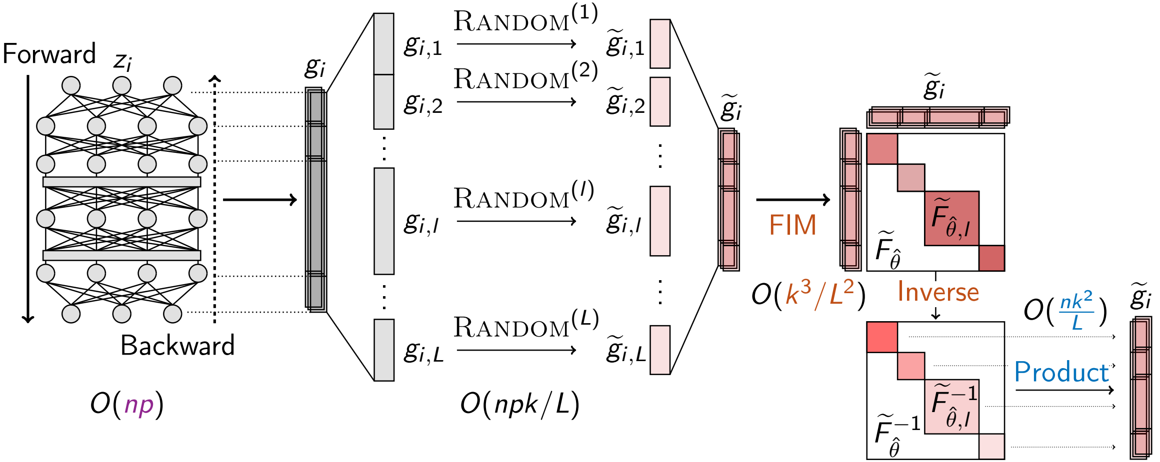
- **Computation**: RANDOM with matrix multiplication implementation $O(npk/L)$
- **Storage**: compressed vectors $O(nk)$

Stage 2: Compute iFVP using $\widetilde{g}_i$:

- **Computation**: inverse-FIM + product $O(k^3/L^2 + nk^2/L)$
- **Storage**: inverse-FIM $O(k^2/L)$

Putting Everything Together: Random

Stage 0	Stage 1	Stage 2
vectors	Compressed vectors	iFVP



Overhead of Gradient Compression

As previously seen (Computation Cost)

1. RANDOM *with matrix multiplication implementation* $O(npk/L)$

Overhead of Gradient Compression

As previously seen (Computation Cost)

1. RANDOM *with matrix multiplication implementation* $O(npk/L)$
2. *vectors* + *inverse-FIM* + *product* $O(np + k^3/L^2 + nk^2/L)$

Overhead of Gradient Compression

As previously seen (Computation Cost)

1. RANDOM *with matrix multiplication implementation* $O(npk/L)$
2. *vectors* + *inverse-FIM* + *product* $O(np + k^3/L^2 + nk^2/L)$

To provide some context:

Overhead of Gradient Compression

As previously seen (Computation Cost)

1. RANDOM *with matrix multiplication implementation* $O(npk/L)$
2. *vectors* + *inverse-FIM* + *product* $O(np + k^3/L^2 + nk^2/L)$

To provide some context:

- $O(np)$ for vectors is roughly *one training epoch*

Overhead of Gradient Compression

As previously seen (Computation Cost)

1. RANDOM *with matrix multiplication implementation* $O(npk/L)$
2. *vectors* + *inverse-FIM* + *product* $O(np + k^3/L^2 + nk^2/L)$

To provide some context:

- $O(np)$ for vectors is roughly *one training epoch*
- Per-layer projection dimension is typically $k/L \approx 4096$.

Overhead of Gradient Compression

As previously seen (Computation Cost)

1. RANDOM *with matrix multiplication implementation* $O(npk/L)$
2. *vectors* + *inverse-FIM* + *product* $O(np + k^3/L^2 + nk^2/L)$

To provide some context:

- $O(np)$ for vectors is roughly *one training epoch*
- Per-layer projection dimension is typically $k/L \approx 4096$.
- Overhead of RANDOM is 4096 more epochs of training

Overhead of Gradient Compression

As previously seen (Computation Cost)

1. Random *with matrix multiplication implementation* $O(npk/L)$
2. *vectors* + *inverse-FIM* + *product* $O(np + k^3/L^2 + nk^2/L)$

To provide some context:

- $O(np)$ for vectors is roughly *one training epoch*
- Per-layer projection dimension is typically $k/L \approx 4096$.
- Overhead of Random is 4096 more epochs of training

This is clearly infeasible.

Overhead of Gradient Compression

As previously seen (Computation Cost)

1. RANDOM *with matrix multiplication implementation* $O(npk/L)$
2. *vectors* + *inverse-FIM* + *product* $O(np + k^3/L^2 + nk^2/L)$

To provide some context:

- $O(np)$ for vectors is roughly *one training epoch*
- Per-layer projection dimension is typically $k/L \approx 4096$.
- Overhead of RANDOM is 4096 more epochs of training

This is clearly infeasible.

Problem

How to speed up the overhead of compression?

A natural idea is to search for faster compression algorithm:

[3] This is also used in TRAK's implementation (https://github.com/MadryLab/trak).

Fast Johnson-Lindenstrauss Transform

A natural idea is to search for faster compression algorithm:

- Compress vectors faster than matrix multiplication (i.e., RANDOM)

[3] This is also used in TRAK's implementation (https://github.com/MadryLab/trak).

A natural idea is to search for faster compression algorithm:

- Compress vectors faster than matrix multiplication (i.e., RANDOM)
- One alternative: *fast Johnson-Lindenstrauss transform*![3]

[3] This is also used in TRAK's implementation (https://github.com/MadryLab/trak).

A natural idea is to search for faster compression algorithm:

- Compress vectors faster than matrix multiplication (i.e., RANDOM)
- One alternative: *fast Johnson-Lindenstrauss transform*![3]

FJLT leverages discrete Fast Fourier Transform (FFT):

- Projection time per $g_{i,l}$ can be reduced from $O(kp/L^2)$ to $O(\frac{p+k}{L}\log p)$.

[3] This is also used in TRAK's implementation (https://github.com/MadryLab/trak).

Fast Johnson-Lindenstrauss Transform

A natural idea is to search for faster compression algorithm:

- Compress vectors faster than matrix multiplication (i.e., RANDOM)
- One alternative: *fast Johnson-Lindenstrauss transform*![3]

FJLT leverages discrete Fast Fourier Transform (FFT):

- Projection time per $g_{i,l}$ can be reduced from $O(kp/L^2)$ to $O(\frac{p+k}{L}\log p)$.

In total, for all data points and all layers, FJLT takes $O(n(p+k)\log p)$

Remark

It's roughly the same for one training epoch!

[3] This is also used in TRAK's implementation (https://github.com/MadryLab/trak).

Putting Everything Together: FJLT

Stage 0	Stage 1	Stage 2
vectors	Compressed vectors	iFVP

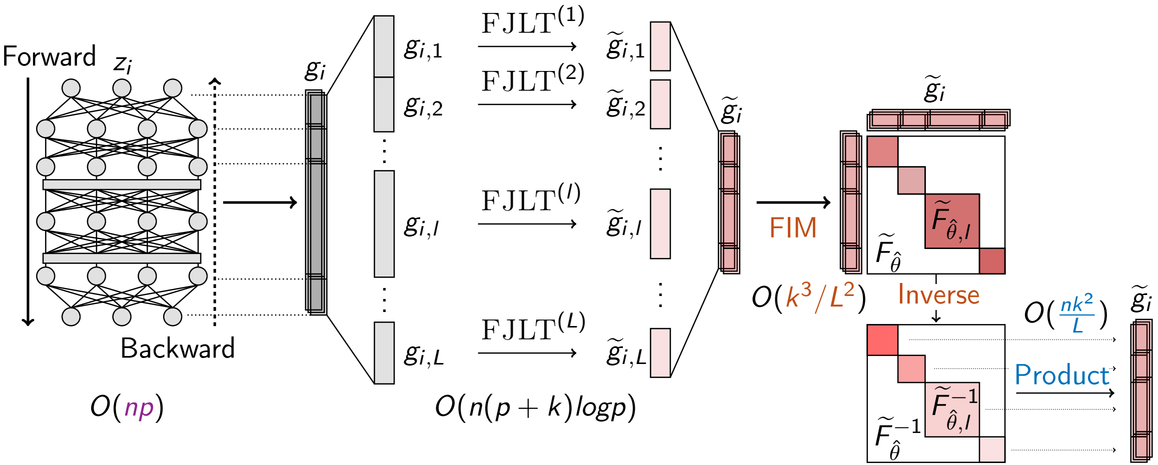


Table of Content

Investigating RANDOM

In RANDOM, with a Rademacher projection matrix $P^{(l)}$:

- **Dense Matrix**: Each entry of $P^{(l)}$ is sampled i.i.d. from $\mathcal{U}(\{\pm 1\})$.

Investigating RANDOM

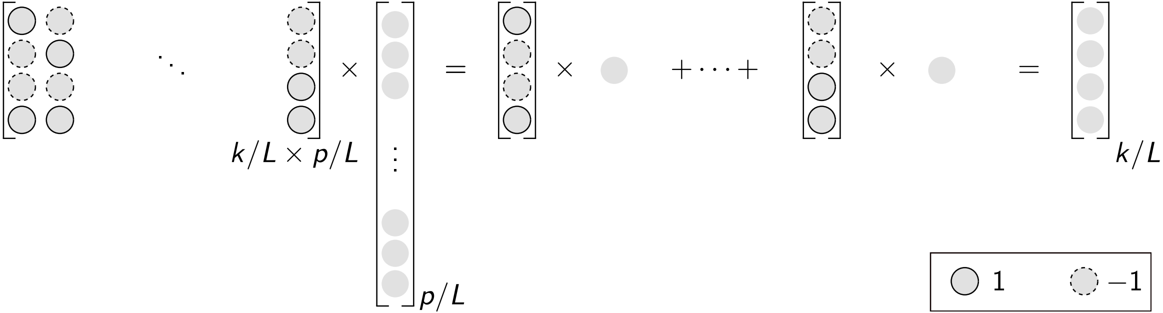
In RANDOM, with a Rademacher projection matrix $P^{(l)}$:

- **Dense Matrix**: Each entry of $P^{(l)}$ is sampled i.i.d. from $\mathcal{U}(\{\pm 1\})$.
- Matrix multiplication takes $O(kp/L^2)$ per $g_{i,l}$:

In RANDOM, with a Rademacher projection matrix $P^{(l)}$:

- **Dense Matrix**: Each entry of $P^{(l)}$ is sampled i.i.d. from $\mathcal{U}(\{\pm 1\})$.
- Matrix multiplication takes $O(kp/L^2)$ per $g_{i,l}$:

$$P^{(l)} \times g_{i,l} = P^{(l)}_{:\,1} \times (g_{i,l})_1 + \cdots + P^{(l)}_{:\,p/L} \times (g_{i,l})_{p/L} = \widetilde{g}_{i,l}$$



Sparse Johnson-Lindenstrauss transform [DKS10; KN14] considers a **sparser** $P^{(l)}$ instead:

- **Sparse Matrix**: For every column of $P^{(l)}$, only choose $s \ll k/L$ elements to be non-zero.

Sparser Johnson-Lindenstrauss Transform

Sparse Johnson-Lindenstrauss transform [DKS10; KN14] considers a **sparser** $P^{(l)}$ instead:

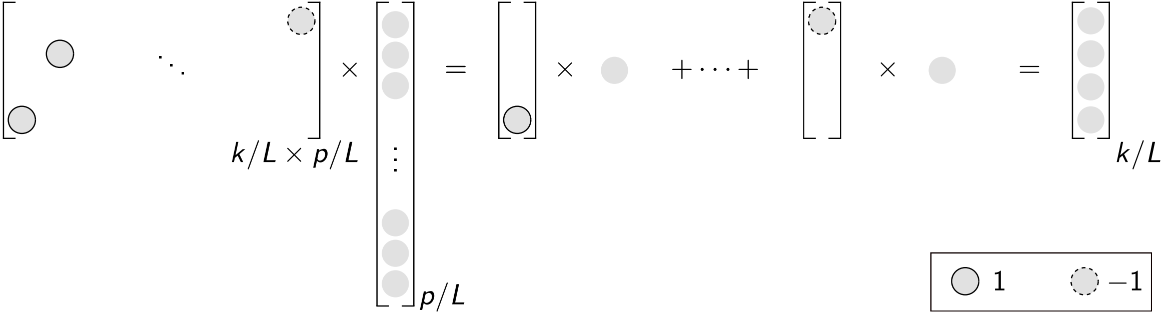
- **Sparse Matrix**: For every column of $P^{(l)}$, only choose $s \ll k/L$ elements to be non-zero.
- SJLT takes only $O(s \cdot p/L) = O(p/L)$ per $g_{i,l}$, *proportional* to **input size**.

Sparser Johnson-Lindenstrauss Transform

Sparse Johnson-Lindenstrauss transform [DKS10; KN14] considers a **sparser** $P^{(l)}$ instead:

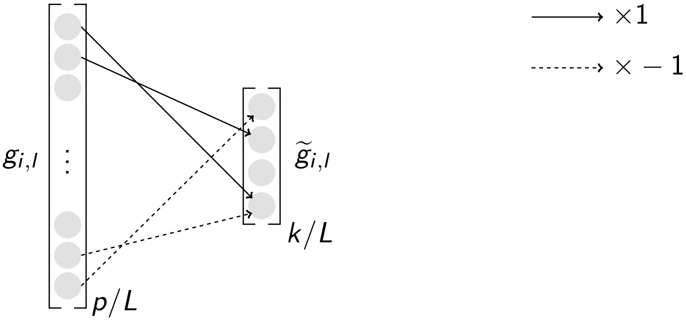
- **Sparse Matrix**: For every column of $P^{(l)}$, only choose $s \ll k/L$ elements to be non-zero.
- SJLT takes only $O(s \cdot p/L) = O(p/L)$ per $g_{i,l}$, *proportional* to **input size**.

$$P^{(l)} \times g_{i,l} = P^{(l)}_{:\,1} \times (g_{i,l})_1 + \cdots + P^{(l)}_{:\,p/L} \times (g_{i,l})_{p/L} = \widetilde{g}_{i,l}$$



SJLT: Alternative Viewpoint

Equivalently, you can think about SJLT as follows:



Intuition

For each entry of $g_{i,l}$, we select s entries in $\widetilde{g}_{i,l}$ to add on (or subtract from, depending on ± 1).

Computational Complexity of SJLT

SJLT only depends on input dimension p/L:

[4] https://github.com/TRAIS-Lab/sjlt/tree/main

Computational Complexity of SJLT

SJLT only depends on input dimension p/L:

- Per $g_{i,l}$ cost reduced from $O(\frac{p+k}{L}\log p)$ to $O(p/L)$:

[4] https://github.com/TRAIS-Lab/sjlt/tree/main

Computational Complexity of SJLT

SJLT only depends on input dimension p/L:

- Per $g_{i,l}$ cost reduced from $O(\frac{p+k}{L}\log p)$ to $O(p/L)$:
- In total, from $O(n(p+k)\log p)$ to $O(np)$.

[4] https://github.com/TRAIS-Lab/sjlt/tree/main

Computational Complexity of SJLT

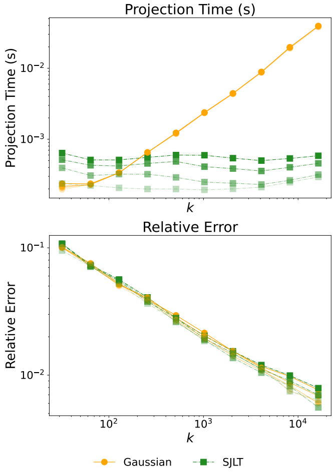
SJLT only depends on input dimension p/L:

- Per $g_{i,l}$ cost reduced from $O(\frac{p+k}{L}\log p)$ to $O(p/L)$:
- In total, from $O(n(p+k)\log p)$ to $O(np)$.

Remark (Potential speedup)

SJLT *exploits input sparsity, each runs only in* $O(\mathrm{nnz}(g_{i,l}))$.

- *Potentially,* SJLT *can run faster than* $O(np)$ *in total.*



$p = 131{,}072$ on several sparsity levels[4]

[4] https://github.com/TRAIS-Lab/sjlt/tree/main

Sub-Linear Compression

It seems like we can't go faster, as we need to read through the input at least?

Sub-Linear Compression

It seems like we can't go faster, as we need to read through the input at least?

- **Wrong**! We can throw out (some) information!

Sub-Linear Compression

It seems like we can't go faster, as we need to read through the input at least?

- **Wrong**! We can throw out (some) information!

Compression via selecting a few parameters ($\Leftrightarrow$ masking out most parameters):

Sub-Linear Compression

It seems like we can't go faster, as we need to read through the input at least?

- **Wrong**! We can throw out (some) information!

Compression via selecting a few parameters ($\Leftrightarrow$ masking out most parameters):

Intuition

Instead of "compress everything succinctly," we select a few parameters to look at.

Sub-Linear Compression

It seems like we can't go faster, as we need to read through the input at least?

- **Wrong**! We can throw out (some) information!

Compression via selecting a few parameters ($\Leftrightarrow$ masking out most parameters):

Intuition

Instead of "compress everything succinctly," we select a few parameters to look at.

- In the literature, people find out that only a few parameters are important for "inference"
- Idea of *localization* emerges [He+25; Yad+23; Wan+24].
- Used for task merging, sparsification, etc.

Mask

We call this Mask:

Mask

We call this Mask:

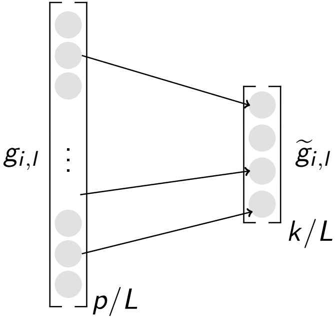
- By neglecting the information, we get a further speedup.

We call this MASK:

- By neglecting the information, we get a further speedup.
- MASK takes only $O(k/L)$ per $g_{i,l}$, *proportional* to **output size**.

We call this Mask:

- By neglecting the information, we get a further speedup.
- Mask takes only $O(k/L)$ per $g_{i,l}$, *proportional* to **output size**.



Computational Complexity of Mask

Mask only depends on output dimension k/L:

Computational Complexity of MASK

MASK only depends on output dimension k/L:

- Per $g_{i,l}$ cost reduced from $O(p/L)$ to $O(k/L)$:
- In total, from $O(np)$ to $O(nk)$.

Computational Complexity of Mask

Mask only depends on output dimension k/L:

- Per $g_{i,l}$ cost reduced from $O(p/L)$ to $O(k/L)$:
- In total, from $O(np)$ to $O(nk)$.

Remark

We finally achieve sub-linear *compression:*

Computational Complexity of MASK

MASK only depends on output dimension k/L:

- Per $g_{i,l}$ cost reduced from $O(p/L)$ to $O(k/L)$:
- In total, from $O(np)$ to $O(nk)$.

Remark

We finally achieve sub-linear compression:

- *To compress, we don't even need to read through all the input!*

Computational Complexity of MASK

MASK only depends on output dimension k/L:

- Per $g_{i,l}$ cost reduced from $O(p/L)$ to $O(k/L)$:
- In total, from $O(np)$ to $O(nk)$.

Remark

We finally achieve sub-linear compression:

- *To compress, we don't even need to read through all the input!*
- *Complexity is dominated by "outputting" the result.*

Computational Complexity of Mask

Mask only depends on output dimension k/L:

- Per $g_{i,l}$ cost reduced from $O(p/L)$ to $O(k/L)$:
- In total, from $O(np)$ to $O(nk)$.

Remark

We finally achieve sub-linear compression:

- *To compress, we don't even need to read through all the input!*
- *Complexity is dominated by "outputting" the result.*

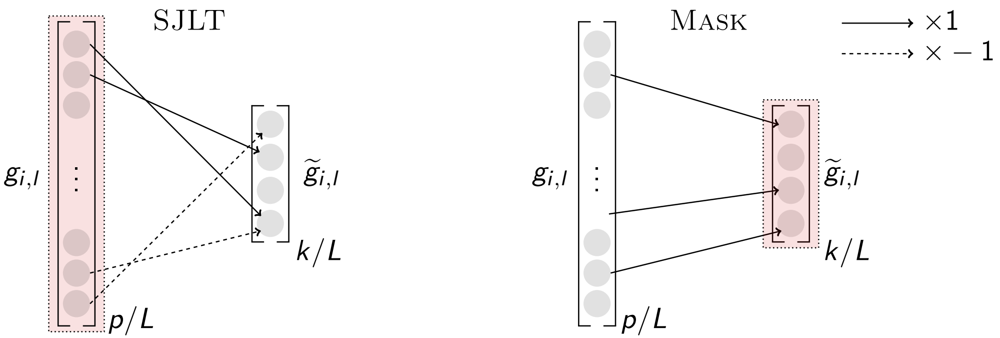
This complexity should now be impossible to beat.

Problem

In what cost?

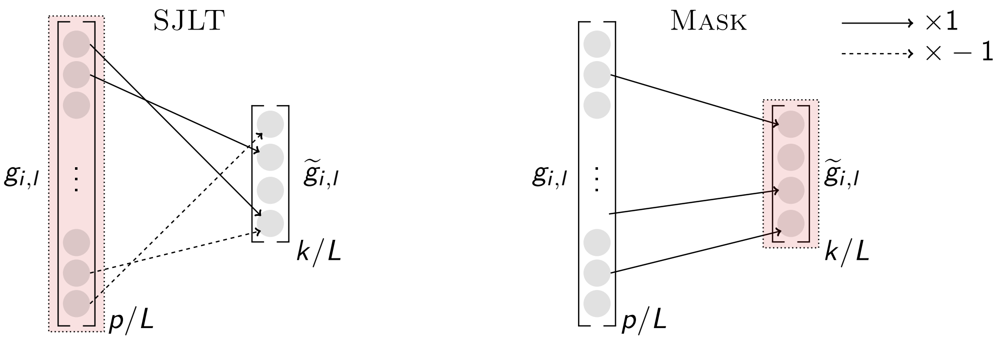
Situation Now

We now have two candidates, SJLT and MASK:



Situation Now

We now have two candidates, SJLT and MASK:

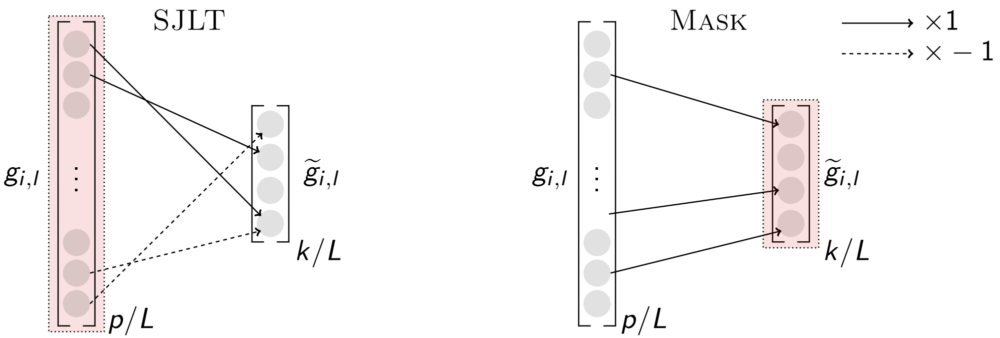


Problem (Pros and Cons)

- SJLT: *Very good compression guarantees, but cost $\propto$ input dimension.*

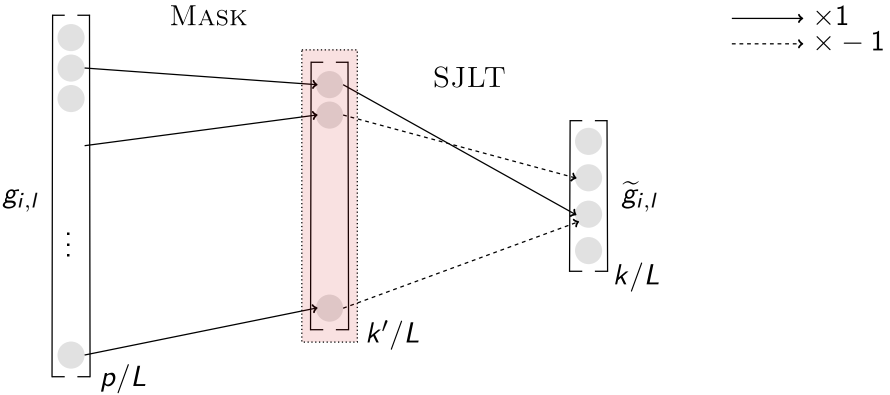
Situation Now

We now have two candidates, SJLT and MASK:



Problem (Pros and Cons)

- SJLT: *Very good compression guarantees, but cost $\propto$ input dimension.*
- MASK: *Extremely fast with cost $\propto$ output dimension, but will lose a lot of information.*



Intuition

First Mask *to a moderate dimension k'/L, then* SJLT *to the final dimension k/L!*

We term this method as GraSS: **Gra**dient **S**parsification and **S**parse projection.

Computational Complexity of GraSS

We term this method as GraSS: **Gra**dient **S**parsification and **S**parse projection.

- *Sparsification*: Mask to an intermediate dimension k'/L with $k < k' \ll p$
- *Sparse projection*: SJLT the sparsified vector of dimension k'/L down to k/L

Computational Complexity of GraSS

We term this method as GraSS: **Gra**dient **S**parsification and **S**parse projection.

- *Sparsification*: Mask to an intermediate dimension k'/L with $k < k' \ll p$
- *Sparse projection*: SJLT the sparsified vector of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

- Mask: cost $\propto$ output dimension, $O(k'/L)$

Computational Complexity of GRASS

We term this method as GRASS: **Gra**dient **S**parsification and **S**parse projection.

- *Sparsification*: MASK to an intermediate dimension k'/L with $k < k' \ll p$
- *Sparse projection*: SJLT the sparsified vector of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

- MASK: cost $\propto$ output dimension, $O(k'/L)$
- SJLT: cost $\propto$ input dimension, $O(k'/L)$

We term this method as GRASS: **Gra**dient **S**parsification and **S**parse projection.

- *Sparsification*: MASK to an intermediate dimension k'/L with $k < k' \ll p$
- *Sparse projection*: SJLT the sparsified vector of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

- MASK: cost $\propto$ output dimension, $O(k'/L)$
- SJLT: cost $\propto$ input dimension, $O(k'/L)$

$\Rightarrow$ Together takes $O(k'/L + k'/L) = O(k'/L)$

Computational Complexity of GraSS

We term this method as GraSS: **Gra**dient **S**parsification and **S**parse projection.

- *Sparsification*: Mask to an intermediate dimension k'/L with $k < k' \ll p$
- *Sparse projection*: SJLT the sparsified vector of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

- Mask: cost $\propto$ output dimension, $O(k'/L)$
- SJLT: cost $\propto$ input dimension, $O(k'/L)$

$\Rightarrow$ Together takes $O(k'/L + k'/L) = O(k'/L)$

In total, for all data points and all layers, GraSS takes $O(nk')$.

Putting Everything Together Again

Let's put everything together again, this time with GraSS.

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

Putting Everything Together Again

Let's put everything together again, this time with GRASS.

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Putting Everything Together Again

Let's put everything together again, this time with GRASS.

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

Putting Everything Together Again

Let's put everything together again, this time with GRASS.

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

- **Computation**: GRASS takes $O(nk')$ for some k' such that $k < k' \ll p$.
- **Storage**: compressed vectors $O(nk)$

Let's put everything together again, this time with GRASS.

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

- **Computation**: GRASS takes $O(nk')$ for some k' such that $k < k' \ll p$.
- **Storage**: compressed vectors $O(nk)$

Stage 2: Compute iFVP using $\widetilde{g}_i$:

Putting Everything Together Again

Let's put everything together again, this time with GraSS.

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

- **Computation**: GraSS takes $O(nk')$ for some k' such that $k < k' \ll p$.
- **Storage**: compressed vectors $O(nk)$

Stage 2: Compute iFVP using $\widetilde{g}_i$:

- **Computation**: inverse-FIM + product $O(k^3/L^2 + nk^2/L)$

Putting Everything Together Again

Let's put everything together again, this time with GRASS.

Stage 0: Compute all per-sample gradients $g_i \in \mathbb{R}^p$

- **Computation**: Forward/Backward passes for vectors $O(np)$
- **Storage**: None (immediately processed to next stage in memory)

Stage 1: Compressed $g_{i,l} \in \mathbb{R}^{p/L}$ down to $\widetilde{g}_{i,l} \in \mathbb{R}^{k/L}$, giving $\widetilde{g}_i \in \mathbb{R}^k$.

- **Computation**: GRASS takes $O(nk')$ for some k' such that $k < k' \ll p$.
- **Storage**: compressed vectors $O(nk)$

Stage 2: Compute iFVP using $\widetilde{g}_i$:

- **Computation**: inverse-FIM + product $O(k^3/L^2 + nk^2/L)$
- **Storage**: inverse-FIM $O(k^2/L)$

Stage 0	Stage 1	Stage 2
vectors	Compressed vectors	iFVP

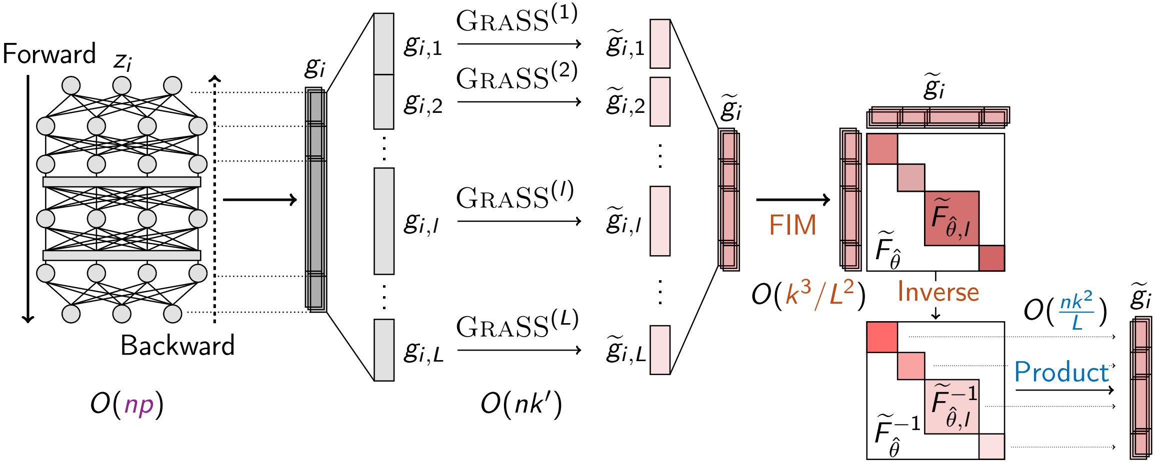


Table of Content

Why Linear Layers?

In modern model architectures:

Why Linear Layers?

In modern model architectures:

- Linear layers usually contain most of the parameters (since it is dense)

Why Linear Layers?

In modern model architectures:

- Linear layers usually contain most of the parameters (since it is dense)
- Gradient of linear layers has nice structures

Why Linear Layers?

In modern model architectures:

- Linear layers usually contain most of the parameters (since it is dense)
- Gradient of linear layers has nice structures

Due to the above, many have looked into accelerating linear layers in particular:

Why Linear Layers?

In modern model architectures:

- Linear layers usually contain most of the parameters (since it is dense)
- Gradient of linear layers has nice structures

Due to the above, many have looked into accelerating linear layers in particular:

- K-FAC [MG15], EK-FAC [Gro+23]: factorized FIM computation

Why Linear Layers?

In modern model architectures:

- Linear layers usually contain most of the parameters (since it is dense)
- Gradient of linear layers has nice structures

Due to the above, many have looked into accelerating linear layers in particular:

- K-FAC [MG15], EK-FAC [Gro+23]: factorized FIM computation
- Ghost Inner Product [Wan+25a]: allowing "batched" per-sample gradient computation

Why Linear Layers?

In modern model architectures:

- Linear layers usually contain most of the parameters (since it is dense)
- Gradient of linear layers has nice structures

Due to the above, many have looked into accelerating linear layers in particular:

- K-FAC [MG15], EK-FAC [Gro+23]: factorized FIM computation
- Ghost Inner Product [Wan+25a]: allowing "batched" per-sample gradient computation

We will see their fundamental ideas next. Let's first recall some basic facts about linear layers.

Gradient of **One** Linear Layer

We now take a closer look at linear layers.

Gradient of **One** Linear Layer

We now take a closer look at linear layers.

- Consider a model with only one linear layer (i.e., logistic regression)
- Let the weight be W, with activation $\sigma(\cdot)$

Gradient of **One** Linear Layer

We now take a closer look at linear layers.

- Consider a model with only one linear layer (i.e., logistic regression)
- Let the weight be W, with activation $\sigma(\cdot)$

The forward pass is:

$$z_i^{\text{out}} = W \cdot z_i, \quad z_i^{\text{pred}} = \sigma(z_i^{\text{out}})$$

Gradient of **One** Linear Layer

We now take a closer look at linear layers.

- Consider a model with only one linear layer (i.e., logistic regression)
- Let the weight be W, with activation $\sigma(\cdot)$

The forward pass is:

$$z_i^{\text{out}} = W \cdot z_i, \quad z_i^{\text{pred}} = \sigma(z_i^{\text{out}})$$

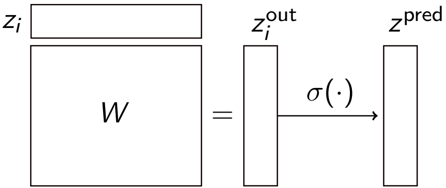
From chain rule, the backward pass is

$$\frac{\partial \ell_i}{\partial z_i^{\text{out}}} = \frac{\partial \ell_i}{\partial z_i^{\text{pred}}} \odot \frac{\partial z_i^{\text{pred}}}{\partial z_i^{\text{out}}} = \frac{\partial \ell_i}{\partial z_i^{\text{pred}}} \odot \sigma'(z_i^{\text{out}}), \quad \frac{\partial \ell_i}{\partial z_i} = W^\top \frac{\partial \ell_i}{\partial z_i^{\text{out}}}$$

Gradient of **One** Linear Layer

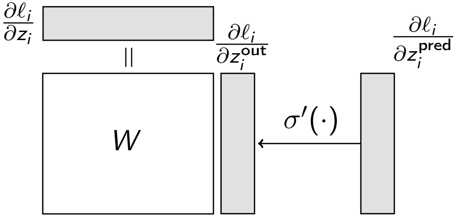
Forward Pass

$$z_i^{\text{out}} = W \cdot z_i, \quad z_i^{\text{pred}} = \sigma(z_i^{\text{out}})$$



Backward Pass

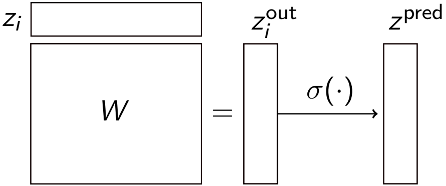
$$\frac{\partial \ell_i}{\partial z_i^{\text{out}}} = \frac{\partial \ell_i}{\partial z_i^{\text{pred}}} \odot \sigma'(z_i^{\text{out}}), \quad \frac{\partial \ell_i}{\partial z_i} = W^\top \frac{\partial \ell_i}{\partial z_i^{\text{out}}}$$



Gradient of **One** Linear Layer

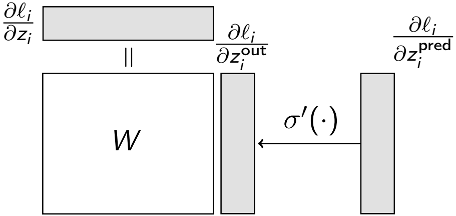
Forward Pass

$$z_i^{\text{out}} = W \cdot z_i, \quad z_i^{\text{pred}} = \sigma(z_i^{\text{out}})$$



Backward Pass

$$\frac{\partial \ell_i}{\partial z_i^{\text{out}}} = \frac{\partial \ell_i}{\partial z_i^{\text{pred}}} \odot \sigma'(z_i^{\text{out}}), \quad \frac{\partial \ell_i}{\partial z_i} = W^\top \frac{\partial \ell_i}{\partial z_i^{\text{out}}}$$



Remark

What we actually want is g_i:

$$g_i = \frac{\partial \ell_i}{\partial W} = \frac{\partial \ell_i}{\partial z_i^{out}} \frac{\partial z_i^{out}}{\partial W} = z_i \otimes \frac{\partial \ell_i}{\partial z_i^{out}}$$

Gradient of **An** Linear Layer

Now, let's consider linear layers in a deeper model:

Gradient of **An** Linear Layer

Now, let's consider linear layers in a deeper model:

- Consider a model with L linear layers (i.e., deep MLP)
- For the l^{th} linear layer, let the weight be W_l with activation $\sigma(\cdot)$

Gradient of **An** Linear Layer

Now, let's consider linear layers in a deeper model:

- Consider a model with L linear layers (i.e., deep MLP)
- For the l^{th} linear layer, let the weight be W_l with activation $\sigma(\cdot)$

The forward pass is

$$z_{i,l}^{\text{out}} = W_l \cdot z_{i,l}^{\text{in}}, \quad z_{i,l+1}^{\text{in}} = \sigma(z_{i,l}^{\text{out}})$$

Gradient of **An** Linear Layer

Now, let's consider linear layers in a deeper model:

- Consider a model with L linear layers (i.e., deep MLP)
- For the l^{th} linear layer, let the weight be W_l with activation $\sigma(\cdot)$

The forward pass is

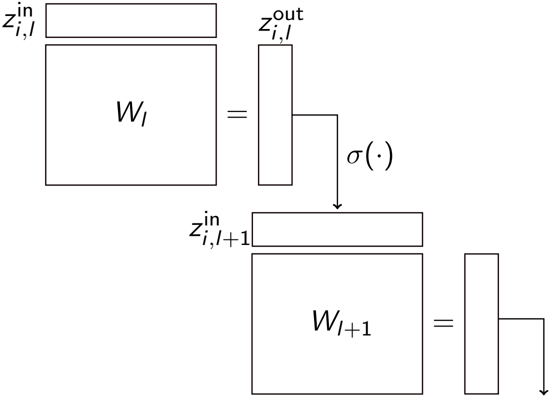
$$z_{i,l}^{\text{out}} = W_l \cdot z_{i,l}^{\text{in}}, \quad z_{i,l+1}^{\text{in}} = \sigma(z_{i,l}^{\text{out}})$$

From the chain rule, the backward pass is

$$\frac{\partial \ell_i}{\partial z_{i,l}^{\text{out}}} = \frac{\partial \ell_i}{\partial z_{i,l+1}^{\text{in}}} \odot \frac{\partial z_{i,l+1}^{\text{in}}}{\partial z_{i,l}^{\text{out}}} = \frac{\partial \ell_i}{\partial z_{i,l+1}^{\text{in}}} \odot \sigma'(z_{i,l}^{\text{out}}), \quad \frac{\partial \ell_i}{\partial z_{i,l}^{\text{in}}} = W_l^{\top} \frac{\partial \ell_i}{\partial z_{i,l}^{\text{out}}}$$

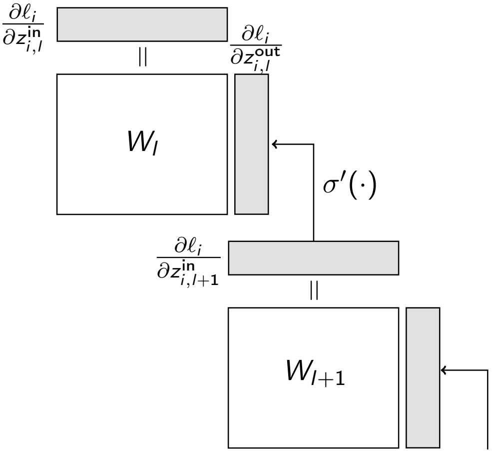
Forward Pass

$$z_{i,l}^{\text{out}} = W_l \cdot z_{i,l}^{\text{in}}, \quad z_{i,l+1}^{\text{in}} = \sigma(z_{i,l}^{\text{out}})$$



Backward Pass

$$\frac{\partial \ell_i}{\partial z_{i,l}^{\text{out}}} = \frac{\partial \ell_i}{\partial z_{i,l+1}^{\text{in}}} \odot \sigma'(z_{i,l}^{\text{out}}), \quad \frac{\partial \ell_i}{\partial z_{i,l}^{\text{in}}} = W_l^\top \frac{\partial \ell_i}{\partial z_{i,l}^{\text{out}}}$$



Materializing Gradients

Remark

What we actually want:

$$g_{i,l} = \frac{\partial \ell_i}{\partial W_l} = \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \frac{\partial z_{i,l}^{out}}{\partial W_l} = z_{i,l}^{in} \otimes \frac{\partial \ell_i}{\partial z_{i,l}^{out}}$$

Remark

What we actually want:

$$g_{i,l} = \frac{\partial \ell_i}{\partial W_l} = \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \frac{\partial z_{i,l}^{out}}{\partial W_l} = z_{i,l}^{in} \otimes \frac{\partial \ell_i}{\partial z_{i,l}^{out}}$$

We should now see the problem:

Problem

In the computational graph, we never materialize $g_{i,l}$.

Materializing Gradients

Remark

What we actually want:

$$g_{i,l} = \frac{\partial \ell_i}{\partial W_l} = \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \frac{\partial z_{i,l}^{out}}{\partial W_l} = z_{i,l}^{in} \otimes \frac{\partial \ell_i}{\partial z_{i,l}^{out}}$$

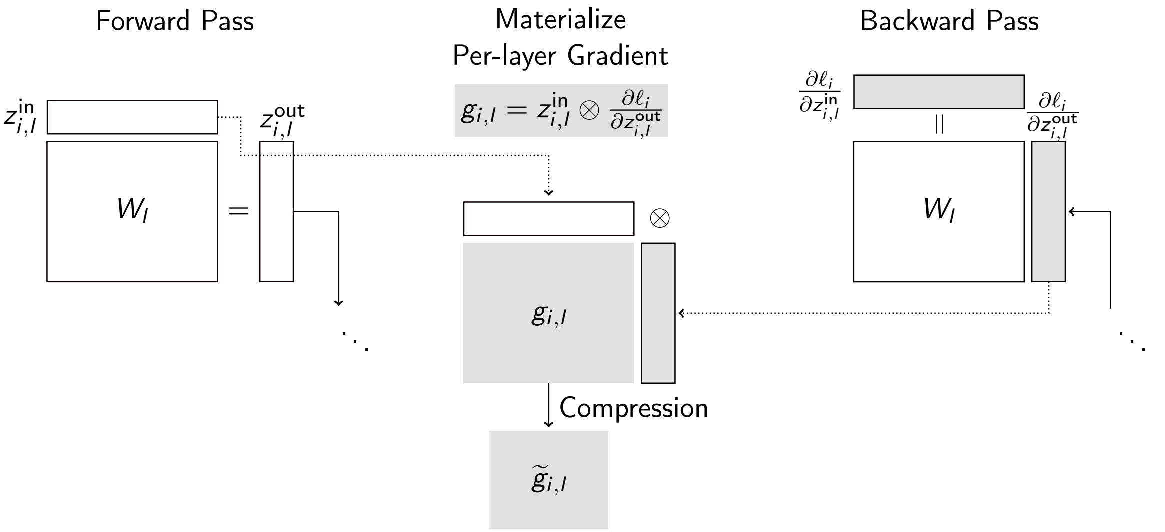
We should now see the problem:

Problem

In the computational graph, we never materialize $g_{i,l}$.

Hence, our previous analysis neglects the cost of computing $g_{i,l}$!

Materializing Gradients



Cost of Materializing Gradients

Assuming W_l is roughly square:

Cost of Materializing Gradients

Assuming W_l is roughly square:

- Both $z_{i,l}^{\text{in}}$ and $\partial \ell_i / \partial z_{i,l}^{\text{out}}$ are roughly of dimension $\sqrt{p/L}$
- $z_{i,l}^{\text{in}} \otimes {}^{\partial \ell_i}/_{\partial z_{i,l}^{\text{out}}}$ costs $O(\sqrt{p/L}^2) = O(p/L)$
- Overall, it'll take $O(np)$...

Cost of Materializing Gradients

Assuming W_l is roughly square:

- Both $z_{i,l}^{\text{in}}$ and $\partial \ell_i / \partial z_{i,l}^{\text{out}}$ are roughly of dimension $\sqrt{p/L}$
- $z_{i,l}^{\text{in}} \otimes {}^{\partial \ell_i}/_{\partial z_{i,l}^{\text{out}}}$ costs $O(\sqrt{p/L}^2) = O(p/L)$
- Overall, it'll take $O(np)$...

Remark

Even if GRASS *takes only $O(nk') \ll O(np)$, once we materialize $g_{i,l}$, it'll take $O(np)$.*

Assuming W_l is roughly square:

- Both $z_{i,l}^{\text{in}}$ and $\partial \ell_i / \partial z_{i,l}^{\text{out}}$ are roughly of dimension $\sqrt{p/L}$
- $z_{i,l}^{\text{in}} \otimes {}^{\partial \ell_i}/_{\partial z_{i,l}^{\text{out}}}$ costs $O(\sqrt{p/L}^2) = O(p/L)$
- Overall, it'll take $O(np)$...

Remark

Even if GRASS *takes only* $O(nk') \ll O(np)$, *once we materialize* $g_{i,l}$, *it'll take* $O(np)$.

However, is this really a concern?

Cost of Materializing Gradients

Assuming W_l is roughly square:

- Both $z_{i,l}^{\text{in}}$ and $\partial \ell_i / \partial z_{i,l}^{\text{out}}$ are roughly of dimension $\sqrt{p/L}$
- $z_{i,l}^{\text{in}} \otimes \partial \ell_i / \partial z_{i,l}^{\text{out}}$ costs $O(\sqrt{p/L}^2) = O(p/L)$
- Overall, it'll take $O(np)$...

Remark

Even if GRASS *takes only* $O(nk') \ll O(np)$, *once we materialize* $g_{i,l}$, *it'll take* $O(np)$.

However, is this really a concern?

- I mean, how can you compress $g_{i,l}$ without materializing it?
- Seems like this $O(np)$ cost will lay in the background and we can't get rid of?

Putting Everything Together for Linear Layers

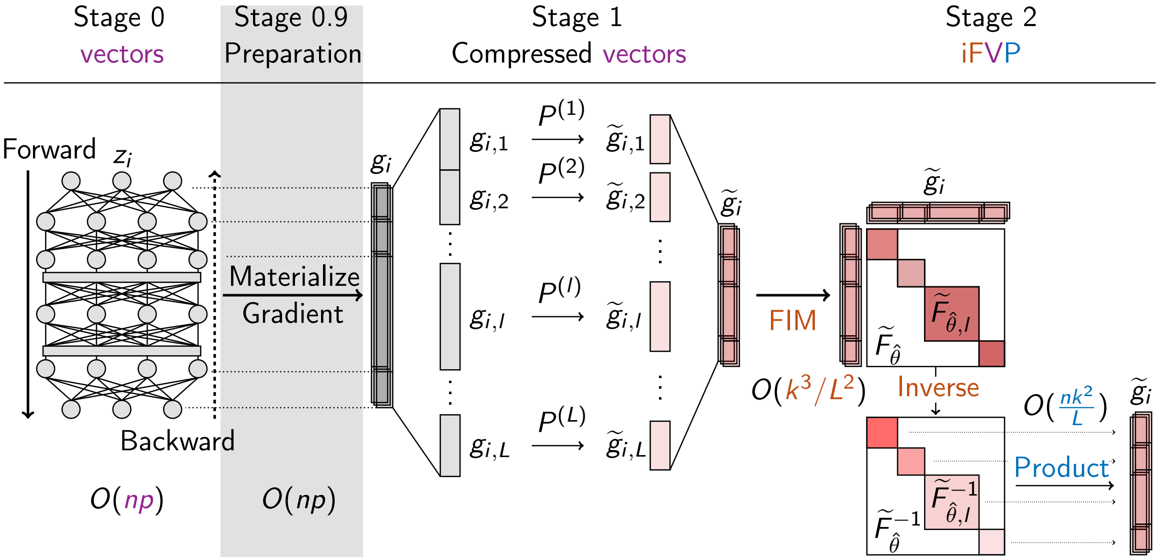


Table of Content

Sadly, the reality is always harsh:

[5] It is worth noting that from [Wan+25a], the calculation can even be batched.

Sadly, the reality is always harsh:

Theorem (LoGra)

There is a gradient compression algorithm that ***does not*** *require materializing* $g_{i,l}$ *(for MLP layer).*[5]

[5] It is worth noting that from [Wan+25a], the calculation can even be batched.

Sadly, the reality is always harsh:

Theorem (LoGra)

*There is a gradient compression algorithm that **does not** require materializing $g_{i,l}$ (for MLP layer).*[5]

Intuition

To compress $g_{i,l}$, just compress the components individually:

[5] It is worth noting that from [Wan+25a], the calculation can even be batched.

Sadly, the reality is always harsh:

Theorem (LoGra)

*There is a gradient compression algorithm that **does not** require materializing $g_{i,l}$ (for MLP layer).*[5]

Intuition

To compress $g_{i,l}$, just compress the components individually:

$$P^{(l)} g_{i,l} := (P_{in}^{(l)} \otimes P_{out}^{(l)}) \cdot \left(z_{i,l}^{in} \otimes \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \right) = (P_{in}^{(l)} \cdot z_{i,l}^{in}) \otimes \left(P_{out}^{(l)} \cdot \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \right)$$

[5] It is worth noting that from [Wan+25a], the calculation can even be batched.

Final Boss: LoGra

Sadly, the reality is always harsh:

Theorem (LoGra)

*There is a gradient compression algorithm that **does not** require materializing $g_{i,l}$ (for MLP layer).*[5]

Intuition

To compress $g_{i,l}$, just compress the components individually:

$$P^{(l)} g_{i,l} := (P_{in}^{(l)} \otimes P_{out}^{(l)}) \cdot \left(z_{i,l}^{in} \otimes \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \right) = (P_{in}^{(l)} \cdot z_{i,l}^{in}) \otimes \left(P_{out}^{(l)} \cdot \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \right)$$

- *Allocating k/L equally $\Rightarrow$ target dimension for both is $\sqrt{k/L}$*

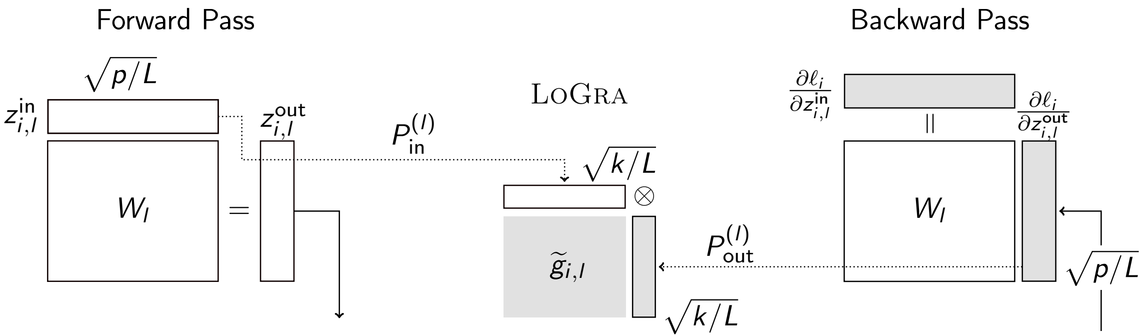
[5] It is worth noting that from [Wan+25a], the calculation can even be batched.

As previously seen (LoGra)

$$\widetilde{g}_{i,l} = P^{(l)} g_{i,l} = (P_{in}^{(l)} \cdot z_{i,l}^{in}) \otimes \left(P_{out}^{(l)} \cdot \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \right)$$

As previously seen (LoGra)

$$\widetilde{g}_{i,l} = P^{(l)} g_{i,l} = (P_{in}^{(l)} \cdot z_{i,l}^{in}) \otimes \left(P_{out}^{(l)} \cdot \frac{\partial \ell_i}{\partial z_{i,l}^{out}} \right)$$



We see that for a linear layer l:

- By assuming $P^{(l)} = P_{\text{in}}^{(l)} \otimes P_{\text{out}}^{(l)}$, we "decompose" the projection

Computational Complexity of LoGra

We see that for a linear layer l:

- By assuming $P^{(l)} = P_{\text{in}}^{(l)} \otimes P_{\text{out}}^{(l)}$, we "decompose" the projection
- Let $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ can be any compression algorithm

Computational Complexity of LoGra

We see that for a linear layer l:

- By assuming $P^{(l)} = P_{\text{in}}^{(l)} \otimes P_{\text{out}}^{(l)}$, we "decompose" the projection
- Let $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ can be any compression algorithm

Say both $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ are the simple Random:

- $P_{\text{in}}^{(l)} z_{i,l}^{\text{in}}$ and $P_{\text{out}}^{(l)} \partial \ell_i / \partial z_{i,l}^{\text{out}}$ both takes $O(\sqrt{kp}/L)$

Computational Complexity of LoGra

We see that for a linear layer l:

- By assuming $P^{(l)} = P_{\text{in}}^{(l)} \otimes P_{\text{out}}^{(l)}$, we "decompose" the projection
- Let $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ can be any compression algorithm

Say both $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ are the simple Random:

- $P_{\text{in}}^{(l)} z_{i,l}^{\text{in}}$ and $P_{\text{out}}^{(l)} \partial \ell_i / \partial z_{i,l}^{\text{out}}$ both takes $O(\sqrt{kp}/L)$
- Reconstructing $\widetilde{g}_{i,l}$ via $\otimes$ takes only $O(k/L)$

Computational Complexity of LoGra

We see that for a linear layer l:

- By assuming $P^{(l)} = P_{\text{in}}^{(l)} \otimes P_{\text{out}}^{(l)}$, we "decompose" the projection
- Let $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ can be any compression algorithm

Say both $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ are the simple Random:

- $P_{\text{in}}^{(l)} z_{i,l}^{\text{in}}$ and $P_{\text{out}}^{(l)} \partial \ell_i / \partial z_{i,l}^{\text{out}}$ both takes $O(\sqrt{kp}/L)$
- Reconstructing $\widetilde{g}_{i,l}$ via $\otimes$ takes only $O(k/L)$
- Per $g_{i,l}$ cost hence is $O(\sqrt{kp}/L + k/L) = O(\sqrt{kp}/L)$

Computational Complexity of LoGra

We see that for a linear layer l:

- By assuming $P^{(l)} = P_{\text{in}}^{(l)} \otimes P_{\text{out}}^{(l)}$, we "decompose" the projection
- Let $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ can be any compression algorithm

Say both $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ are the simple Random:

- $P_{\text{in}}^{(l)} z_{i,l}^{\text{in}}$ and $P_{\text{out}}^{(l)} {}^{\partial \ell_i}/_{\partial z_{i,l}^{\text{out}}}$ both takes $O(\sqrt{kp}/L)$
- Reconstructing $\widetilde{g}_{i,l}$ via $\otimes$ takes only $O(k/L)$
- Per $g_{i,l}$ cost hence is $O(\sqrt{kp}/L + k/L) = O(\sqrt{kp}/L)$

Overall, LoGra only takes $O(n\sqrt{kp}) < O(np)$

Putting Everything Together: LoGra

Stage 0	Stage 0.9	Stage 1	Stage 2
vectors	Preparation	Compressed vectors	iFVP

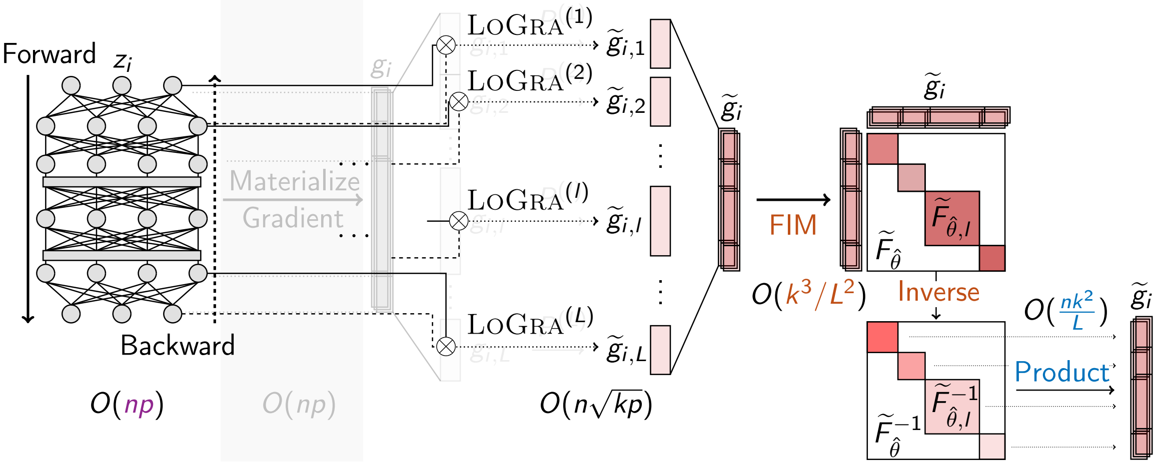


Table of Content

Now What?

Let's summarize the situation a bit. For *general layers*:

- GRASS takes $O(np) + O(nk')$ considering the cost of materializing g_i

$\Rightarrow$ Fastest gradient compression algorithm so far

Let's summarize the situation a bit. For *general layers*:

- GRASS takes $O(np) + O(nk')$ considering the cost of materializing g_i

$\Rightarrow$ Fastest gradient compression algorithm so far

However, for *linear layers*:

- GRASS takes $O(np) + O(nk')$, considering the cost of materializing g_i
- LOGRA takes $O(n\sqrt{kp})$, without materializing g_i

$\Rightarrow$ LOGRA beats GRASS **by a lot**

Now What?

Let's summarize the situation a bit. For *general layers*:

- GraSS takes $O(np) + O(nk')$ considering the cost of materializing g_i

$\Rightarrow$ Fastest gradient compression algorithm so far

However, for *linear layers*:

- GraSS takes $O(np) + O(nk')$, considering the cost of materializing g_i
- LoGra takes $O(n\sqrt{kp})$, without materializing g_i

$\Rightarrow$ LoGra beats GraSS **by a lot**

Problem

How to beat LoGra*?*

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

- Theoretically, sure! In practice, no.

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

- Theoretically, sure! In practice, no.

Problem

Two projection problems are too small ($\sqrt{p/L} \to \sqrt{k/L}$*, e.g.,* $4096 \to 64$)*:*

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

- Theoretically, sure! In practice, no.

Problem

Two projection problems are too small ($\sqrt{p/L} \to \sqrt{k/L}$, e.g., $4096 \to 64$):

- RANDOM *(i.e., matrix multiplication) is extremely fast (`PyTorch` low-level optimization)*

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

- Theoretically, sure! In practice, no.

Problem

Two projection problems are too small ($\sqrt{p/L} \to \sqrt{k/L}$, e.g., $4096 \to 64$):

- RANDOM *(i.e., matrix multiplication) is extremely fast (`PyTorch` low-level optimization)*

MASK *is still efficient, problem lies in* SJLT*'s practical implementation:*

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

- Theoretically, sure! In practice, no.

Problem

Two projection problems are too small ($\sqrt{p/L} \to \sqrt{k/L}$, e.g., $4096 \to 64$):

- RANDOM *(i.e., matrix multiplication) is extremely fast (*`PyTorch` *low-level optimization)*

MASK *is still efficient, problem lies in* SJLT*'s practical implementation:*

- ***Overhead****: small problem size suffer...*

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

- Theoretically, sure! In practice, no.

Problem

Two projection problems are too small ($\sqrt{p/L} \to \sqrt{k/L}$, e.g., $4096 \to 64$):

- RANDOM *(i.e., matrix multiplication) is extremely fast (`PyTorch` low-level optimization)*

MASK *is still efficient, problem lies in* SJLT*'s practical implementation:*

- ***Overhead**: small problem size suffer...*
- ***Hash Collision**: even slower on small dimensions than on moderate dimensions*

Naive Approach

A naive idea is to simply replace $P_{\text{in}}^{(l)}$ and $P_{\text{out}}^{(l)}$ with GRASS!

- Theoretically, sure! In practice, no.

Problem

Two projection problems are too small ($\sqrt{p/L} \to \sqrt{k/L}$, e.g., $4096 \to 64$):

- RANDOM *(i.e., matrix multiplication) is extremely fast (*`PyTorch` *low-level optimization)*

MASK *is still efficient, problem lies in* SJLT*'s practical implementation:*

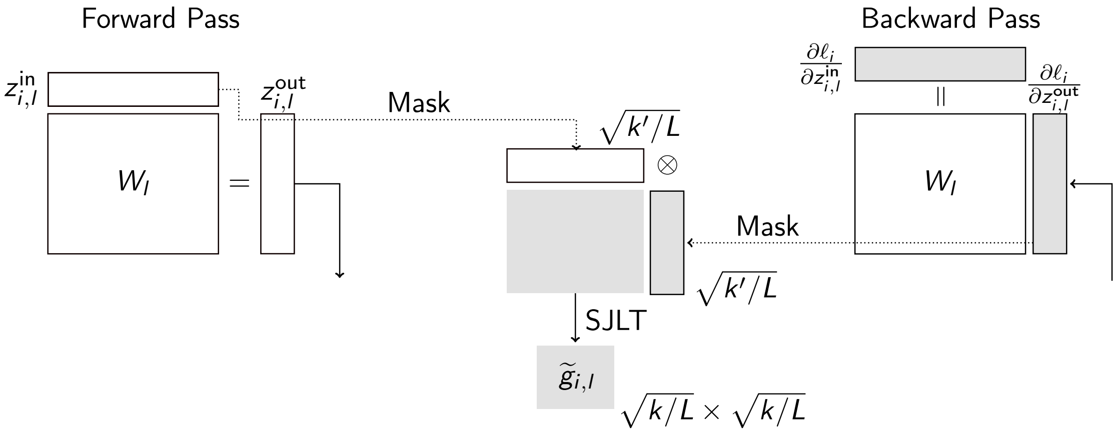
- ***Overhead****: small problem size suffer...*
- ***Hash Collision****: even slower on small dimensions than on moderate dimensions*

Intuition

Apply SJLT *to a moderate dimension!*

Exploiting this intuition, we propose FactGraSS: **Fact**orized version of GraSS:

Exploiting this intuition, we propose FactGraSS: **Fact**orized version of GraSS:



We see that FactGraSS for one $\widetilde{g}_{i,l}$ involves:

We see that FactGraSS for one $\widetilde{g}_{i,l}$ involves:

1. *Sparsification*: Mask both factors of $g_{i,l}$ to $\sqrt{k'/L}$ with $k < k' \ll p$
2. *Reconstruction*: construct the "sparsified gradient" of dimension k'/L
3. *Sparse projection*: SJLT the sparsified gradient of dimension k'/L down to k/L

We see that FactGraSS for one $\widetilde{g}_{i,l}$ involves:

1. *Sparsification*: Mask both factors of $g_{i,l}$ to $\sqrt{k'/L}$ with $k < k' \ll p$
2. *Reconstruction*: construct the "sparsified gradient" of dimension k'/L
3. *Sparse projection*: SJLT the sparsified gradient of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

1. Two Mask from $\sqrt{p/L}$ to $\sqrt{k'/L}$: $O(\sqrt{k'/L})$

We see that FactGraSS for one $\widetilde{g}_{i,l}$ involves:

1. *Sparsification*: Mask both factors of $g_{i,l}$ to $\sqrt{k'/L}$ with $k < k' \ll p$
2. *Reconstruction*: construct the "sparsified gradient" of dimension k'/L
3. *Sparse projection*: SJLT the sparsified gradient of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

1. Two Mask from $\sqrt{p/L}$ to $\sqrt{k'/L}$: $O(\sqrt{k'/L})$
2. Tensor product between two vectors of size $O(\sqrt{k'/L})$: $O(k'/L)$

We see that FactGraSS for one $\widetilde{g}_{i,l}$ involves:

1. *Sparsification*: Mask both factors of $g_{i,l}$ to $\sqrt{k'/L}$ with $k < k' \ll p$
2. *Reconstruction*: construct the "sparsified gradient" of dimension k'/L
3. *Sparse projection*: SJLT the sparsified gradient of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

1. Two Mask from $\sqrt{p/L}$ to $\sqrt{k'/L}$: $O(\sqrt{k'/L})$
2. Tensor product between two vectors of size $O(\sqrt{k'/L})$: $O(k'/L)$
3. SJLT from $O(k'/L)$ to $O(k/L)$: $O(k'/L)$

We see that FactGraSS for one $\widetilde{g}_{i,l}$ involves:

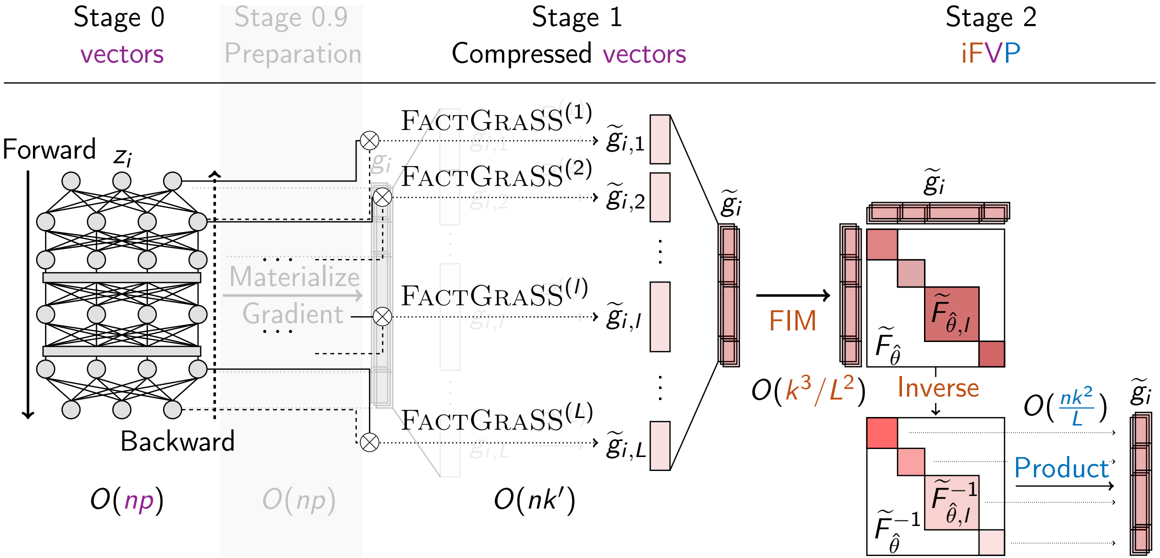
1. *Sparsification*: Mask both factors of $g_{i,l}$ to $\sqrt{k'/L}$ with $k < k' \ll p$
2. *Reconstruction*: construct the "sparsified gradient" of dimension k'/L
3. *Sparse projection*: SJLT the sparsified gradient of dimension k'/L down to k/L

We see that the compression time per $g_{i,l}$ consists of:

1. Two Mask from $\sqrt{p/L}$ to $\sqrt{k'/L}$: $O(\sqrt{k'/L})$
2. Tensor product between two vectors of size $O(\sqrt{k'/L})$: $O(k'/L)$
3. SJLT from $O(k'/L)$ to $O(k/L)$: $O(k'/L)$

Overall, FactGraSS takes $O(nk')$, same as GraSS, *but without materializing* $g_{i,l}$!

Putting Everything Together: FactGraSS



Summary

We summarize the results in the following:

Summary

We summarize the results in the following:

Theorem (GRASS & FACTGRASS [Hu+25])

There is a sublinear compression-based influence function algorithm with an overhead of

$$O(nk'), \text{ where } k < k' \ll p.$$

Summary

We summarize the results in the following:

Theorem (GraSS & FactGraSS [Hu+25])

There is a sublinear compression-based influence function algorithm with an overhead of

$$O(nk'), \text{ where } k < k' \ll p.$$

Moreover, this extends to linear layers, where layer-wise gradients are never materialized.

Summary

We summarize the results in the following:

Theorem (GraSS & FactGraSS [Hu+25])

There is a sublinear compression-based influence function algorithm with an overhead of

$$O(nk'), \text{ where } k < k' \ll p.$$

Moreover, this extends to linear layers, where layer-wise gradients are never materialized.

Remark

Compared to LoGra which takes $O(n\sqrt{kp})$, FactGraSS is faster when

$$nk' < n\sqrt{kp} \Leftrightarrow k' < \sqrt{kp}.$$

Let $k' = ck$, then above is equivalent to $ck \leq \sqrt{kp} \Leftrightarrow c \leq \sqrt{p/k}$.

Table of Content

Experimental Setup

We consider the following setups:

- experiment on TRAK and influence function
- focus on *speed* and *accuracy* of our method

Quantitative Study: Small model and datasets

- Accuracy: Able to measure *LDS scores*
- Efficiency: Compare *wall-time* difference for projection

Qualitative Study: Large model and datasets

- Accuracy: Case study on the most influential data points
- Efficiency: Focus on *throughput*

Table of Content

Quantitative Study

	Sparsification			Sparse Projection			Baselines					
	$\textsc{Mask}_k$			SJLT_k			FJLT_k			$\textsc{Random}_k$		
k	2048	4096	8192	2048	4096	8192	2048	4096	8192	2048	4096	8192
LDS	0.3803	0.4054	0.4318	**0.4171**	**0.4280**	**0.4357**	0.4146	0.4359	0.4347	0.4101	0.4253	0.4346
Time (s)	**0.1517**	**0.1458**	**0.1501**	0.4919	0.5172	0.4754	0.8997	1.4341	2.4387	3.0806	5.5421	10.8355

Table: MLP with MNIST on TRAK.

	Sparsification			Sparse Projection			GraSS			Baseline		
	$\textsc{Mask}_k$			SJLT_k			$\text{SJLT}_k \circ \textsc{Mask}_{4k_{\text{max}}}$			FJLT_k		
k	2048	4096	8192	2048	4096	8192	2048	4096	8192	2048	4096	8192
LDS	0.3690	0.4116	0.4236	0.4131	**0.4499**	0.4747	0.4123	0.4357	0.4545	**0.4157**	0.4497	**0.4753**
Time (s)	**0.1026**	**0.1074**	**0.1296**	12.3590	12.2393	17.4836	0.3652	0.3648	0.3993	31.5491	48.1669	81.9322

Table: ResNet9 with CIFAR2 on TRAK.

Quantitative Study

	Sparsification			Sparse Projection			GraSS			Baseline		
	Mask_k			SJLT_k			$\text{SJLT}_k \circ \text{Mask}_{64k_{\max}}$			FJLT_k		
k	2048	4096	8192	2048	4096	8192	2048	4096	8192	2048	4096	8192
LDS	0.1281	0.1456	0.1469	**0.3062**	0.3533	0.3861	0.2840	0.3242	0.3413	0.2907	**0.3585**	**0.4011**
Time (s)	**0.5341**	**0.5067**	**0.5179**	21.6460	21.1881	21.3192	2.6934	2.6071	2.7202	100.8136	156.0613	269.9093

Table: MusicTransformer with MAESTRO on TRAK.

	Sparsification			Sparse Projection			FactGraSS			Baseline (LoGra)		
	$\text{Mask}_{\sqrt{\hat{k}} \otimes \sqrt{\hat{k}}}$			$\text{SJLT}_{\sqrt{\hat{k}} \otimes \sqrt{\hat{k}}}$			$\text{SJLT}_{\sqrt{\hat{k}}^2} \circ \text{Mask}_{2\sqrt{\hat{k}} \otimes 2\sqrt{\hat{k}}}$			$\text{Random}_{\sqrt{\hat{k}} \otimes \sqrt{\hat{k}}}$		
$\hat{k}$ $(= k/L)$	256	1024	4096	256	1024	4096	256	1024	4096	256	1024	4096
LDS	0.1034	0.1479	**0.2391**	**0.1240**	**0.1897**	0.2389	0.1126	0.1784	0.2360	0.1188	0.1818	0.2338
Time (s)	**5.4933**	**5.3643**	**5.6385**	132.5404	133.4029	136.5163	6.5790	7.4161	6.3075	20.4839	20.9835	22.2157

Table: GPT2-small with WikiText on (block-diagonal FIM) influence function.

Table of Content

Qualitative Study

Next, we compare FactGraSS and LoGra on billion-scale model and dataset

$\hat{k}$ $(= k/L)$	Compress			iHVP		
	256	1024	4096	256	1024	4096
LoGra	27,292	27,255	26,863	7,307	7,478	7,367
FactGraSS	**72,218**	**72,684**	**73,811**	**8,584**	**8,594**	**8,681**

Table: Throughput (tokens/s) for Llama-3.1-8B-Instruct on (block-diagonal FIM) influence function.

Remark

In terms of gradient compression, FactGraSS *outperforms* LoGra *by 160%.*

Qualitative Study

To improve data privacy,

To improve data privacy, the European Union has implemented the General Data Protection Regulation (GDPR). ...
Data Protection Principles
The GDPR sets out six data protection principles...

- **Lawfulness, fairness, and transparency**: Businesses must process personal data in a way that is lawful, fair, and transparent. ...
- **Storage limitation**: Businesses must not store personal data for longer than necessary. ...

Data Subject Rights
The GDPR gives individuals a range of rights when it comes to their personal data. These rights include:

- **Right to access**: Individuals have the right to access their personal data and obtain information about how it is being processed. ...
- **Right to erasure**: Individuals have the right to have their personal data deleted if it is no longer necessary for the purposes for which it was collected. ...

Influential Data

...
The fact of registration and authorization of users on Sputnik websites via users' account or accounts on social networks indicates acceptance of these rules.
Users are obliged abide by national and international laws. ... The administration has the right to delete comments made in languages other than the language of the majority of the websites ...
...

- violates privacy, distributes personal data of third parties without their consent or violates privacy of correspondence; ...
- pursues commercial objectives, contains improper advertising unlawful political advertisement or links to other online resources ...

The administration has the right to block a user's access to the page or delete a user's account without notice if the user is in violation of these rules or if behavior indicating said violation is detected.
If the moderators deem it possible to restore the account/unlock access, it will be done. In the case of repeated violations of the rules above resulting in a second block of a user account, access cannot be restored. ...

Q&A Time!

Thanks! Ask ***anything*** you want!

[ABH17] Naman Agarwal, Brian Bullins, and Elad Hazan. "Second-order stochastic optimization for machine learning in linear time". In: *Journal of Machine Learning Research* 18.116 (2017), pp. 1–40.

[Cho+24] Sang Keun Choe et al. *What Is Your Data Worth to GPT? LLM-Scale Data Valuation with Influence Functions*. May 22, 2024. DOI: 10.48550/arXiv.2405.13954. arXiv: 2405.13954 [cs]. URL: http://arxiv.org/abs/2405.13954 (visited on 09/14/2024).

[DKS10] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. "A sparse johnson: Lindenstrauss transform". In: *Proceedings of the forty-second ACM symposium on Theory of computing*. 2010, pp. 341–350.

[Gro+23] Roger Grosse et al. "Studying large language model generalization with influence functions". In: *arXiv preprint arXiv:2308.03296* (2023).

[He+25] Yifei He et al. "Localize-and-Stitch: Efficient Model Merging via Sparse Task Arithmetic". In: *Transactions on Machine Learning Research* (2025). ISSN: 2835-8856. URL: https://openreview.net/forum?id=9CWU80i86d.

[HNM19] Satoshi Hara, Atsushi Nitanda, and Takanori Maehara. "Data cleansing for models trained with SGD". In: *Advances in Neural Information Processing Systems* 32 (2019).

[Hu+25] Pingbang Hu et al. "GraSS: Scalable Influence Function with Sparse Gradient Compression". In: *Advances in Neural Information Processing Systems*. 2025.

References II

[KL17] Pang Wei Koh and Percy Liang. "Understanding black-box predictions via influence functions". In: *International conference on machine learning*. PMLR. 2017, pp. 1885–1894.

[KN14] Daniel M Kane and Jelani Nelson. "Sparser johnson-lindenstrauss transforms". In: *Journal of the ACM (JACM)* 61.1 (2014), pp. 1–23.

[Kwo+24] Yongchan Kwon et al. "DataInf: Efficiently Estimating Data Influence in LoRA-tuned LLMs and Diffusion Models". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=9m02ib92Wz.

[MG15] James Martens and Roger Grosse. "Optimizing Neural Networks with Kronecker-factored Approximate Curvature". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 2408–2417. URL: https://proceedings.mlr.press/v37/martens15.html.

[Par+23] Sung Min Park et al. "TRAK: Attributing Model Behavior at Scale". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 27074–27113.

[Sch+22] Andrea Schioppa et al. "Scaling Up Influence Functions". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.8 (June 2022), pp. 8179–8186. DOI: 10.1609/aaai.v36i8.20791. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20791.

[Wan+24] Ke Wang et al. "Localizing task information for improved model merging and compression". In: *arXiv preprint arXiv:2405.07813* (2024).

References III

[Wan+25a] Jiachen T Wang et al. "GREATS: Online Selection of High-Quality Data for LLM Training in Every Iteration". In: *Advances in Neural Information Processing Systems* 37 (2025), pp. 131197–131223.

[Wan+25b] Jiachen T. Wang et al. "Capturing the Temporal Dependence of Training Data Influence". In: *The Thirteenth International Conference on Learning Representations*. 2025. URL: https://openreview.net/forum?id=uHLgDEgiS5.

[Woj+16] Mike Wojnowicz et al. ""Influence Sketching": Finding Influential Samples in Large-Scale Regressions". In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016 IEEE International Conference on Big Data (Big Data). Washington DC,USA: IEEE, Dec. 2016, pp. 3601–3612. ISBN: 978-1-4673-9005-7. DOI: 10.1109/BigData.2016.7841024. URL: http://ieeexplore.ieee.org/document/7841024/ (visited on 12/06/2023).

[Yad+23] Prateek Yadav et al. "Ties-merging: Resolving interference when merging models". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 7093–7115.