Robust Learning

Amir H. Payberah
payberah@kth.se
2020-11-09
https://fid3024.github.io
Adversarial Goals

- Confidentiality and privacy
  - Confidentiality of the model itself (e.g., intellectual property)
  - Privacy of the training or test data (e.g., medical records)
- Integrity
  - Integrity of the predictions
- Availability
  - Availability of the system deploying machine learning
Adversarial Goals

- **Confidentiality and privacy**
  - Confidentiality of the *model* itself (e.g., intellectual property)
  - Privacy of the *training* or *test data* (e.g., medical records)
Adversarial Goals

- **Confidentiality and privacy**
  - Confidentiality of the model itself (e.g., intellectual property)
  - Privacy of the training or test data (e.g., medical records)

- **Integrity**
  - Integrity of the predictions
Adversarial Goals

- **Confidentiality and privacy**
  - Confidentiality of the model itself (e.g., intellectual property)
  - Privacy of the training or test data (e.g., medical records)

- **Integrity**
  - Integrity of the predictions

- **Availability**
  - Availability of the system deploying machine learning
Adversarial Capabilities for Integrity Attacks

- **Training** phase

- Adding noise during training

- Data poisoning during training

- Inference phase

- White box

- Black box

[Papernot et al., SoK: Security and Privacy in Machine Learning, 2018]
Adversarial Capabilities for Integrity Attacks

- **Training phase**
  - White box
  - Black box

- **Inference phase**
  - White box
  - Black box

Reference:
[Papernot et al., SoK: Security and Privacy in Machine Learning, 2018]
Our Focus and Goal

- Data parallelization

Our Focus and Goal

▶ Data parallelization
▶ Each worker is prone to adversarial attack.

Our Focus and Goal

- Data parallelization
- Each worker is prone to adversarial attack.
- Adversarial attacks: some unknown subset of computing devices are compromised and behave adversarially (e.g., sending out malicious messages)

Our Focus and Goal

- Data parallelization
- Each worker is prone to adversarial attack.
- Adversarial attacks: some unknown subset of computing devices are compromised and behave adversarially (e.g., sending out malicious messages)
- Our goal: integrity of the model in the training phase

Distributed Stochastic Gradient Descent (1/3)

- One parameter server, and $n$ workers.

Distributed Stochastic Gradient Descent (1/3)

- One **parameter server**, and \( n \) workers.
- Computation is divided into **synchronous rounds**.

Distributed Stochastic Gradient Descent (1/3)

- One **parameter server**, and **n workers**.
- Computation is divided into **synchronous rounds**.
- During round **t**, the **parameter server** broadcasts its parameter vector \( \mathbf{w} \in \mathbb{R}^d \) to all the workers.

At each round $t$, each correct worker $i$ computes $G_i(w_t, \beta)$. 

$G_i(w_t, \beta)$: the local estimate of the gradient of the loss function $\nabla J(w_t)$.

$\beta$: a mini-batch of i.i.d. samples drawn from the dataset.

$G_i(w_t, \beta) = \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l_i(w_t, x)$

At each round $t$, each correct worker $i$ computes $G_i(w_t, \beta)$.

$G_i(w_t, \beta)$: the local estimate of the gradient of the loss function $\nabla J(w_t)$.

At each round $t$, each correct worker $i$ computes $G_i(w_t, \beta)$.

$G_i(w_t, \beta)$: the local estimate of the gradient of the loss function $\nabla J(w_t)$.

$\beta$: a mini-batch of i.i.d. samples drawn from the dataset.
Distributed Stochastic Gradient Descent (2/3)

- At each round $t$, each correct worker $i$ computes $G_i(w_t, \beta)$.
- $G_i(w_t, \beta)$: the local estimate of the gradient of the loss function $\nabla J(w_t)$.
- $\beta$: a mini-batch of i.i.d. samples drawn from the dataset.
- $G_i(w_t, \beta) = \frac{1}{|\beta|} \sum_{x \in \beta} \nabla l_i(w_t, x)$

Distributed Stochastic Gradient Descent (3/3)

- The parameter server computes $F(G_1, G_2, \cdots, G_n)$

The parameter server computes $F(G_1, G_2, \cdots, G_n)$

**Gradient Aggregation Rule (GAR):**

$$F(G_1, G_2, \cdots, G_n) = \frac{1}{n} \sum_{i=1}^{n} G_i$$

Distributed Stochastic Gradient Descent (3/3)

- The parameter server computes $F(G_1, G_2, \cdots, G_n)$
- **Gradient Aggregation Rule (GAR):** $F(G_1, G_2, \cdots, G_n) = \frac{1}{n} \sum_{i=1}^{n} G_i$
- The parameter server updates the parameter vector $w \leftarrow w - \gamma F(G_1, G_2, \cdots, G_n)$

Distributed SGD with Byzantine Workers

Among the $n$ workers, $f$ of them are possibly Byzantine (behaving arbitrarily).

[El-Mhamdi et al., Fast and Secure Distributed Learning in High Dimension, 2019]
Among the $n$ workers, $f$ of them are possibly Byzantine (behaving arbitrarily).

A Byzantine worker $b$ proposes a vector $G_b$ that can deviate arbitrarily from the vector it is supposed.

[El-Mhamdi et al., Fast and Secure Distributed Learning in High Dimension, 2019]
Averaging GAR and Byzantine Workers

- Averaging GAR: \( F(G_1, G_2, \cdots, G_n) = \frac{1}{n} \sum_{i=1}^{n} G_i \)

- \( w \leftarrow w - \gamma F(G_1, G_2, \cdots, G_n) \)

Even a single Byzantine worker can prevent convergence. Proof: if the Byzantine worker proposes \( G_n = U - \sum_{i=1}^{n-1} G_i \), then \( F = U \).
Averaging GAR and Byzantine Workers

- Averaging GAR: \( F(G_1, G_2, \ldots, G_n) = \frac{1}{n} \sum_{i=1}^{n} G_i \)

- \( w \leftarrow w - \gamma F(G_1, G_2, \ldots, G_n) \)

- Even a **single Byzantine** worker can prevent convergence.
Averaging GAR and Byzantine Workers

- Averaging GAR: $F(G_1, G_2, \cdots, G_n) = \frac{1}{n} \sum_{i=1}^{n} G_i$

- $w \leftarrow w - \gamma F(G_1, G_2, \cdots, G_n)$

- Even a single Byzantine worker can prevent convergence.

- Proof: if the Byzantine worker proposes $G_n = nU - \sum_{i=1}^{n-1} G_i$, then $F = U$. 
Assume $n$ workers, where $f$ of them are Byzantine workers.
(\(\alpha, f\))-Byzantine-Resilience (1/2)

- Assume \(n\) workers, where \(f\) of them are Byzantine workers.
- \(\alpha \in [0, \pi/2]\) and \(f \in \{0, \cdots, n\}\).
(\(\alpha, f\))-Byzantine-Resilience (1/2)

- Assume \(n\) workers, where \(f\) of them are Byzantine workers.

- \(\alpha \in [0, \pi/2]\) and \(f \in \{0, \cdots, n\}\).

- \((G_1, \cdots, G_{n-f}) \in (\mathbb{R}^d)^{n-f}\) are i.i.d. random vectors.
\((\alpha, f)-\text{Byzantine-Resilience (1/2)}\)

- Assume \(n\) workers, where \(f\) of them are Byzantine workers.

- \(\alpha \in [0, \pi/2]\) and \(f \in \{0, \cdots, n\}\).

- \((G_1, \cdots, G_{n-f}) \in \mathbb{R}^{d}_{n-f}\) are i.i.d. random vectors
  - \(G_i \sim g\)
  - \(\mathbb{E}[g] = J\), where \(J = \nabla J(w)\)
Assume $n$ workers, where $f$ of them are Byzantine workers.

- $\alpha \in [0, \pi/2]$ and $f \in \{0, \ldots, n\}$.

- $(G_1, \ldots, G_{n-f}) \in (\mathbb{R}^d)^{n-f}$ are i.i.d. random vectors
  - $G_i \sim g$
  - $\mathbb{E}[g] = J$, where $J = \nabla J(w)$

- $(B_1, \ldots, B_f) \in (\mathbb{R}^d)^f$ are random vectors, possibly dependent between them and the vectors $(G_1, \ldots, G_{n-f})$
A GAR $F$ is said to be $(\alpha, f)$-Byzantine-resilient if, for any $1 \leq j_1 < \cdots < j_f \leq n$, the vector $F(G_1, \cdots, B_1, \cdots, B_f, \cdots, G_n)$ satisfies:

1. Vector $F$ that is not too far from the real gradient $J$, i.e., $||E[F] - J|| \leq r$.
2. Moments of $F$ should be controlled by the moments of the (correct) gradient estimator $E[g] = J$.

[Blanchard et al., Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent, 2017]
A GAR $F$ is said to be $(\alpha, f)$-Byzantine-resilient if, for any $1 \leq j_1 < \cdots < j_f \leq n$, the vector $F(G_1, \cdots, B_1, \cdots, B_f, \cdots, G_n)$ satisfies:

1. Vector $F$ that is not too far from the real gradient $\mathcal{J}$, i.e., $\|\mathbb{E}[F] - \mathcal{J}\| \leq r$.

[Blanchard et al., Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent, 2017]
(\(\alpha, f\))-Byzantine-Resilience (2/2)

A GAR \(F\) is said to be \((\alpha, f)\)-Byzantine-resilient if, for any \(1 \leq j_1 < \cdots < j_f \leq n\), the vector \(F(G_1, \cdots, B_1, \cdots, B_f, \cdots, G_n)\) satisfies:

1. Vector \(F\) that is not too far from the real gradient \(\mathcal{J}\), i.e., \(\|\mathbb{E}[F] - \mathcal{J}\| \leq r\).

2. Moments of \(F\) should be controlled by the moments of the (correct) gradient estimator \(g\), where \(\mathbb{E}[g] = \mathcal{J}\).

[Blanchard et al., Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent, 2017]
Byzantine-Resilience GAR

- Median
- Krum
- Multi-Krum
- Brute
Median

- $n \geq 2f + 1$
Median

- $n \geq 2f + 1$

- $\text{median}(x_1, \cdots, x_n) = \arg \min_{x \in \mathbb{R}} \sum_{i=1}^{n} |x_i - x|$
Median

- \( n \geq 2f + 1 \)

- \( \text{median}(x_1, \cdots, x_n) = \arg\min_{x \in \mathbb{R}} \sum_{i=1}^{n} |x_i - x| \)

- \( d \): the gradient vectors dimension.

- Geometric median
  \[
  F = \text{GeoMed}(G_1, \cdots, G_n) = \arg\min_{G \in \mathbb{R}^d} \sum_{i=1}^{n} ||G_i - G||
  \]
Median

- \( n \geq 2f + 1 \)
- \( \text{median}(x_1, \cdots, x_n) = \arg \min_{x \in \mathbb{R}} \sum_{i=1}^{n} |x_i - x| \)
- \( d \): the gradient vectors dimension.

- **Geometric median**
  \[ F = \text{GeoMed}(G_1, \cdots, G_n) = \arg \min_{G \in \mathbb{R}^d} \sum_{i=1}^{n} ||G_i - G|| \]

- **Marginal median**
  \[ F = \text{MarMed}(G_1, \cdots, G_n) = \begin{pmatrix} \text{median}(G_1[1], \cdots, G_n[1]) \\ \vdots \\ \text{median}(G_1[d], \cdots, G_n[d]) \end{pmatrix} \] (1)
- \( n \geq 2f + 3 \)
▶ $n \geq 2f + 3$

▶ Idea: to preclude the vectors that are too far away.
- $n \geq 2f + 3$

- Idea: to preclude the vectors that are too far away.

- $s(i) = \sum_{i \rightarrow j} ||G_i - G_j||^2$, the score of the worker $i$. 
Krum

- $n \geq 2f + 3$
- Idea: to preclude the vectors that are too far away.
- $s(i) = \sum_{i \rightarrow j} \|G_i - G_j\|^2$, the score of the worker $i$.
- $i \rightarrow j$ denotes that $G_j$ belongs to the $n - f - 2$ closest vectors to $G_i$. 
n ≥ 2f + 3

Idea: to preclude the vectors that are too far away.

s(i) = \sum_{i \to j} ||G_i - G_j||^2, the score of the worker i.

i \to j denotes that G_j belongs to the n – f – 2 closest vectors to G_i.

F(G_1, \cdots, G_n) = G_{i^*}
- $n \geq 2f + 3$
- Idea: to preclude the vectors that are too far away.
- $s(i) = \sum_{i \rightarrow j} ||G_i - G_j||^2$, the score of the worker $i$.
- $i \rightarrow j$ denotes that $G_j$ belongs to the $n - f - 2$ closest vectors to $G_i$.
- $F(G_1, \ldots, G_n) = G_{i^*}$
- $G_{i^*}$ refers to the worker minimizing the score, $s(i^*) \leq s(i)$ for all $i$. 
Multi-Krum

- Multi-Krum computes the score for each vector proposed (as in Krum).
Multi-Krum

- Multi-Krum computes the score for each vector proposed (as in Krum).
- It selects $m$ vectors $G_{1*}, \ldots, G_{m*}$, which score the best ($1 \leq m \leq n - f - 2$).
Multi-Krum

- Multi-Krum computes the score for each vector proposed (as in Krum).
- It selects $m$ vectors $G_{1*}, \cdots, G_{m*}$, which score the best ($1 \leq m \leq n - f - 2$).
- It outputs their average $\frac{1}{m} \sum_{i} G_{i*}$.
Multi-Krum

- Multi-Krum computes the score for each vector proposed (as in Krum).
- It selects \( m \) vectors \( G_{1^*}, \ldots, G_{m^*} \), which score the best \( (1 \leq m \leq n - f - 2) \).
- It outputs their average \( \frac{1}{m} \sum_i G_{i^*} \).
- The cases \( m = 1 \) and \( m = n \) correspond to Krum and averaging, respectively.

[Blanchard et al., Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent, 2017]
Brute

- $n \geq 2f + 1$
• The set of all the subsets of \( n - f \):

\[ S = \arg \min_{X \in R} \left( \max_{(G_i, G_j) \in X} 2 \left( ||G_i - G_j|| \right) \right) \]

• Selects the \( n - f \) most clumped gradients among the submitted ones.

\[ F(G_1, \ldots, G_n) = \frac{1}{n-f} \sum_{G \in S} G \]
Brute

- $n \geq 2f + 1$

- $Q = \{G_1, G_2, \cdots, G_n\}$

- $R = \{X | X \subset Q, |X| = n - f\}$
  - The set of all the subsets of $n - f$
\[ n \geq 2f + 1 \]

\[ Q = \{ G_1, G_2, \cdots, G_n \} \]

\[ R = \{ \mathcal{X} | \mathcal{X} \subset Q, |\mathcal{X}| = n - f \} \]

- The set of all the subsets of \( n - f \)

\[ S = \arg \min_{\mathcal{X} \in R} \left( \max_{(G_i, G_j) \in \mathcal{X}} (||G_i - G_j||) \right) \]

- Selects the \( n - f \) most clumped gradients among the submitted ones.
Brute

- \( n \geq 2f + 1 \)

- \( Q = \{ G_1, G_2, \cdots, G_n \} \)

- \( R = \{ X | X \subset Q, |X| = n - f \} \)
  - The set of all the subsets of \( n - f \)

- \( S = \arg \min_{X \in R} (\max_{(G_i, G_j) \in X^2} (||G_i - G_j||)) \)
  - Selects the \( n - f \) most clumped gradients among the submitted ones.

- \( F(G_1, \cdots, G_n) = \frac{1}{n-f} \sum_{G \in S} G \)
[El Mhamdi et al., The Hidden Vulnerability of Distributed Learning in Byzantium, 2018]
Limitation of previous aggregation methods.

If gradient dimension \( d \gg 1 \), then the distance function between two vectors \( \| X - Y \|_p \), cannot distinguish these two cases:
Weak Byzantine Resilience

- **Limitation** of previous aggregation methods.

- If gradient dimension $d \gg 1$, then the distance function between two vectors $\|X - Y\|_p$, cannot distinguish these two cases:

  1. Does $X$ and $Y$ disagree a bit on each coordinate?
Weak Byzantine Resilience

- **Limitation** of previous aggregation methods.

- If gradient dimension $d \gg 1$, then the distance function between two vectors $\|X - Y\|_p$, cannot distinguish these two cases:
  
  - 1. Does $X$ and $Y$ disagree a bit on each coordinate?
  
  - 2. Does $X$ and $Y$ disagree a lot on only one?
Strong Byzantine Resilience

- Ensuring convergence (as in weak Byzantine resilience functions).
Strong Byzantine Resilience

- Ensuring convergence (as in weak Byzantine resilience functions).

- Ensures that each coordinate is agreed on by a majority of vectors that were selected by a Byzantine resilient aggregation rule $A$. 

Ensuring convergence (as in weak Byzantine resilience functions).

Ensures that each coordinate is agreed on by a majority of vectors that were selected by a Byzantine resilient aggregation rule A.

A can be Brute, Krum, Median, etc.
Strong Byzantine Resilience

- Ensuring convergence (as in weak Byzantine resilience functions).
- Ensures that each coordinate is agreed on by a majority of vectors that were selected by a Byzantine resilient aggregation rule $A$.
- $A$ can be Brute, Krum, Median, etc.
- Bulyan is a strong Byzantine-resilience algorithm.
The Hidden Vulnerability of Distributed Learning in Byzantium
Bulyan - Step One (1/2)

- \( n \geq 4f + 3 \)
- A two step process.
A two step process.

The first one is to recursively use $A$ to select $\theta = n - 2f$ gradients:

- $n \geq 4f + 3$
A two step process.

The first one is to recursively use $A$ to select $\theta = n - 2f$ gradients:

1. With $A$, choose, among the proposed vectors, the closest one to $A$’s output (for Krum this would be the exact output of $A$).
- $n \geq 4f + 3$

- A two step process.

- The first one is to recursively use $A$ to select $\theta = n - 2f$ gradients:
  
  1. With $A$, choose, among the proposed vectors, the closest one to $A$’s output (for Krum this would be the exact output of $A$).
  2. Remove the chosen gradient from the received set and add it to the selection set $S$. 
n ≥ 4f + 3

A two step process.

The first one is to recursively use $A$ to select $\theta = n - 2f$ gradients:

1. With $A$, choose, among the proposed vectors, the closest one to $A$’s output (for Krum this would be the exact output of $A$).
2. Remove the chosen gradient from the received set and add it to the selection set $S$.
3. Loop back to step 1 if $|S| < \theta$. 
\[ \theta = n - 2f \geq 2f + 3, \text{ thus } S = (S_1, \cdots, S_\theta) \text{ contains a majority of non-Byzantine gradients.} \]
\[ \theta = n - 2f \geq 2f + 3, \text{ thus } S = (S_1, \ldots, S_\theta) \text{ contains a majority of non-Byzantine gradients.} \]

- For each \(i \in [1..d]\), the median of the \(\theta\) coordinates \(i\) of the selected gradients is always bounded by coordinates from non-Byzantine submissions.
The second step is to generate the resulting gradient $F = (F[1], \cdots, F[d])$. 
The second step is to generate the resulting gradient $F = (F[1], \cdots, F[d])$.

$\forall i \in [1..d], F[i] = \frac{1}{\beta} \sum_{X \in M[i]} X[i]$
Bulyan - Step Two

- The second step is to generate the resulting gradient $F = (F[1], \ldots, F[d])$.
- $\forall i \in [1..d], F[i] = \frac{1}{\beta} \sum_{X \in M[i]} X[i]$
- $\beta = \theta - 2f \geq 3$
The second step is to generate the resulting gradient $F = (F[1], \cdots, F[d])$.

- $\forall i \in [1..d], F[i] = \frac{1}{\beta} \sum_{X \in M[i]} X[i]$ 
- $\beta = \theta - 2f \geq 3$ 
- $M[i] = \arg \min_{R \subseteq S, |R| = \beta} (\sum_{X \in R} |X[i] - \text{median}[i]|)$
The second step is to generate the resulting gradient \( F = (F[1], \ldots, F[d]) \).

\[ \forall i \in [1..d], F[i] = \frac{1}{\beta} \sum_{X \in M[i]} X[i] \]

\[ \beta = \theta - 2f \geq 3 \]

\[ M[i] = \arg \min_{R \subseteq S, |R| = \beta} (\sum_{X \in R} |X[i] - \text{median}[i]|) \]

\[ \text{median}[i] = \arg \min_{m = Y[i], Y \in S} (\sum_{Z \in S} |Z[i] - m|) \]
The second step is to generate the resulting gradient $F = (F[1], \ldots, F[d])$.

- $\forall i \in [1..d], F[i] = \frac{1}{\beta} \sum_{X \in M[i]} X[i]$
- $\beta = \theta - 2f \geq 3$
- $M[i] = \arg \min_{R \subseteq S, |R| = \beta} (\sum_{X \in R} |X[i] - \text{median}[i]|)$
- $\text{median}[i] = \arg \min_{m = Y[i], Y \in S} (\sum_{Z \in S} |Z[i] - m|)$

Each $i$th coordinate of $F$ is equal to the average of the $\beta$ closest $i$th coordinates to the median $i$th coordinate of the $\theta$ selected gradients.
[El Mhamdi et al., The Hidden Vulnerability of Distributed Learning in Byzantium, 2018]
AggregaThor: Byzantine Machine Learning via Robust Gradient Aggregation
AggregaThor (1/2)

- A framework that handles the distribution of the training of a TensorFlow neural network graph over a cluster of machines.
- This distribution is robust to Byzantine cluster nodes.

[Damaskinos et al., AggregaThor: Byzantine Machine Learning via Robust Gradient Aggregation, 2019]
Relies on Multi-Krum and Bulyan.
AggregaThor (2/2)

- Relies on **Multi-Krum** and **Bulyan**.

- **Multi-Krum** selects $m$ gradients that deviate less from the majority
  - Based on their relative distances.
- Relies on **Multi-Krum** and **Bulyan**.

- **Multi-Krum** selects $m$ gradients that deviate less from the majority
  - Based on their relative distances.

- **Bulyan** takes the aforementioned $m$ vectors.
AggregaThor (2/2)

- Relies on Multi-Krum and Bulyan.
  - Multi-Krum selects $m$ gradients that deviate less from the majority
    - Based on their relative distances.
  - Bulyan takes the aforementioned $m$ vectors.
    - Computes their coordinate-wise median.
Relies on Multi-Krum and Bulyan.

- **Multi-Krum** selects $m$ gradients that deviate less from the majority
  - Based on their relative distances.

- **Bulyan** takes the aforementioned $m$ vectors.
  - Computes their coordinate-wise median.
  - Produces a gradient that coordinates are the average of the $m - 2f$ closest values to the median.
In TensorFlow, Byzantine resilience cannot be achieved solely through the use of a Byzantine-resilient GAR.
TensorFlow Limitation

- In TensorFlow, Byzantine resilience cannot be achieved solely through the use of a Byzantine-resilient GAR.

- TensorFlow allows any node in the cluster to execute arbitrary operations anywhere in the cluster.
In TensorFlow, Byzantine resilience cannot be achieved solely through the use of a Byzantine-resilient GAR.

TensorFlow allows any node in the cluster to execute arbitrary operations anywhere in the cluster.

A single Byzantine worker could continually overwrite the shared parameters with arbitrary values.
In TensorFlow, Byzantine resilience cannot be achieved solely through the use of a Byzantine-resilient GAR.

TensorFlow allows any node in the cluster to execute arbitrary operations anywhere in the cluster.

A single Byzantine worker could continually overwrite the shared parameters with arbitrary values.

AggregaThor patches TensorFlow to overcome the above issues.
\[ \text{mini-batch size} = 250 \]

\[ \text{mini-batch size} = 20 \]

\[ n = 19, f = 4 \quad (n \geq 4f + 3) \]

[Damaskinos et al., AggregaThor: Byzantine Machine Learning via Robust Gradient Aggregation, 2019]
What if parameter servers are Byzantine?
SGD: Decentralized Byzantine Resilience
[El Hhamdi et al., SGD: Decentralized Byzantine Resilience, 2019]
GuanYu

- Byzantine tolerant learning algorithm that is
GuanYu

- Byzantine tolerant learning algorithm that is
  1. Resilience to Byzantine workers.
GuanYu

Byzantine tolerant learning algorithm that is
1. Resilience to Byzantine workers.
2. Resilience to Byzantine parameter servers.
Byzantine tolerant learning algorithm that is
1. Resilience to Byzantine workers.
2. Resilience to Byzantine parameter servers.

GuanYu tolerates up to $\frac{1}{3}$ Byzantine servers and $\frac{1}{3}$ Byzantine workers.
GuanYu

- Byzantine tolerant learning algorithm that is
  1. Resilience to Byzantine workers.
  2. Resilience to Byzantine parameter servers.

- GuanYu tolerates up to $\frac{1}{3}$ Byzantine servers and $\frac{1}{3}$ Byzantine workers.

- GuanYu uses a GAR for aggregating workers’ gradients and Median for aggregating models received from servers.
Asynchronous network: the lack of any bound on communication delays.
Assumptions and Notations (1/2)

- **Asynchronous network**: the lack of any bound on communication delays.
- **Synchronous training**: bulk-synchronous training.
Assumptions and Notations (1/2)

- **Asynchronous network**: the lack of any bound on communication delays.

- **Synchronous training**: bulk-synchronous training.
  - The parameter server does not need to wait for all the workers’ gradients to make progress, and vice versa.
Assumptions and Notations (1/2)

- **Asynchronous network**: the lack of any bound on communication delays.

- **Synchronous training**: bulk-synchronous training.
  - The parameter server does not need to wait for all the workers’ gradients to make progress, and vice versa.
  - The quorums indicate the number of messages to wait before aggregating them.
Assumptions and Notations (2/2)

- $n_{ps} \geq 3f_{ps} + 3$ the total number of parameter servers, among which $f_{ps}$ are Byzantine.
Assumptions and Notations (2/2)

- \( n_{ps} \geq 3f_{ps} + 3 \) the total number of parameter servers, among which \( f_{ps} \) are Byzantine.
- \( n_{wr} \geq 3f_{wr} + 3 \) the total number of workers, among which \( f_{wr} \) are Byzantine.
Assumptions and Notations (2/2)

▶ $n_{ps} \geq 3f_{ps} + 3$ the total number of parameter servers, among which $f_{ps}$ are Byzantine.
▶ $n_{wr} \geq 3f_{wr} + 3$ the total number of workers, among which $f_{wr}$ are Byzantine.
▶ $M$ the coordinate-wise median (used in both workers and servers).
Assumptions and Notations (2/2)

- $n_{ps} \geq 3f_{ps} + 3$ the total number of parameter servers, among which $f_{ps}$ are Byzantine.
- $n_{wr} \geq 3f_{wr} + 3$ the total number of workers, among which $f_{wr}$ are Byzantine.
- $M$ the coordinate-wise median (used in both workers and servers).
- $F$ the GAR function (used in the servers)
Assumptions and Notations (2/2)

- \( n_{ps} \geq 3f_{ps} + 3 \) the total number of parameter servers, among which \( f_{ps} \) are Byzantine.
- \( n_{wr} \geq 3f_{wr} + 3 \) the total number of workers, among which \( f_{wr} \) are Byzantine.
- \( M \) the coordinate-wise median (used in both workers and servers).
- \( F \) the GAR function (used in the servers)
- \( 2f_{ps} + 3 \leq q_{ps} \leq n_{ps} - f_{ps} \) the quorum used for \( M \).
Assumptions and Notations (2/2)

- $n_{ps} \geq 3f_{ps} + 3$ the total number of parameter servers, among which $f_{ps}$ are Byzantine.
- $n_{wr} \geq 3f_{wr} + 3$ the total number of workers, among which $f_{wr}$ are Byzantine.
- $M$ the coordinate-wise median (used in both workers and servers).
- $F$ the GAR function (used in the servers)
- $2f_{ps} + 3 \leq q_{ps} \leq n_{ps} - f_{ps}$ the quorum used for $M$.
- $2f_{wr} + 3 \leq q_{wr} \leq n_{wr} - f_{wr}$ the quorum used for $F$. 
Assumptions and Notations (2/2)

- $n_{ps} \geq 3f_{ps} + 3$ the total number of parameter servers, among which $f_{ps}$ are Byzantine.
- $n_{wr} \geq 3f_{wr} + 3$ the total number of workers, among which $f_{wr}$ are Byzantine.
- $M$ the coordinate-wise median (used in both workers and servers).
- $F$ the GAR function (used in the servers)
- $2f_{ps} + 3 \leq q_{ps} \leq n_{ps} - f_{ps}$ the quorum used for $M$.
- $2f_{wr} + 3 \leq q_{wr} \leq n_{wr} - f_{wr}$ the quorum used for $F$.
- $d$ the dimension of the parameter space $\mathbb{R}^d$. 
At each step $t$, each non-Byzantine server $i$ broadcasts its current parameter vector $w^t_i$ to every worker.
GuanYu Algorithm - Step 1

- At each step $t$, each non-Byzantine server $i$ broadcasts its current parameter vector $w_i^t$ to every worker.

- Each non-Byzantine worker $j$ aggregates with $M$ the $q_{ps}$ first received $w^t$. 
GuanYu Algorithm - Step 1

- At each step $t$, each non-Byzantine server $i$ broadcasts its current parameter vector $w^t_i$ to every worker.

- Each non-Byzantine worker $j$ aggregates with $M$ the $q_{ps}$ first received $w^t$.

- And computes an estimate $G^t_j$ of the gradient at the aggregated parameters.
GuanYu Algorithm - Step 2

- Each non-Byzantine worker $j$ broadcasts its computed gradient estimation $G^t_j$ to every parameter server.
GuanYu Algorithm - Step 2

- Each non-Byzantine worker $j$ broadcasts its computed gradient estimation $G^t_j$ to every parameter server.

- Each non-Byzantine parameter server $i$ aggregates with $F$ the $q_{wr}$ first received $G^t$. 
Each non-Byzantine worker $j$ broadcasts its computed gradient estimation $G^t_j$ to every parameter server.

Each non-Byzantine parameter server $i$ aggregates with $F$ the $q_{wr}$ first received $G^t$.

And performs a local parameter update with the aggregated gradient, resulting in $\overline{w}^t_i$. 
GuanYu Algorithm - Step 3

- Each non-Byzantine parameter server $i$ broadcasts $\overline{w}_i^{t+1}$ to every other parameter servers.
GuanYu Algorithm - Step 3

- Each non-Byzantine parameter server $i$ broadcasts $\overline{w}_i^{t+1}$ to every other parameter servers.

- They aggregate with $M$ the $q_{ps}$ first received $\overline{w}_k^{t+1}$. 
GuanYu Algorithm - Step 3

- Each non-Byzantine parameter server $i$ broadcasts $\overline{w}_i^{t+1}$ to every other parameter servers.

- They aggregate with $M$ the $q_{ps}$ first received $\overline{w}_k^{t+1}$.

- This aggregated parameter vector is $\overline{w}_i^{t+1}$. 
[El Mhamdi et al., SGD: Decentralized Byzantine Resilience, 2019]
GuanYu Limitations

- **Network asynchrony assumption is costly:**
  - It requires *three communication rounds*, instead of two in the vanilla case.
GuanYu Limitations

- **Network asynchrony assumption is costly:**
  - It requires three communication rounds, instead of two in the vanilla case.

- It requires a large number of compute nodes and server replicas to work, as one cannot differentiate between a Byzantine machine and a slow one in such network.
Fast Machine Learning with Byzantine Workers and Servers
LiuBei

- Uses a GAR to aggregate workers’ gradients.
LiuBei

- Uses a GAR to aggregate workers’ gradients.

- Tolerating Byzantine servers using a filtering technique and the scatter/gather protocol (both assume network synchrony).
- Uses a GAR to aggregate workers’ gradients.

- Tolerating Byzantine servers using a filtering technique and the scatter/gather protocol (both assume network synchrony).
  - Scatter phase: servers work independently and do not communicate with each other.
Uses a GAR to aggregate workers' gradients.

Tolerating Byzantine servers using a filtering technique and the scatter/gather protocol (both assume network synchrony).

- Scatter phase: servers work independently and do not communicate with each other.
- Gather phase: correct servers communicate to bring their view of models back close to each other.
- Uses a GAR to aggregate workers’ gradients.

- Tolerating Byzantine servers using a filtering technique and the scatter/gather protocol (both assume network synchrony).
  - Scatter phase: servers work independently and do not communicate with each other.
  - Gather phase: correct servers communicate to bring their view of models back close to each other.
  - The number of gather steps is usually very small and hence, their overhead is insignificant.
Assumptions and Notations

- **Network synchrony**: an upper bound on communication machines.

- $n_{ps} \geq 3f_{ps} + 1$ the total number of parameter servers, among which $f_{ps}$ are Byzantine.

- $n_{wr} \geq 2f_{wr} + 1$ the total number of workers, among which $f_{wr}$ are Byzantine.
LiuBei - Byzantine Workers

- LiuBei can use any existing synchronous GAR that follows the robustness definition $(\alpha, f)$-Byzantine-Resilience.
Tolerating Byzantine servers using robust aggregation requires communication with all servers in each round: big communication overhead.
Tolerating Byzantine servers using robust aggregation requires communication with all servers in each round: big communication overhead.

LiuBei lets each worker pull only one model from any of the server replicas and then checks if the pulled model is suspicious or not.
A worker does this check by applying two filters on the pulled model: the Lipschitz filter and the models filter.
A worker does this check by applying two filters on the pulled model: the Lipschitz filter and the models filter.

If the model is suspicious, the worker discards it and pulls a new model from another parameter server.
A worker does this check by applying two filters on the pulled model: the Lipschitz filter and the models filter.

If the model is suspicious, the worker discards it and pulls a new model from another parameter server.

The maximum number of models that can be pulled by a worker in one iteration is $f_{ps} + 1$. 
Assume at time $t$ worker $j$ owns a model $w^t_j$ and computes gradient $G^t_j$ based on that model.
Assume at time $t$ worker $j$ owns a model $w^t_j$ and computes gradient $G^t_j$ based on that model.

A correct server $i$ should include $G^t_j$ while updating its model $w^t_i$, given network synchrony.
Assume at time $t$, worker $j$ owns a model $\mathbf{w}_j^t$ and computes gradient $G_j^t$ based on that model.

A correct server $i$ should include $G_j^t$ while updating its model $\mathbf{w}_i^t$, given network synchrony.

The worker $j$ then does two steps in parallel: $\mathbf{w}_i^{t+1}$
Lipschitz Filter

- Assume at time $t$ worker $j$ owns a model $w^t_j$ and computes gradient $G^t_j$ based on that model.

- A correct server $i$ should include $G^t_j$ while updating its model $w^t_i$, given network synchrony.

- The worker $j$ then does two steps in parallel: $w^{t+1}_i$
  1. Estimates the updated model locally based on its own gradient: $w^{t+1}_{j(1)}$
Lipschitz Filter

- Assume at time $t$ worker $j$ owns a model $w_{jt}$ and computes gradient $G_{jt}$ based on that model.

- A correct server $i$ should include $G_{jt}$ while updating its model $w_{it}$, given network synchrony.

- The worker $j$ then does two steps in parallel: $w_{it+1}$
  1. Estimates the updated model locally based on its own gradient: $w_{jt+1}(t)$
  2. Pulls a model $w_{it+1}$ from a parameter server $i$. 


Assume at time $t$ worker $j$ owns a model $w^t_j$ and computes gradient $G^t_j$ based on that model.

A correct server $i$ should include $G^t_j$ while updating its model $w^t_i$, given network synchrony.

The worker $j$ then does two steps in parallel: $w^{t+1}_i$

1. Estimates the updated model locally based on its own gradient: $w^{t+1}_{j(1)}$
2. Pulls a model $w^{t+1}_i$ from a parameter server $i$.

If server $i$ is correct, then, the growth of the pulled model $w^{t+1}_i$ should be close to that of the estimated local model $w^{t+1}_{j(1)}$. 
LiuBei uses the model filter to bound the distance between models in any two successive iterations.
LiuBei uses the **model filter** to **bound the distance** between **models** in any two successive iterations.

- We assume all **correct machines initialize models with the same state**.
LiuBei uses the **model filter** to **bound the distance** between **models** in any two successive iterations.

We assume all **correct machines initialize models** with the same state.

Building upon the **guarantees given by the used GAR**, at iteration $t$, a worker can **estimate an upper bound** on the **distance between two successive states** of a correct model.
LiuBei Algorithm (1/2)

- LiuBei operates in two phases: scatter and gather.

---

Algorithm 1 Worker Algorithm

1: Calculate the value of $T$ and a value for $seed$
2: $model \leftarrow \text{init}\_\text{model}(seed)$
3: $r \leftarrow \text{random}\_\text{int}(1,n_{ps})$
4: $t \leftarrow 0$
5: $\text{grad} \leftarrow \text{model}.\text{backprop}()$
6: repeat
7:   $\text{local}\_\text{model} \leftarrow \text{apply}\_\text{grad}(\text{model,grad})$
8:   if $t \mod T = 0$ then
9:     $\text{models} \leftarrow \text{read}\_\text{models}()$
10:    $\text{model} \leftarrow \text{MeaMed}(\text{models})$
11: else
12:   $i \leftarrow 0$
13: repeat
14:     $\text{new}\_\text{model} \leftarrow \text{read}\_\text{model}(\(r + t + i \mod n_{ps}\))$
15:     $\text{new}\_\text{grad} \leftarrow \text{new}\_\text{model}.\text{backprop}()$
16:     $i \leftarrow i + 1$
17: until $\text{pass}\_\text{filters}(\text{new}\_\text{model})$
18: $\text{model} \leftarrow \text{new}\_\text{model}$
19: $\text{grad} \leftarrow \text{new}\_\text{grad}$
20: end if
21: $t \leftarrow t + 1$
22: until $t > \text{num}\_\text{iterations}$
LiuBei Algorithm (1/2)

- LiuBei operates in two phases: scatter and gather.
- One gather step is executed every \( T \) iterations (line 8 to 11).

```
Algorithm 1 Worker Algorithm
1: Calculate the value of \( T \) and a value for \( seed \)
2: model ← init_model(seed)
3: \( r ← \) random_int(1, \( n_{ps} \))
4: \( t ← 0 \)
5: grad ← model.backprop()
6: repeat
7:   local_model ← apply_grad(model, grad)
8:   if \( t \mod T = 0 \) then
9:     models ← read_models()
10:    model ← MeaMed(models)
11:  else
12:    \( i ← 0 \)
13:   repeat
14:     new_model ← read_model((\( r + t + i \)) \mod \( n_{ps} \))
15:     new_grad ← new_model.backprop()
16:     \( i ← i + 1 \)
17:   until pass_filters(new_model)
18: model ← new_model
19: grad ← new_grad
20: end if
21: \( t ← t + 1 \)
22: until \( t > \) num_iterations
```
LiuBei Algorithm (1/2)

- LiuBei operates in two phases: scatter and gather.
- One gather step is executed every $T$ iterations (line 8 to 11).
- We call the whole $T$ iterations a scatter step.

\begin{algorithm}
\caption{Worker Algorithm}
\begin{algorithmic}[1]
\STATE Calculate the value of $T$ and a value for seed
\STATE $\text{model} \leftarrow \text{init\_model}(\text{seed})$
\STATE $r \leftarrow \text{random\_int}(1,n_{\text{ps}})$
\STATE $t \leftarrow 0$
\STATE $\text{grad} \leftarrow \text{model}.\text{backprop}()$
\REPEAT
\STATE $\text{local\_model} \leftarrow \text{apply\_grad(model,grad)}$
\STATE \textbf{if} $t \mod T = 0$ \textbf{then}
\STATE \quad $\text{models} \leftarrow \text{read\_models}()$
\STATE \quad $\text{model} \leftarrow \text{MeaMed(models)}$
\STATE \textbf{else}
\STATE \quad $i \leftarrow 0$
\REPEAT
\STATE \quad $\text{new\_model} \leftarrow \text{read\_model}(r + t + i) \mod n_{\text{ps}}$
\STATE \quad $\text{new\_grad} \leftarrow \text{new\_model}.\text{backprop}()$
\STATE \quad $i \leftarrow i + 1$
\UNTIL \text{pass\_filters(new\_model)}
\STATE $\text{model} \leftarrow \text{new\_model}$
\STATE $\text{grad} \leftarrow \text{new\_grad}$
\STATE \textbf{end if}
\STATE $t \leftarrow t + 1$
\UNTIL $t > \text{num\_iterations}$
\end{algorithmic}
\end{algorithm}
LiuBei Algorithm (2/2)

- LiuBei operates in two phases: **scatter** and **gather**.
- **One gather step** is executed every $T$ iterations (line 8 to 11).
- We call the whole $T$ iterations a **scatter step**.

### Algorithm 2 Parameter Server Algorithm

1: Calculate the value of $T$ and a value for seed
2: model ← init_model(seed)
3: $t ← 0$
4: **repeat**
5:   grads ← read_gradients()
6:   grad ← MDA(grads)
7:   model.update(grad)
8:   **if** $t \mod T = 0$ **then**
9:     model ← read_models()
10:    model ← MeaMed(models)
11: **end if**
12: $t ← t + 1$
13: **until** $t > \text{num\_iterations}$
El Mhamdi et al., Fast Machine Learning with Byzantine Workers and Servers, 2019

(a) mini-batch size = 250

(b) mini-batch size = 100

[El Mhamdi et al., Fast Machine Learning with Byzantine Workers and Servers, 2019]
Summary
Summary

- Integrity in data-parallel learning
- Weak Byzantine resilience
- Strong Byzantine resilience
- Byzantine parameter servers
Reference

- Xie et al., Generalized Byzantine-tolerant SGD, 2018
- Blanchard et al., Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent, 2017
- El Mhamdi et al., The Hidden Vulnerability of Distributed Learning in Byzantium, 2018
- Damaskinos et al., AGGREGATHOR: Byzantine Machine Learning via Robust Gradient Aggregation, 2019
- El Mhamdi et al., SGD: Decentralized Byzantine Resilience, 2019
- El Mhamdi et al., Fast Machine Learning with Byzantine Workers and Servers, 2019
Questions?