Compression and Acceleration of Pre-trained Language Models

Lu Hou

HKUST, Oct 28, 2020

Content

1. **Pre-trained Language Models**
   - NEZHA
2. **Inference on the edge**
   - Distillation: TinyBERT
   - Pruning/Dynamic network: DynaBERT
   - Quantization: TernaryBERT
3. **Training on the edge**
   - distributed low-bit training

Language Model

- **Language Model:** a probability distribution over sequences of words (sentences) in a given language L
- Given a length-m sequence, it assigns a probability $P(w_1, w_2 \cdots, w_m)$ to it

Any task which could be transferred to a sequential problems is suitable for LMs

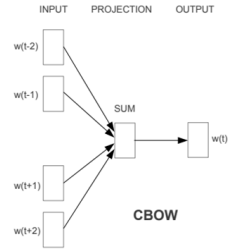
- Text generation/Image captioning/Text summarization
- Machine translation
- Speech Recognition
- Reading Comprehension
- POS tagging / Entity recognition / Parsing

$\boldsymbol{x}$ "input"	$\boldsymbol{w}$ "**text** output"
An author	A document written by that author
A topic label	An article about that topic
{SPAM, NOT_SPAM}	An email
A sentence in French	Its English translation
A sentence in English	Its French translation
A sentence in English	Its Chinese translation
An image	A text description of the image
A document	Its summary
A document	Its translation
Meterological measurements	A weather report
Acoustic signal	Transcription of speech
Conversational history + database	Dialogue system response
A question + a document	Its answer
A question + an image	Its answer

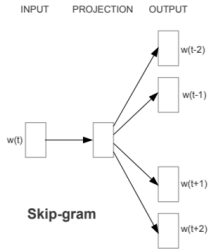
Representation of Natural Language: First Generation

word embedding: numerical representation of words

- Continuous Bag of Words (CBOW)/Skip-gram/GloVe
  - word embedding does **not** change as the **context** changes
  - **distance** between words in the embedding space is related to **semantic similarity**



CBOW: predicts the current word based on the context



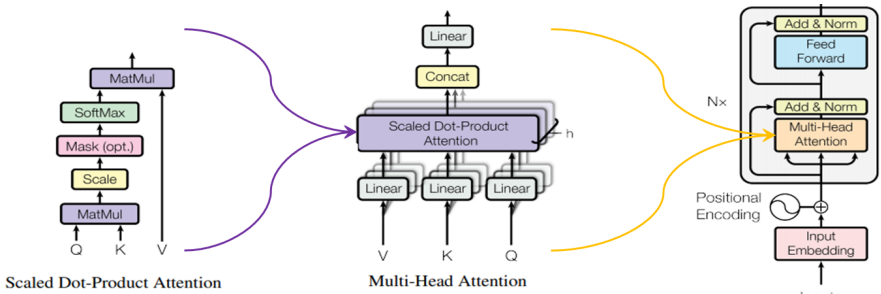
Skip-gram: predicts surrounding words given the current word

[1] Mikolov et al., Efficient Estimation of Word Representations in Vector Space, 2013.

[2] Pennington et al., Glove: Global vectors for word representation, EMNLP 2014.

Representation of Natural Language: Second Generation

- **Transformer-based** Pre-trained Language Models
  - embedding jointly learned with the model
  - **contextual** word embedding



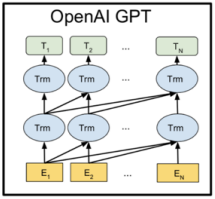
$$\text{Attn}^{h}_{\mathbf{W}^Q_h,\mathbf{W}^K_h,\mathbf{W}^V_h}(\mathbf{X}) = \text{Softmax}(1/\sqrt{d}\mathbf{X}\mathbf{W}^Q_h\mathbf{W}^{K\top}_h\mathbf{X}^\top)\mathbf{X}\mathbf{W}^V_h$$

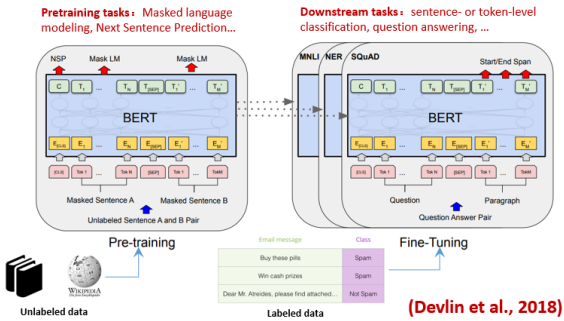
$$\text{MHAttn}_{\mathbf{W}^Q,\mathbf{W}^K,\mathbf{W}^V,\mathbf{W}^O}(\mathbf{X}) = \text{Concat}(\text{Attn}^1,\cdots,\text{Attn}^{N_H})\mathbf{W}^O$$

[1] Vaswani et al., Attention is all you need, NIPS 2017.

Transformer-based Pre-trained Language Models

- **Pre-training** on large unlabeled corpus using MLM, NSP
- **Fine-tuning** on downstream tasks





[1]Devlin et al., Bert: Pre-training of deep bidirectional transformers for language understanding, NAACL 2018.

[2]Radford et al., Improving Language Understanding by Generative Pre-Training, 2019.

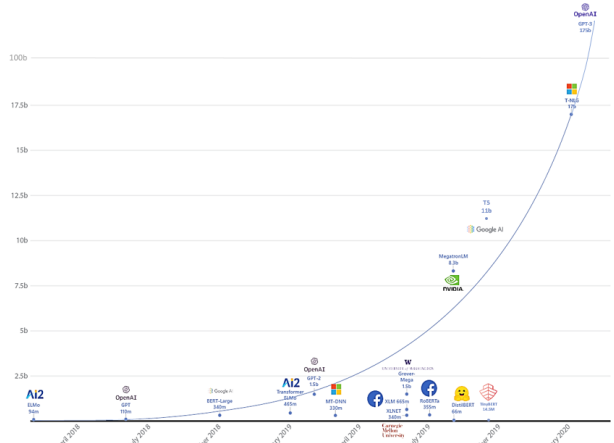
NEZHA

- **Model architecture:** similar size as BERT
- **Techniques:**
  - Relative Positional Embedding
  - Whole Word Masking: the original prediction task was too 'easy' for words that had been split into multiple WordPieces
  - Mixed Precision Training
  - Lamb optimizer
  - PMLM (Probabilistically Masked Language Model)[2]
- **Data:** Wiki, Chinese news, Chinese wiki
- **Resources:** Ascend 910, NVIDIA GPU
- **Framework:** Tensorflow, Pytorch, Mindspore

[1]Wei et al., NEZHA: Neural contextualized representation for chinese language understanding, 2019.

[2]Liao et al., Probabilistically Masked Language Model Capable of Autoregressive Generation in Arbitrary Word Order, ACL 2020.

Pre-trained language models are becoming larger



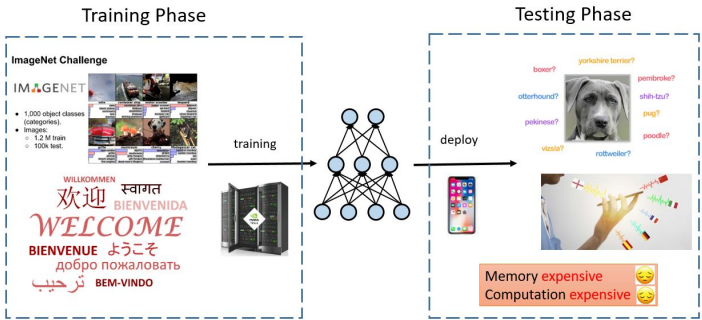
Model	#Parameters
BERT-Base	110M
BERT-Large	340M
RoBERTa	355M
GPT-2	1.5B
Megatron-LM	8.3B
T5	11B
Turing-NLG	17B
GPT-3	175B
...	

3

[1]Figure from https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

Inference on the edge

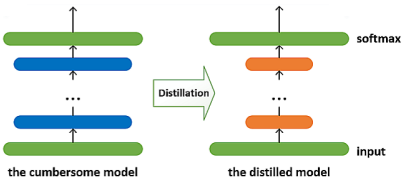
- Problems with conventional cloud computing
  - privacy concern
  - latency constraint
  - availability, reliability and cost of data transmission



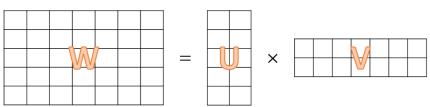
→ model compression and acceleration

BERT Compression I

1. **Distillation**
   - **various knowledge:** logits, hidden representations, attention patterns ...
   - DistilBERT, BERT-PKD, TinyBERT, MiniLM, MobileBERT



2. **Low-rank Approximation:**
   - **On the embedding:** ALBERT
   - **On the weights:** Tensorized transformer (Ma et al., 2019)



3. **Weight Sharing**
   - Universal Transformer, ALBERT, Deep Equilibrium Model (Bai et al., 2019)

BERT Compression II

4. **Quantization**
   - **Quantization-aware training:** Q-BERT, Q8BERT, Quant-noise, TernaryBERT
   - **Post-training quantization:** GOBO
5. **Pruning**
   - **Unstructured:** Connections: Gordon et al., 2020; Attention: Cui et al., 2019
   - **Structured:**
     - heads: Michel et al., 2019, Voita et al., 2019
     - layers: LayerDrop
     - others: McCarley. et al., 2019
6. **More Compact Modules and AutoML**
   - **convolution:** Light/DynamicConv(Wu et al., 2018), AdaBERT
   - **convolution + attention:** Transformer Lite, SqueezeBERT
   - others: BERT-OF-THESEUS

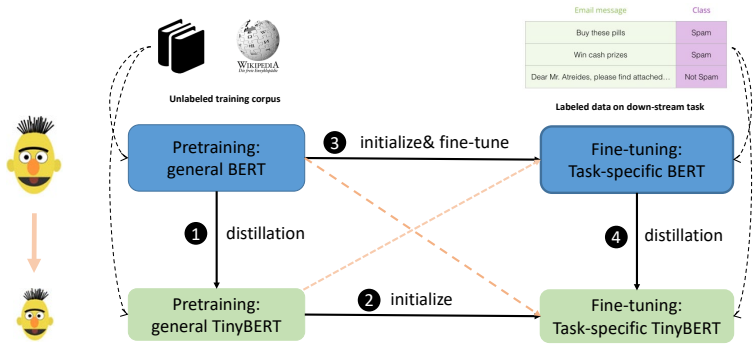
TinyBERT: Distilling BERT for Natural Language Understanding

(Jiao et al., 2020)

[1] Jiao et al, Tinybert: Distilling bert for natural language understanding, EMNLP Findings 2019.
[2] https://github.com/huawei-noah/Pretrained-Language-Model/tree/master/TinyBERT

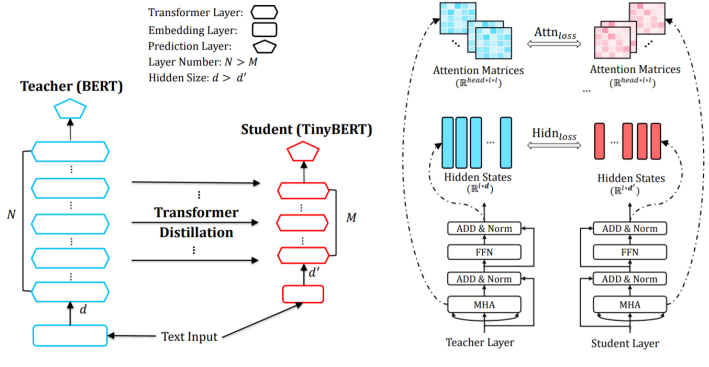
Knowledge Distilation

How to distill knowledge from **a pre-trained BERT** to **a small task-specific BERT**?



- general distillation (steps 1+2)
- task-specific distillation (steps 3+4)

What to distill?



- prediction layer distillation: logits

$$\mathcal{L}_{pred} = \mathrm{SCE}(\mathbf{P}^S, \mathbf{P}^T).$$

- Transformer layer distillation: attention scores and hidden states

$$\mathcal{L}_{trm} = \sum_{m=0}^{M} \mathrm{MSE}(\mathbf{H}_m^S, \mathbf{H}_{g(m)}^T) + \sum_{m=0}^{M} \mathrm{MSE}(\mathbf{A}_m^S, \mathbf{A}_g(m)^T).$$

Empirical Results on the GLUE Benchmark

System	#Params	#FLOPs	Speedup	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Avg
$\text{BERT}_{\text{BASE}}$ (Teacher)	109M	22.5B	1.0x	83.9/83.4	71.1	90.9	93.4	52.8	85.2	87.5	67.0	79.5
$\text{BERT}_{\text{TINY}}$	14.5M	1.2B	9.4x	75.4/74.9	66.5	84.8	87.6	19.5	77.1	83.2	62.6	70.2
$\text{BERT}_{\text{SMALL}}$	29.2M	3.4B	5.7x	77.6/77.0	68.1	86.4	89.7	27.8	77.0	83.4	61.8	72.1
BERT_4-PKD	52.2M	7.6B	3.0x	79.9/79.3	70.2	85.1	89.4	24.8	79.8	82.6	62.3	72.6
DistilBERT_4	52.2M	7.6B	3.0x	78.9/78.0	68.5	85.2	91.4	32.8	76.1	82.4	54.1	71.9
$\text{MobileBERT}_{\text{TINY}}$†	15.1M	3.1B	-	81.5/81.6	68.9	**89.5**	91.7	**46.7**	80.1	**87.9**	65.1	**77.0**
TinyBERT_4 (ours)	14.5M	1.2B	9.4x	**82.5/81.8**	**71.3**	87.7	**92.6**	44.1	**80.4**	86.4	**66.6**	**77.0**
BERT_6-PKD	67.0M	11.3B	2.0x	81.5/81.0	70.7	89.0	92.0	-	-	85.0	65.5	-
PD	67.0M	11.3B	2.0x	82.8/82.2	70.4	88.9	91.8	-	-	86.8	65.3	-
DistilBERT_6	67.0M	11.3B	2.0x	82.6/81.3	70.1	88.9	92.5	49.0	81.3	86.9	58.4	76.8
TinyBERT_6 (ours)	67.0M	11.3B	2.0x	**84.6/83.2**	**71.6**	**90.4**	**93.1**	**51.1**	**83.7**	**87.3**	**70.0**	**79.4**

- Compared to BERT base, 4-layer TinyBERT is 7.5x smaller and 9.4x faster at inference with only 2.5% accuracy drop
- 6-layer TinyBERT has comparable performance as BERT base
- widely used in various Huawei products like voice assistant

DynaBERT: Dynamic BERT with Adaptive Width and Depth

(Hou et al., 2020)

[1]Hou et al., DynaBERT: Dynamic BERT with Adaptive Width and Depth, NeurIPS 2020.
[2]https://github.com/huawei-noah/Pretrained-Language-Model/tree/master/DynaBERT

Motivation

- Transformer-based Pre-trained language models like BERT and RoBERTa have achieved remarkable results on various NLP tasks
- compute and memory inefficient due to large number of parameters

Difficulty in the deployment

1. hardware performances of different devices vary a lot
   $\rightarrow$ infeasible to deploy one single BERT model to different edge devices
2. resource condition of one device under different circumstances can be quite different
   $\rightarrow$ dynamically select a part of the deployed model for inference

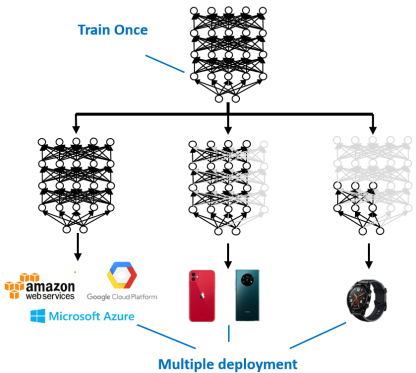
Proposed Model

problem with previous BERT compression methods

- compress a model to a fixed size; or
- only depth-adaptive by varying #Transformer layers → limited configurations
  - e.g., LayerDrop, FastBERT, DeeBERT

DynaBERT

- flexibility in both width and depth
- better exploitation between accuracy and model size
- once trained, no further fine-tuning is required for each sub-network.

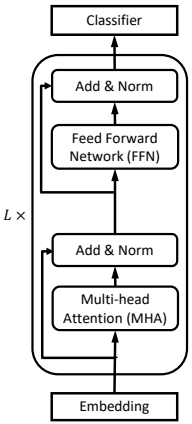


Summary

1. **teacher assistant:**
   - an only width-adaptive DynaBERT$_{\text{W}}$
2. **knowledge distillation:**
   - **BERT** $\rightarrow$ **DynaBERT$_{\text{W}}$**:
     - teacher: fined-tuned BERT
     - student: sub-networks of DynaBERT
   - **DynaBERT$_{\text{W}}$** $\rightarrow$ **DynaBERT**:
     - teacher: sub-networks of DynaBERT$_{\text{W}}$
     - student: sub-networks of DynaBERT
3. **network rewiring:**
   - the more important heads and neurons are utilized by more sub-networks
4. **empirical results:** DynaBERT or DynaRoBERTa
   - at its largest size performs comparably as BERT$_{\text{BASE}}$/ RoBERTa$_{\text{BASE}}$
   - at smaller sizes outperforms other BERT compression methods

Determine the width

width direction:what can be computed in parallel?



original MHA: N_H attention heads are concatenated

$$\text{Head}_{\mathbf{W}_h^Q,\mathbf{W}_h^K,\mathbf{W}_h^V}^h(\mathbf{X}) = \text{Softmax}(\frac{1}{\sqrt{d}}\mathbf{X}\mathbf{W}_h^Q\mathbf{W}_h^{K\top}\mathbf{X}^\top)\mathbf{X}\mathbf{W}_h^V$$

$$\text{MHAttn}_{\mathbf{W}^Q,\mathbf{W}^K,\mathbf{W}^V,\mathbf{W}^O}(\mathbf{X}) = \text{Concat}(\text{Head}^1,\text{Head}^2,\cdots,\text{Head}^{N_H})\mathbf{W}^O$$

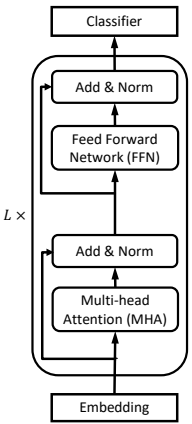
rewrite MHA: N_H attention heads in MHA are computed **in parallel**

$$\text{Attn}_{\mathbf{W}_h^Q,\mathbf{W}_h^K,\mathbf{W}_h^V,\mathbf{W}_h^O}^h(\mathbf{X}) = \text{Softmax}(\frac{1}{\sqrt{d}}\mathbf{X}\mathbf{W}_h^Q\mathbf{W}_h^{K\top}\mathbf{X}^\top)\mathbf{X}\mathbf{W}_h^V\mathbf{W}_h^{O\top}$$

$$\text{MHAttn}_{\mathbf{W}^Q,\mathbf{W}^K,\mathbf{W}^V,\mathbf{W}^O}(\mathbf{X}) = \sum_{h=1}^{N_H}\text{Attn}_{\mathbf{W}_h^Q,\mathbf{W}_h^K,\mathbf{W}_h^V,\mathbf{W}_h^O}^h(\mathbf{X})$$

Determine the width

width direction:what can be computed in parallel?



original FFN: two linear layers

$$\mathrm{FFN}_{\mathbf{W}^1,\mathbf{W}^2,\mathbf{b}^1,\mathbf{b}^2}(\mathbf{A}) = \mathrm{GeLU}(\mathbf{A}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^2 + \mathbf{b}^2$$

rewrite FFN: d_{ff} neurons in the intermediate layer of FFN are computed **in parallel**

$$\mathrm{FFN}_{\mathbf{W}^1,\mathbf{W}^2,\mathbf{b}^1,\mathbf{b}^2}(\mathbf{A}) = \sum_{i=1}^{d_{ff}} \mathrm{GeLU}(\mathbf{A}\mathbf{W}^1_{:,i} + b^1_i)\mathbf{W}^2_{i,:} + \mathbf{b}^2$$

Train with adaptive width: network rewiring

Adapt width by varying #attention heads in MHA and #neurons in FFN

- For width multiplier m_w
  - retain the leftmost $\lfloor m_w N_H \rfloor$ attention heads in MHA
  - retain the leftmost leftmost$\lfloor m_w d_{ff} \rfloor$ neurons in the intermediate layer of FFN.

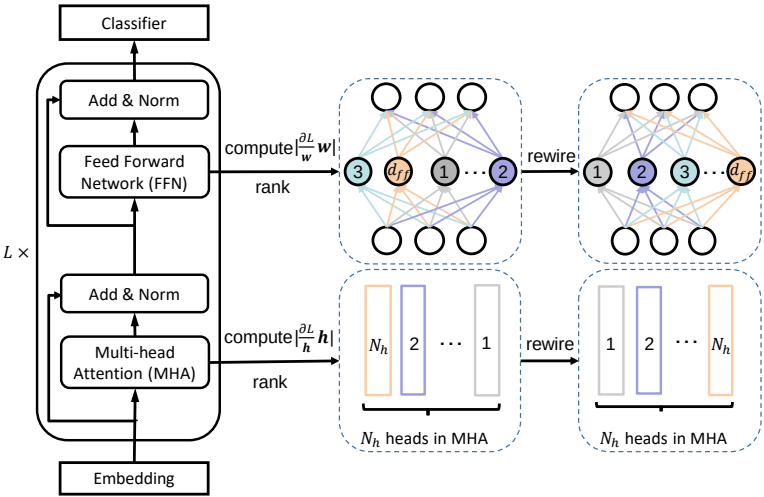
More important heads/neurons utilized by more sub-networks

- **rewire** the connections in each Transformer layer according to importance

Head importance: $$I_{\mathbf{h}} = |\mathcal{L}_{\mathbf{h}} - \mathcal{L}_{\mathbf{h}=\mathbf{0}}| = \left| \mathcal{L}_{\mathbf{h}} - (\mathcal{L}_{\mathbf{h}} - \frac{\partial \mathcal{L}}{\partial \mathbf{h}}(\mathbf{h} - \mathbf{0}) + R_{\mathbf{h}=\mathbf{0}}) \right| \approx \left| \frac{\partial \mathcal{L}}{\partial \mathbf{h}} \mathbf{h} \right|$$

Neuron importance: $$\left| \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \mathbf{w} \right| = \left| \sum_{i=1}^{2d} \frac{\partial \mathcal{L}}{\partial w_i} w_i \right|$$

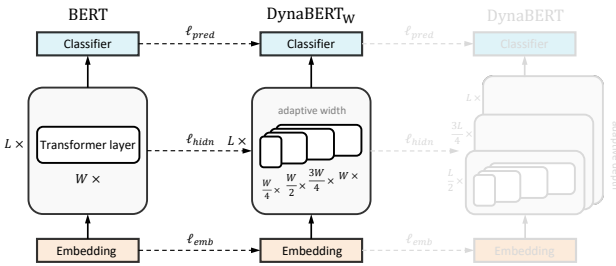
Train with adaptive width: network rewiring



Train with adaptive width: distillation

- distillation over logits: $\ell_{pred}(\mathbf{y}^{(m_w)}, \mathbf{y}) = \mathrm{SCE}(\mathbf{y}^{(m_w)}, \mathbf{y})$
- distillation over word embedding: $\ell_{emb}(\mathbf{E}^{(m_w)}, \mathbf{E}) = \mathrm{MSE}(\mathbf{E}^{(m_w)}, \mathbf{E})$
- distillation over hidden states: $\ell_{hidn}(\mathbf{H}^{(m_w)}, \mathbf{H}) = \sum_{l=1}^{L} \mathrm{MSE}(\mathbf{H}_l^{(m_w)}, \mathbf{H}_l)$.
- **BERT $\rightarrow$ DynaBERT$_\mathbf{W}$:**

$$\mathcal{L} = \lambda_1 \ell_{pred}(\mathbf{y}^{(m_w)}, \mathbf{y}) + \lambda_2(\ell_{emb}(\mathbf{E}^{(m_w)}, \mathbf{E}) + \ell_{hidn}(\mathbf{H}^{(m_w)}, \mathbf{H}))$$



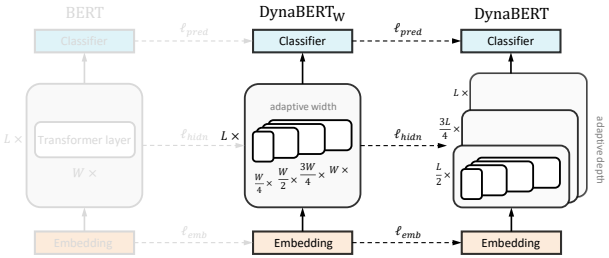
Algorithm 1 Train DynaBERT$_\mathrm{W}$ or DynaBERT.

1: **if** training DynaBERT$_\mathrm{W}$ **then**
2: $\mathcal{L}\leftarrow$(3), $InitM\leftarrow$rewired net, $depthList$=[1].
3: **else**:
4: $\mathcal{L}\leftarrow$(4), $InitM\leftarrow$DynaBERT$_\mathrm{W}$.
5: **initialize** a fixed teacher model and a trainable student model with $InitM$.
6: **for** $iter = 1, \cdots, T_{train}$ **do**
7: Get next mini-batch of training data.
8: Clear gradients in the student model.
9: **for** m_d in $depthList$ **do**
10: **for** m_w in $widthList$ **do**
11: Compute loss $\mathcal{L}$.
12: Accumulate gradient $\mathcal{L}.backward()$.
13: **end for**
14: **end for**
15: Update with the accumulated gradients.
16: **end for**

Train with adaptive width and depth: distillation

- an only width-adaptive DynaBERT$_\text{W}$ to act as a teacher assistant
- **DynaBERT$_\text{W}$ $\rightarrow$ DynaBERT:**

$$\mathcal{L} = \lambda_1 \ell'_{pred}(\mathbf{y}^{(m_w,m_d)}, \mathbf{y}^{(m_w)}) + \lambda_2(\ell'_{emb}(\mathbf{E}^{(m_w,m_d)}, \mathbf{E}^{(m_w)}) + \ell'_{hidn}(\mathbf{H}^{(m_w,m_d)}, \mathbf{H}^{(m_w)}))$$



Algorithm 1 Train DynaBERT$_\text{W}$ or DynaBERT.

1: **if** training DynaBERT$_\text{W}$ **then**
2: $\mathcal{L}\leftarrow$(3), $InitM\leftarrow$rewired net, $depthList$=[1].
3: **else**:
4: $\mathcal{L}\leftarrow$(4), $InitM\leftarrow$DynaBERT$_\text{W}$.
5: **initialize** a fixed teacher model and a trainable student model with $InitM$.
6: **for** $iter = 1, \cdots, T_{train}$ **do**
7: Get next mini-batch of training data.
8: Clear gradients in the student model.
9: **for** m_d in $depthList$ **do**
10: **for** m_w in $widthList$ **do**
11: Compute loss $\mathcal{L}$.
12: Accumulate gradient $\mathcal{L}.backward()$.
13: **end for**
14: **end for**
15: Update with the accumulated gradients.
16: **end for**

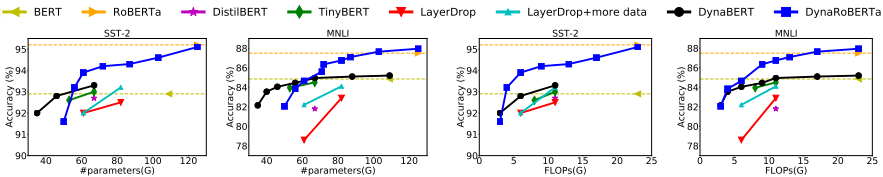
Empirical results

- comparable as $\text{BERT}_{\text{BASE}}$ (or $\text{RoBERTa}_{\text{BASE}}$) with usually smaller size
- full sized model does not necessarily have the best performance
- the width direction is more compressible than depth

Method		CoLA			STS-B			MRPC			RTE		
$\text{BERT}_{\text{BASE}}$		58.1			89.8			87.7			71.1		
	width / depth	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
DynaBERT	1.0x	59.7	59.1	54.6	**90.1**	89.5	88.6	86.3	85.8	85.0	72.2	71.8	66.1
	0.75x	**60.8**	59.6	53.2	90.0	89.4	88.5	**86.5**	85.5	84.1	71.8	**73.3**	65.7
	0.5x	58.4	56.8	48.5	89.8	89.2	88.2	84.8	84.1	83.1	72.2	72.2	67.9
	0.25x	50.9	51.6	43.7	89.2	88.3	87.0	83.8	83.8	81.4	68.6	68.6	63.2
		MNLI-(m/mm)			QQP			QNLI			SST-2		
$\text{BERT}_{\text{BASE}}$		84.8/84.9			90.9			92.0			92.9		
	width / depth	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
DynaBERT	1.0x	**84.9/85.5**	84.4/85.1	83.7/84.6	**91.4**	**91.4**	91.1	92.1	91.7	90.6	93.2	93.3	92.7
	0.75x	84.7/85.5	84.3/85.2	83.6/84.4	**91.4**	91.3	91.2	92.2	91.8	90.7	93.0	93.1	92.8
	0.5x	84.7/85.2	84.2/84.7	83.0/83.6	91.3	91.2	91.0	**92.2**	91.5	90.0	**93.3**	92.7	91.6
	0.25x	83.9/84.2	83.4/83.7	82.0/82.3	90.7	91.1	90.4	91.5	90.8	88.5	92.8	92.0	92.0
		CoLA			STS-B			MRPC			RTE		
$\text{RoBERTa}_{\text{BASE}}$		65.1			91.2			90.7			81.2		
	width / depth	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
DynaRoBERTa	1.0x	63.6	61.0.7	59.5	**91.3**	91.0	90.0	88.7	89.7	88.5	**82.3**	78.7	72.9
	0.75x	**63.7**	61.4	54.9	91.0	90.7	89.7	90.0	89.2	88.2	79.4	77.3	70.8
	0.5x	61.3	58.1	52.9	90.3	90.1	88.9	**90.4**	90.0	86.5	75.1	73.6	71.5
	0.25x	54.2	46.7	39.8	89.6	89.2	87.5	88.2	88.0	84.3	70.0	70.0	66.8
		MNLI-(m/mm)			QQP			QNLI			SST-2		
$\text{RoBERTa}_{\text{BASE}}$		87.5/87.5			91.8			93.1			95.2		
	width / depth	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
DynaRoBERTa	1.0x	**88.3/87.6**	87.7/87.2	86.2/85.8	92.0	**92.0**	91.7	**92.9**	92.5	91.4	**95.1**	94.3	93.3
	0.75x	88.0/87.3	87.5/86.7	85.8/85.4	91.9	91.8	91.6	92.8	92.4	91.3	94.6	94.3	93.3
	0.5x	87.1/86.4	86.8/85.9	84.8/84.2	91.7	91.5	91.2	92.3	91.9	90.8	93.6	94.2	92.9
	0.25x	84.6/84.7	84.0/83.7	82.1/82.0	91.2	91.0	90.5	90.9	90.9	89.3	93.9	93.2	91.6

Empirical results

- **under the same efficiency constraint**: better performance
- **under the same accuracy**: fewer parameters and has faster inference



#parameters(G). FLOPs(G).

Nvidia K40 GPU latency(s). Kirin 810 ARM CPU latency(ms).

Ablation Study

- Training DynaBERT$_W$ with Adaptive Width

	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	avg.
Separate network	82.2	82.2	90.3	87.8	91.0	39.9	84.6	78.8	61.6	77.6
Vanilla DynaBERT$_W$	82.2	82.5	90.6	89.1	91.2	44.0	87.4	80.5	64.2	79.0
+ Network rewiring	83.1	83.0	90.9	90.4	91.7	51.4	89.1	83.8	**69.7**	81.4
+ Distillation and DA	**84.5**	**84.9**	**91.0**	**92.1**	**92.7**	**55.9**	**89.7**	**86.1**	69.5	82.9

- Training DynaBERT with Adaptive Width and Depth

	SST-2	CoLA	MRPC
Vanilla DynaBERT	91.3	46.0	82.1
+ Distillation and DA	92.5	52.8	**84.5**
+ Fine-tuning	**92.7**	**54.8**	83.2

Ablation Study

- DynaBERT$_W$ as a "teacher assistant"

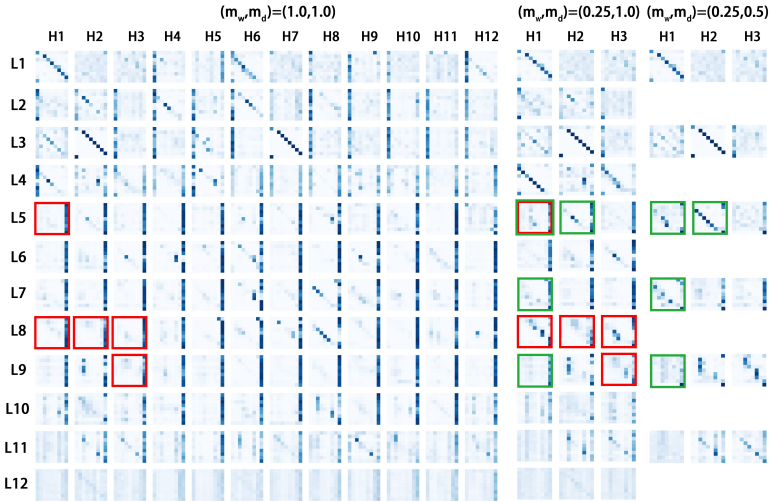
	SST-2	CoLA	MRPC
DynaBERT	**92.7**	**54.8**	**84.5**
- DynaBERT$_W$	92.3	54.1	84.4

- Adaptive Depth First or Adaptive Width First?

	QNLI			SST-2			CoLA			STS-B			MRPC		
m_w or m_d	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x	1.0x	0.75x	0.5x
DynaBERT$_W$	**92.5**	**92.4**	**92.3**	**92.9**	**93.1**	**93.0**	**59.0**	57.9	56.7	**90.0**	**90.0**	**89.9**	86.0	**87.0**	**87.3**
DynaBERT$_D$	92.4	91.9	90.6	**92.9**	92.8	92.1	58.3	**58.3**	52.2	89.9	89.0	88.3	**87.3**	85.8	84.6

Visualization of Attention Maps

Attention maps of sub-networks with different widths and depths in DynaBERT



TernaryBERT: Distillation-aware Ultra-low Bit BERT
(Zhang et al., 2020)

[1]Zhang et al., TernaryBERT: Distillation-aware Ultra-low Bit BERT, EMNLP 2020.

Weight Quantization

Use low bit to represent each weight value

binarization

binarize to -1 or $+1$.

ternarization

convert each weight to $-1, 0$ or $+1$.

m-bit quantization

- linear quantization: quantize each weight to $\mathcal{Q} = \left\{-1, -\frac{k-1}{k}, \ldots, -\frac{1}{k}, 0, \frac{1}{k}, \ldots, \frac{k-1}{k}, 1\right\}$
- logarithmic quantization: quantize each weight to $\mathcal{Q} = \left\{-1, -\frac{1}{2}, \ldots, -\frac{1}{2^{k-1}}, 0, \frac{1}{2^{k-1}}, \ldots, \frac{1}{2}, 1\right\}$

Smaller storage

- Original model: S
- Binary model: $\frac{S}{32}$
- Ternary model: $\frac{S}{16}$
- m-bit quantized model: $\frac{mS}{32}$

Faster Inference

- $\mathbf{x}, \mathbf{y}$ full-precision: $\mathbf{x} \cdot \mathbf{y}$ floating-point multiplication
- $\mathbf{x}$ int8, $\mathbf{y}$ int8: $\mathbf{x} \cdot \mathbf{y}$ integer multiplication

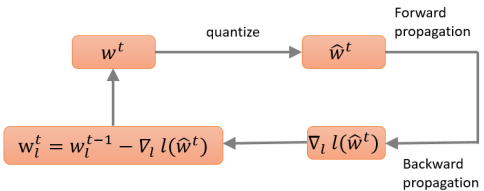
Weight Quantization

Post-training quantization

- train $\rightarrow$ quantize
- use pre-trained full-precision model

Quantization-aware training

- train a quantized network from scratch (or trained full-precision network)
- train and compress simultaneously



tth iteration of quantization-aware training

1. quantization: quantize full-precision weight $\mathbf{w}^t$ to $\hat{\mathbf{w}}^t$
2. forward propagation
3. backpropagate the gradients $\nabla_l \ell(\hat{\mathbf{w}}^t)$
   - straight-through estimator is sometimes used to transform $\nabla_l \ell(\hat{\mathbf{w}}^t)$ to $\nabla_l \ell(\mathbf{w}^t)$
4. update (full-precision) weight as $\mathbf{w}_l^t = \mathbf{w}_l^{t-1} - \eta^t \nabla_l \ell(\hat{\mathbf{w}}^t)$

Ternarization

- **Tenary Weight Network (TWN)**[1]:
  - **Objective:** minimize the distance between the full-precision weight $\mathbf{w}^t$ and ternarized weight $\hat{\mathbf{w}}^t = \alpha^t \mathbf{b}^t$

$$\min_{\alpha_l^t, \mathbf{b}_l^t} \|\mathbf{w}_l^t - \alpha_l^t \mathbf{b}_l^t\|_2^2, \qquad \text{s.t.} \quad \alpha_l^t > 0, \mathbf{b}_l^t \in \{-1, 0, 1\}^{n_l}$$

  - **Solution:** Approximate solution with threshold $\Delta_l^t = 0.7 \cdot E(|\mathbf{w}_l^t|)$

- **Loss-aware Weight Ternarization (LAT)**[2]:
  - **Objective:** directly searchs for the ternary weights that minimize the training loss

$$\min_{\alpha, \mathbf{b}} \mathcal{L}(\alpha \mathbf{b}), \qquad \text{s.t.} \quad \alpha > 0, \mathbf{b} \in \{-1, 0, 1\}^n$$

  - **Solution:** Proximal Newton algorithm, subproblem at each iteration

$$\min_{\alpha^t, \mathbf{b}^t} \|\mathbf{w}^t - \alpha^t \mathbf{b}^t\|^2_{\text{Diag}(\sqrt{\mathbf{v}^t})}, \qquad \text{s.t.} \quad \alpha^t > 0, \mathbf{b}^t \in \{-1, 0, 1\}^n$$

[1] Li et al., Ternary weight networks, 2016.

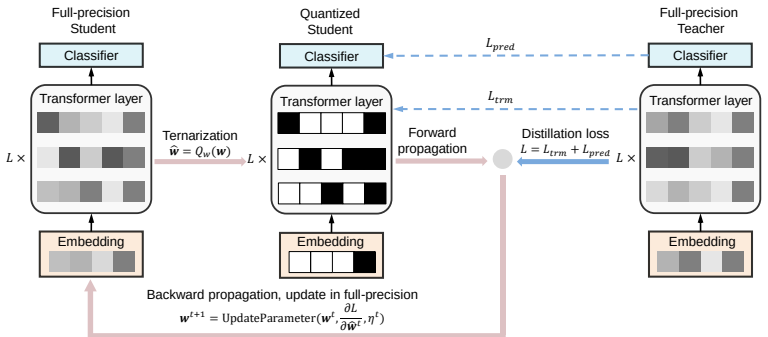
[2] Hou et al., Loss-aware weight quantization of deep networks, ICLR 2018.

Whole framework

- weight: weights in Transformer layers and word embedding (2-bit or 8-bit)
- activation: all input to all matrix multiplication
- min-max 8-bit quantization

$$s = (x_{max} - x_{min})/255, \quad Q(x) = \text{round}((x - x_{min})/s) \times s + x_{min}$$

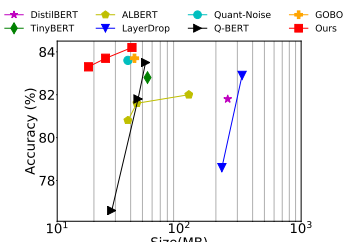
- straight-through estimator to back propagate through quantized neurons



Inference Compression

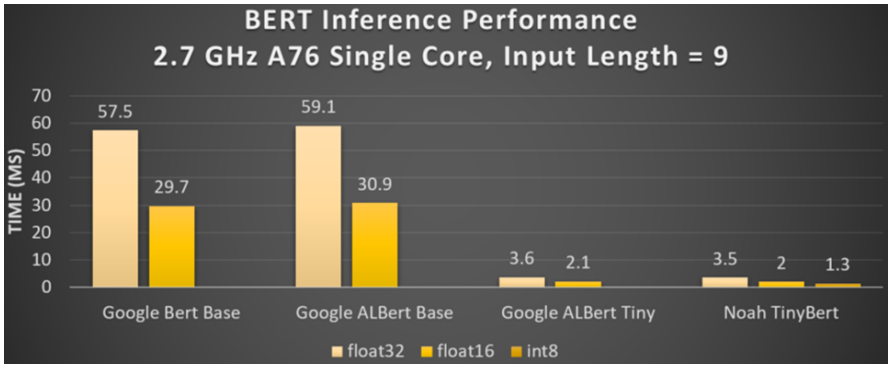
		W-E-A (#bits)	Size (MB)	MNLI-m/mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE
	BERT	32-32-32	418 (×1)	84.5/84.9	87.5/90.9	92.0	93.1	58.1	89.8/89.4	90.6/86.5	71.1
	TinyBERT	32-32-32	258 (×1.6)	84.5/84.5	88.0/91.1	91.1	93.0	54.1	89.8/89.6	91.0/87.3	71.8
2-bit	Q-BERT	2-8-8	43 (×9.7)	76.6/77.0	-	-	84.6	-	-	-	-
	Q2BERT	2-8-8	43 (×9.7)	47.2/47.3	67.0/75.9	61.3	80.6	0	4.4/4.7	81.2/68.4	52.7
	TernaryBERT_TWN (ours)	2-2-8	28 (×14.9)	83.3/83.3	86.7/90.1	91.1	92.8	55.7	87.9/87.7	91.2/87.5	72.9
	TernaryBERT_LAT (ours)	2-2-8	28 (×14.9)	83.5/83.4	86.6/90.1	91.5	92.5	54.3	87.9/87.6	91.1/87.0	72.2
	TernaryTinyBERT_TWN (ours)	2-2-8	18 (×23.2)	83.4/83.8	87.2/90.5	89.9	93.0	53.0	86.9/86.5	91.5/88.0	71.8
8-bit	Q-BERT	8-8-8	106 (×3.9)	83.9/83.8	-	-	92.9	-	-	-	-
	Q8BERT	8-8-8	106 (×3.9)	-/-	88.0/-	90.6	92.2	58.5	89.0/-	89.6/-	68.8
	8-bit BERT (ours)	8-8-8	106 (×3.9)	84.2/84.7	87.1/90.5	91.8	93.7	60.6	89.7/89.3	90.8/87.3	71.8
	8-bit TinyBERT (ours)	8-8-8	65 (×6.4)	84.4/84.6	87.9/91.0	91.0	93.3	54.7	90.0/89.4	91.2/87.5	72.2

- comparable performance as full-precision baseline, while being 15x smaller
- **under the same accuracy:** fewer parameters



Inference Acceleration

- Kirin 990 with BOLT
- https://github.com/huawei-noah/bolt
  - FP32: 3.5ms/seq
  - FP16: 2.0ms/seq
  - INT8: 1.3ms/seq



Weight- and activation-quantized networks

- have small storage and fast inference
- training can still be time-consuming

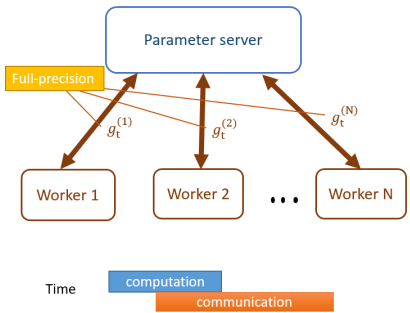
How to speed up training?

Solution:

- Distributed Training

Challenge:

- expensive communication cost incurred during synchronization of the gradients and model parameters



Distributed Low-precision Learning

(Hou et al., 2019)

[1] Hou et al., Analysis of quantized models. ICLR, 2019.

Framework

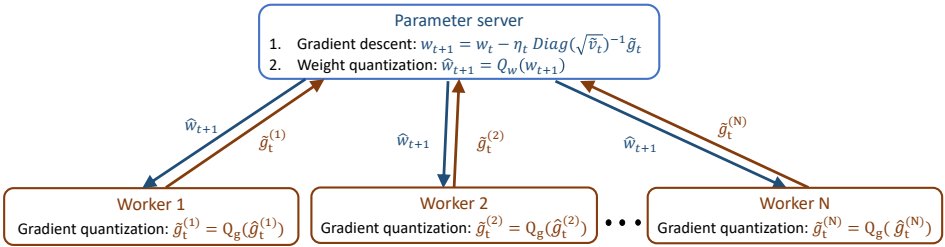


Figure: Distributed weight and gradient quantization with data parallelism.

Worker:

- Keeps low-precision weights only;
- Cheap forward and backward prop;
- Gradient quantization

Server:

- Keeps full-precision weights only
- Parameter update
- Weight quantization

Framework

Cheap communication

- worker-to-server: quantized weights
- server-to-worker: quantized gradients

Federated Learning

- cheap communication
- online learning
- quantized gradients have better privacy?

Theoretically Guaranteed convergence

- parameter server
- all-reduce: tree-based, ring-based

Convergence and Error

Weight Quantization with **full-precision Gradient**: $\frac{R(T)}{T} \leq \underbrace{O\left(\frac{d}{\sqrt{T}}\right)}_{speed} + \underbrace{LD\sqrt{D^2 + \frac{d\alpha^2\Delta_w^2}{4}}}_{error}$

Weight Quantization with **Quantized Gradient**: $\mathbf{E}(\|\tilde{\mathbf{g}}_t\|^2) \leq (\frac{1+\sqrt{2d-1}}{2}\Delta_g + 1)\|\hat{\mathbf{g}}_t\|^2$

Theorem: Convergence of LAQ with quantized gradient

- Speed: slowed down by a factor of $\sqrt{\frac{1+\sqrt{2d-1}}{2}\Delta_g + 1}$
- Error: no change

Problems:

i deep networks typically have a large d

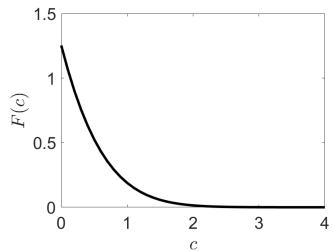
ii distributed learning prefers small number of bits for the gradients $\rightarrow$ large Δ_g

Weight Quantization with Quantized Clipped Gradient

$$\text{Clip}(\hat{g}_{t,i}) = \begin{cases} \hat{g}_{t,i} & |\hat{g}_{t,i}| \leq c\sigma \\ \text{sign}(\hat{g}_{t,i}) \cdot c\sigma & \text{otherwise} \end{cases}$$

Theorem: Convergence of LAQ with quantized clipped gradient

- Speed: slows down by a factor $\sqrt{(2/\pi)^{\frac{1}{2}} c\Delta_g + 1}$; independent of d
- Error: introduce extra error $\sqrt{d}D\sigma(2/\pi)^{\frac{1}{4}}\sqrt{F(c)}$



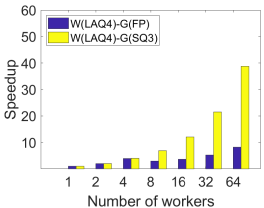
A larger c leads to

- smaller $F(c)$, and thus smaller error
- slower convergence

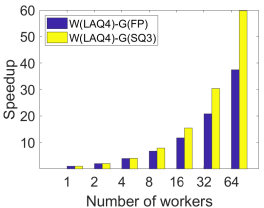
Empirically using c=3

Experiments: Speedup

- Baseline: training with one worker using full-precision gradient
- A 16-node GPU cluster and each node has 4 1080ti GPUs with one PCI switch
- Weight: 4-bit (LAQ4); Gradient: full-precision (FP) or 3-bit (SQ3)



1Gbps Ethernet.



10Gbps Ethernet.

- **small bandwidth:** communication is bottleneck! quantizing gradient much faster
- **larger bandwidth:** difference in speedups smaller

Thank you!

https://github.com/huawei-noah/Pretrained-Language-Model