

Notes

- Project Proposal is due tonight.
- Grade of Lab 1 has been posted.
- Midterm
 - Nov 4, in class.
 - Will cover materials up to this lecture (lecture 8)
 - Discuss about the coverage later this lecture
- Lab 3 will be posted this weekend.
- Lab 3 will be posted this weekend, and students are encouraged to work in groups of two.
- No quiz today.

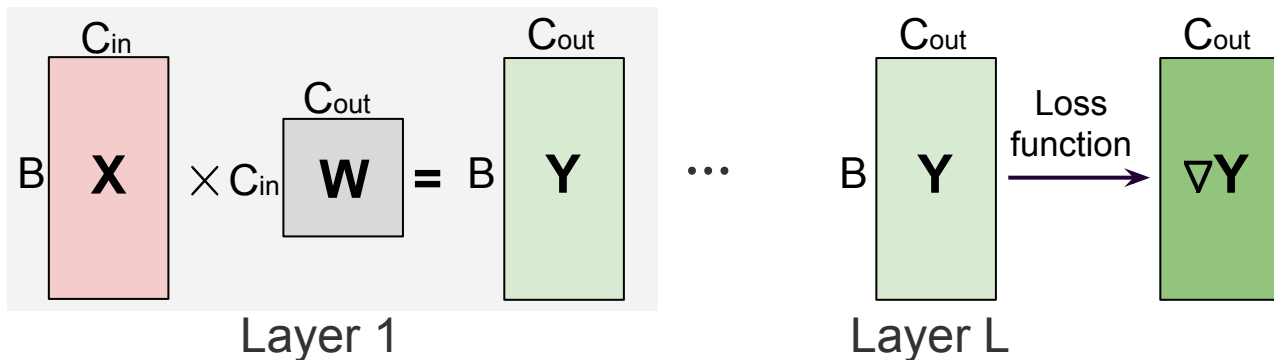
Recap

- Large Model Data Distribution
- Large Model Quantization
- Large Model Pruning
- Low-rank Decomposition for LLM

Topics

- Efficient training of DNNs
 - Efficient computing
 - Efficient storage
- Parameter efficient finetuning
- Federated learning

Forward Pass for Linear Layer



B : batch size

$\mathbf{X}$: input maps

C_{in} : input channels

$\mathbf{W}$: weight filters

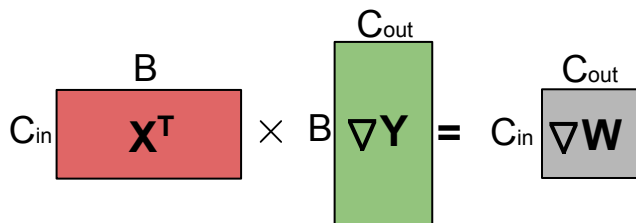
C_{out} : output channels

$\mathbf{Y}$: output maps

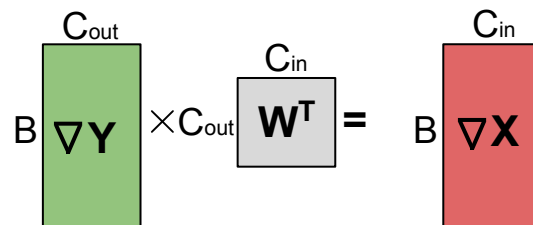
- The fully-connected layer during the forward propagation can be converted into matrix multiplications.

Backward Pass for Linear Layer

Weight Gradient Computation


$$\begin{matrix} C_{in} & B \\ \text{red box} & \text{X}^T \end{matrix} \times \begin{matrix} B & C_{out} \\ \text{green box} & \nabla Y \end{matrix} = \begin{matrix} C_{in} & C_{out} \\ \text{gray box} & \nabla W \end{matrix}$$

Data Gradient Computation


$$\begin{matrix} B & C_{out} \\ \text{green box} & \nabla Y \end{matrix} \times \begin{matrix} C_{out} & C_{in} \\ \text{gray box} & W^T \end{matrix} = \begin{matrix} B & C_{in} \\ \text{red box} & \nabla X \end{matrix}$$

X: input maps
∇X: input gradient

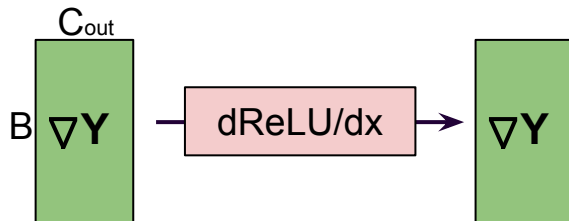
W: weight filters
∇W: weight gradient

Y: output maps
∇Y: output gradient

- DNN backward propagation involves two matrix multiplications

Backward Pass for Linear Layer

Data Gradient Computations



Weight Gradient Updates

$$C_{in} \begin{matrix} C_{out} \\ W \end{matrix} - \eta \times \begin{matrix} \nabla W \end{matrix} = \begin{matrix} W' \end{matrix}$$

X: input maps

∇X : input gradient

W: weight filters

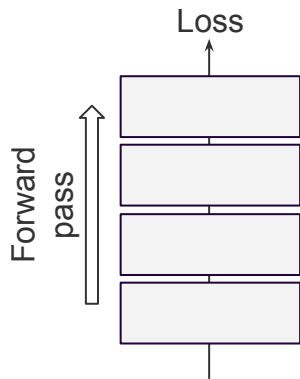
∇W : weight gradient

Y: output maps

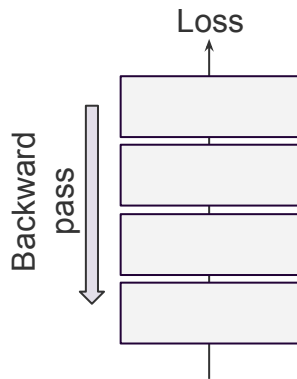
∇Y : output gradient

- DNN backward propagation involves two matrix multiplications

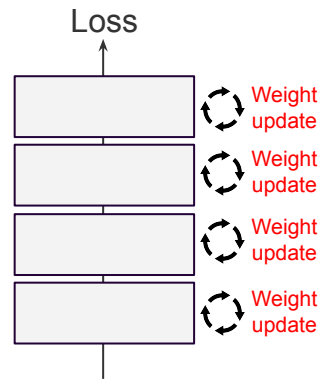
Training Process



```
def forward(self, x):  
    ...  
    return
```

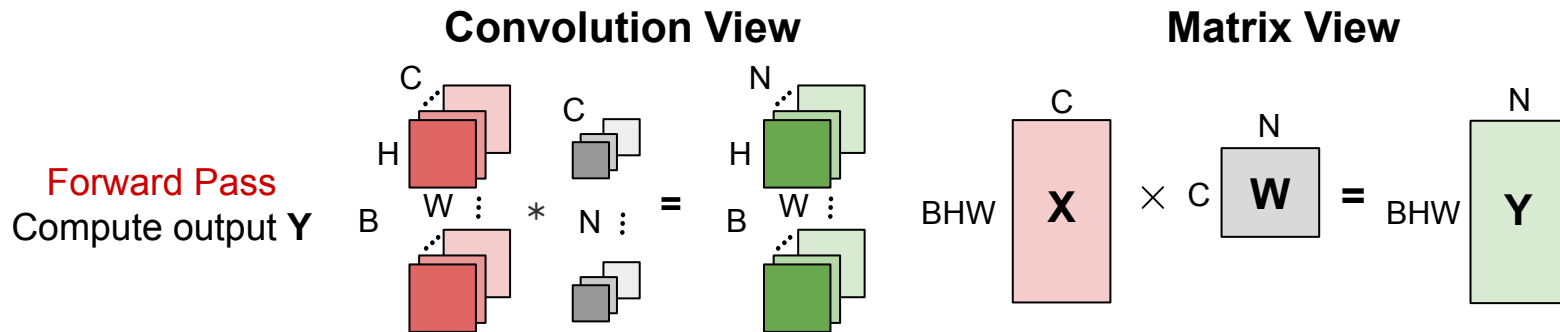


```
loss.backward()
```



```
optimizer.step()
```

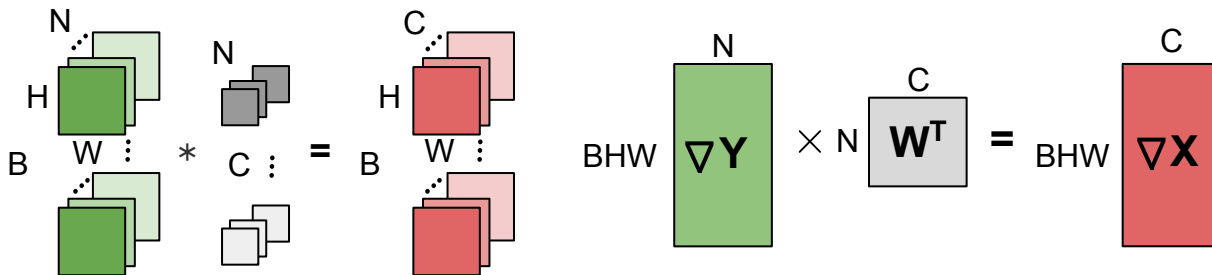
Forward Pass for Convolutional Layer



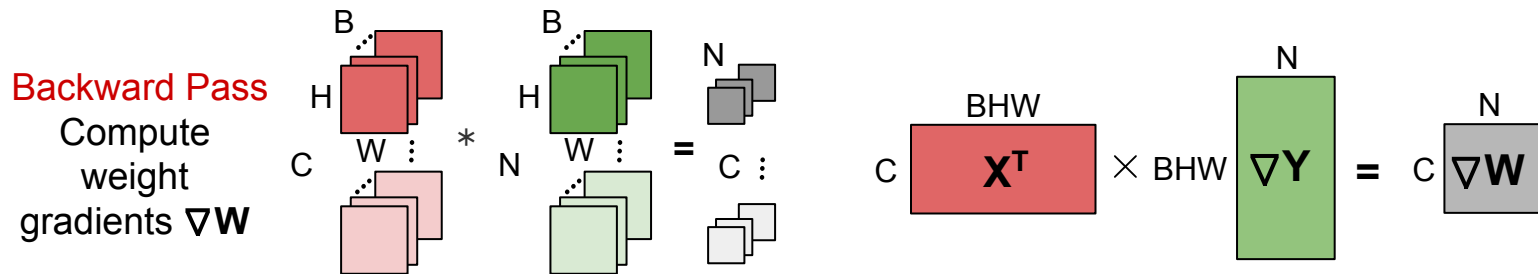
- Assume a weight kernel size of 1×1 .

Backward Pass for Convolutional Layer

Backward Pass
Compute Activation
gradients $\nabla \mathbf{X}$



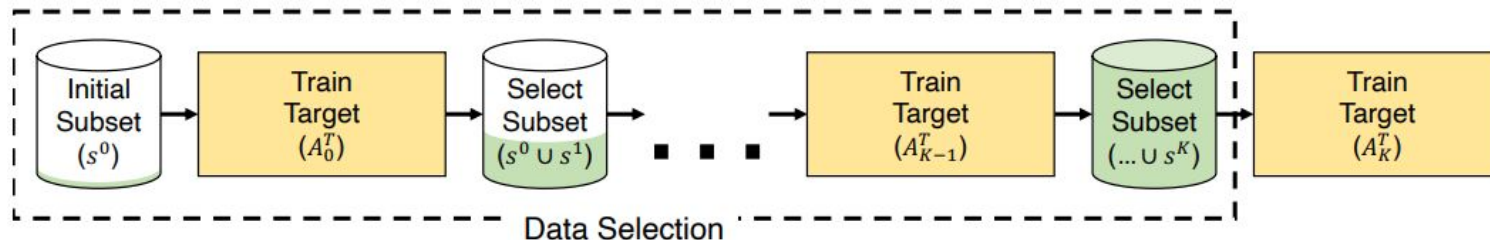
Backward Pass for Convolutional Layer



Efficient Computing during Training

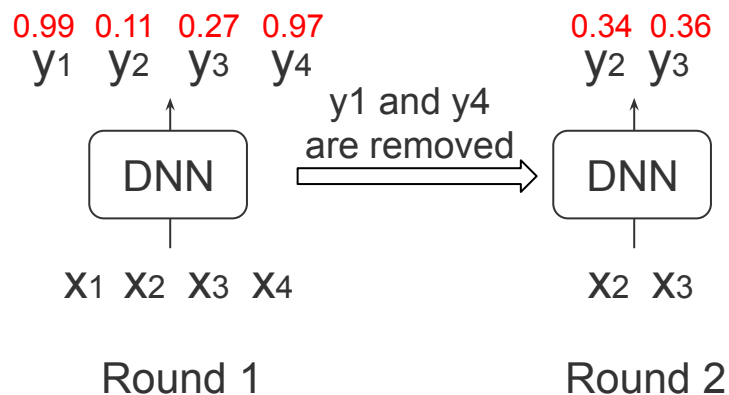
- To reduce the training cost of DNN, we can proceed from the following dimensions:
 - Training data sampling
 - Parameter sampling
 - Pruning during training
 - Quantization during training

Training Data Sampling for Efficiency



- Assume a total b samples are targeted to be picked. We consider a batch setting with K rounds where we select b/K points in every round.
- Training the target model with b/K samples, then evaluate the rest of the sample over the model. Find the batch with the least confidence score. Append it to the training dataset.

Training Data Sampling for Efficiency

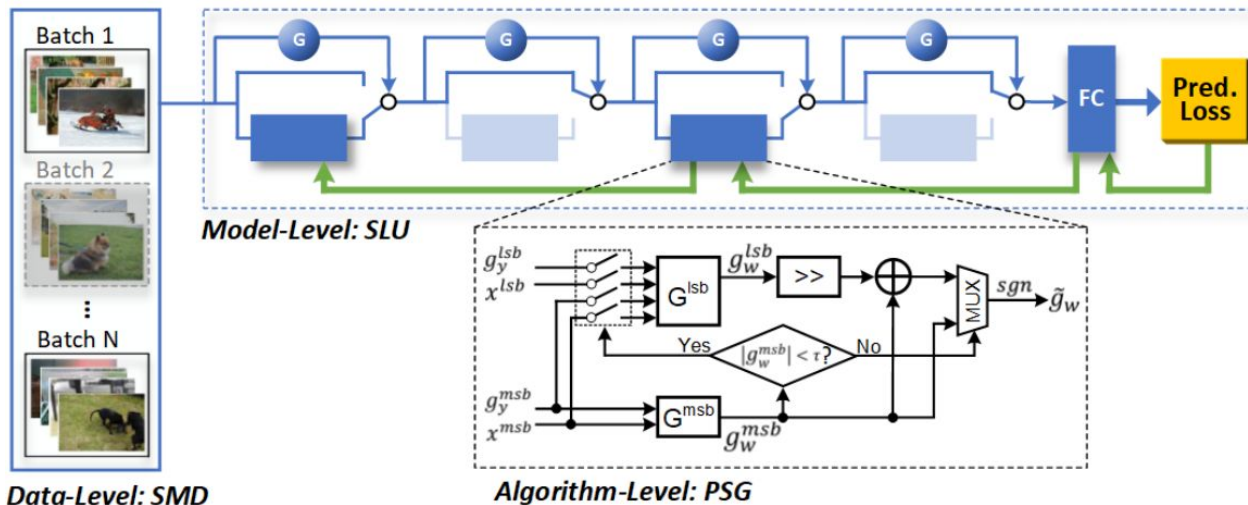


- Assume a total b samples are targeted to be picked. We consider a batch setting with K rounds where we select b/K points in every round.
- Training the target model with b/K samples, then evaluate the rest of the sample over the model. Find the batch with the least confidence score. Append it to the training dataset.

Efficient Computing during Training

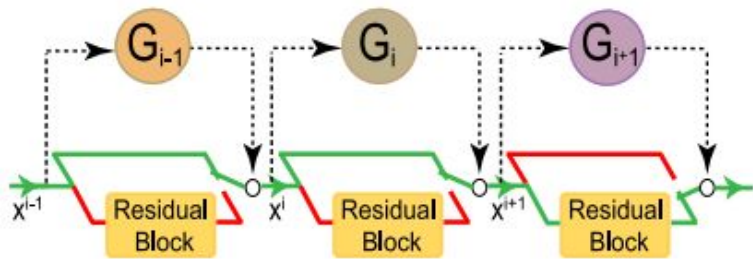
- To reduce the training cost of DNN, we can proceed from the following dimensions:
 - Training data sampling
 - **Parameter sampling**
 - Pruning during training
 - Quantization during training

E2-Train



- A stochastic mini-batch dropping strategy is proposed.
- Stochastic minibatch dropping simply skips every mini-batch with a default probability of 0.5.
- For some easy dataset, this will generate negligible impact on performance.

Dynamically Layer Skipping



- $G_i(x_i) \in \{0, 1\}$ is the gating function for layer i .
- It determines whether to skip to current residual block or not.
- During the training, G and residual blocks are trained together.
- $\text{Loss} = \text{acc_loss} + \text{computation_loss}$
- We will skip different layers adaptively based on inputs.

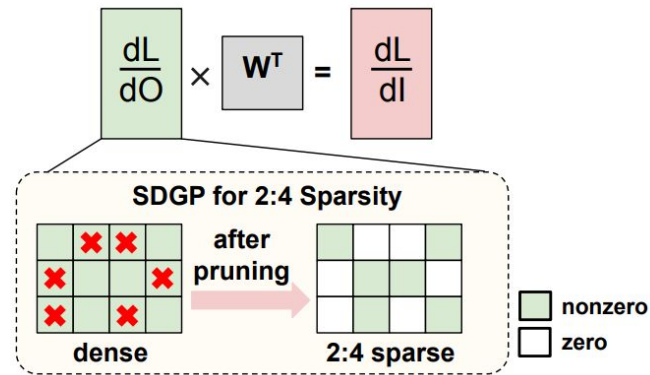
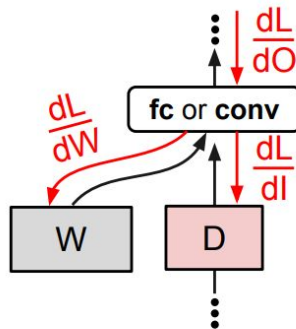
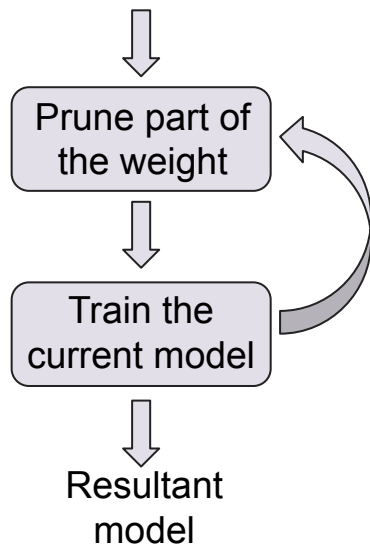
Wang, Xin, et al. "Skipnet: Learning dynamic routing in convolutional networks." *Proceedings of the European conference on computer vision (ECCV)*. 2018.

Wang, Yue, et al. "E2-train: Training state-of-the-art cnns with over 80% energy savings." *Advances in Neural Information Processing Systems* 32 (2019).

Efficient Computing during Training

- To reduce the training cost of DNN, we can proceed from the following dimensions:
 - Training data sampling
 - Parameter sampling
 - Pruning during training
 - Quantization during training

Pruning during Training



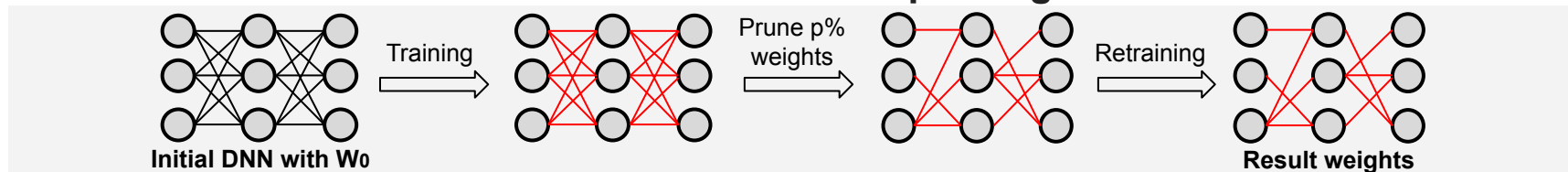
- We can remove the unnecessary weight during the DNN training process.

How to Find the Winning Tickets?

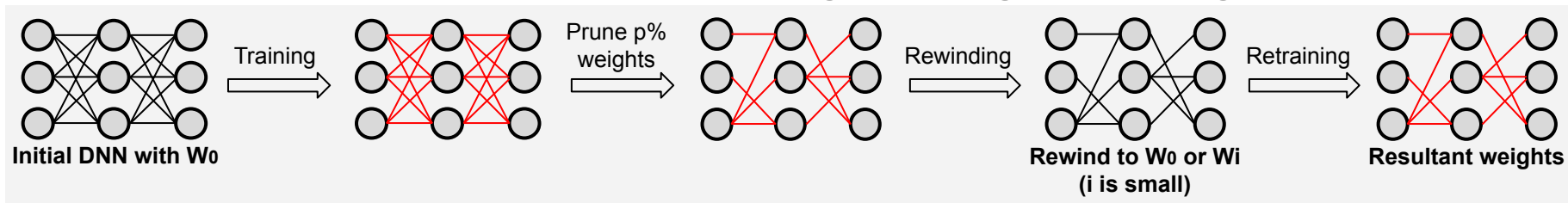
- **Iterative Magnitude Pruning (IMP):**
 - Initialized DNN with random weights w_0 .
 - While the sparsity level has not reached:
 - Train the DNN with k epochs until convergence
 - prune $p\%$ of the nonzero weights.
 - Reinitialize the remaining weights using the values in w_0 , finetune the remaining weights for k epochs (**Rewind**).
 - Return the weights.
- Later work has shown that rewind to w_i (i is small) works better for larger networks.

Weight Rewinding

Conventional iterative pruning



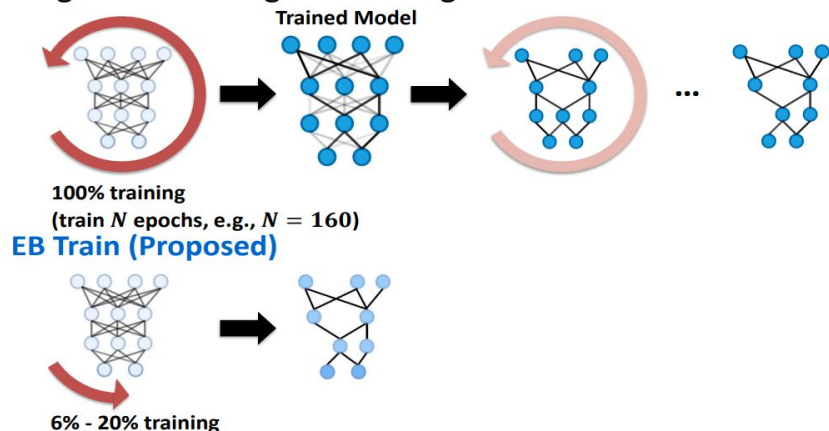
Conventional iterative pruning with weight rewinding



- The pruned architecture itself, rather than a set of inherited “important” weights, is more crucial to the accuracy in the final model, which suggests that in some cases pruning can be useful as an architecture search paradigm.

Early-bird Ticket

Progressive Pruning and Training



- LTH shows that there exist winning tickets (small but critical subnetworks) for dense, randomly initialized networks, that can be trained alone to achieve a comparable accuracy to the latter in a similar number of iterations.
- The winning tickets can be drawn very early in training and with aggressively low-cost training algorithms.
- Early-bird tickets can be founded via low-cost training schemes (e.g., early stopping and low-precision training) at large learning rates

Early-bird Ticket

Algorithm 1: The Algorithm for Searching EB Tickets

```
1: Initialize the weights  $\mathbf{W}$ , scaling factor  $r$ , pruning ratio  $p$ , and the FIFO queue  $Q$  with length  $l$ ;  
2: while  $t$  (epoch)  $< t_{max}$  do  
3:   Update  $\mathbf{W}$  and  $r$  using SGD training;  
4:   Perform structured pruning based on  $r_t$  towards the target ratio  $p$ ;  
5:   Calculate the mask distance between the current and last subnetworks and add to  $Q$ .  
6:    $t = t + 1$   
7:   if  $\text{Max}(Q) < \epsilon$  then → The mask pattern is stable  
8:      $t^* = t$   
9:     Return  $f(x; m_{t^*} \odot \mathbf{W})$  (EB ticket);  
10:  end if  
11: end while
```

- To search for the lottery ticket, we can early stop the DNN training.

Efficient Computing during Training

- To reduce the training cost of DNN, we can proceed from the following dimensions:
 - Training data sampling
 - Parameter sampling
 - Pruning during training
 - Quantization during training

DoReFaNet

Compute $g_{a_L} = \frac{\partial U}{\partial a_L}$ knowing a_L and a^* .

```
10: for  $k = L$  to 1 do
11:   Back-propagate  $g_{a_k}$  through activation function  $h$ 
12:    $g_{a_k}^b \leftarrow f_\gamma^G(g_{a_k})$ 
13:    $g_{a_{k-1}} \leftarrow \text{backward\_input}(g_{a_k}^b, W_k^b)$ 
14:    $g_{W_k^b} \leftarrow \text{backward\_weight}(g_{a_k}^b, a_{k-1}^b)$ 
15:   Back-propagate gradients through pooling layer if there is one
16: end for
    {2. Accumulating the parameters gradients:}
17: for  $k = 1$  to  $L$  do
18:    $g_{W_k} = g_{W_k^b} \frac{\partial W_k^b}{\partial W_k}$ 
19:    $W_k^{t+1} \leftarrow \text{Update}(W_k, g_{W_k}, \eta)$ 
20: end for
```

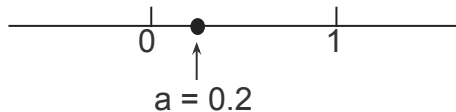
- Linear quantize the weights and activations
- Apply stochastic quantization for the gradients.

DoReFaNet

- Usually gradients requires far more bitwidth than weight and activation.
- Usually gradient requires **stochastic quantization**.

W	A	G	Training Complexity	Inference Complexity	Storage Relative Size	AlexNet Accuracy
1	1	6	7	1	1	0.395
1	1	8	9	1	1	0.395
1	1	32	-	1	1	0.279 (BNN)
1	1	32	-	1	1	0.442 (XNOR-Net)
1	1	32	-	1	1	0.401
1	1	32	-	1	1	0.436 (initialized)
1	2	6	8	2	1	0.461
1	2	8	10	2	1	0.463
1	2	32	-	2	1	0.477
1	2	32	-	2	1	0.498 (initialized)
1	3	6	9	3	1	0.471
1	3	32	-	3	1	0.484
1	4	6	-	4	1	0.482
1	4	32	-	4	1	0.503
1	4	32	-	4	1	0.530 (initialized)
8	8	8	-	-	8	0.530
32	32	32	-	-	32	0.559

Deterministic and Stochastic Quantization



- To quantize a , conventional linear quantization will make $q(a) = 0$. However, this will cause a bias.
- With stochastic quantization:

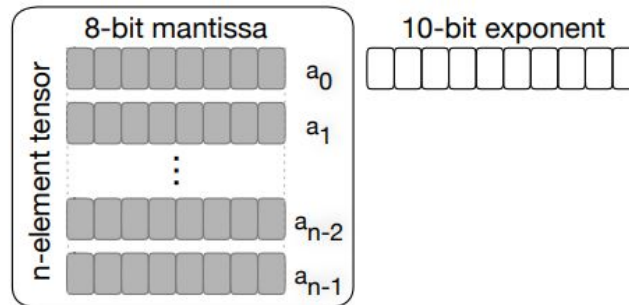
$$q(a) = \begin{cases} 1 & \text{for } p = 0.2 \\ 0 & \text{for } p = 0.8 \end{cases}$$

- Stochastic quantization is extremely useful when applying quantization to accelerate DNN training.

Training DNNs with Hybrid BFP



(a) FP repr. with an exponent per tensor element.

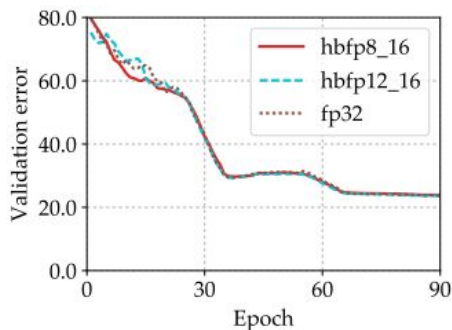
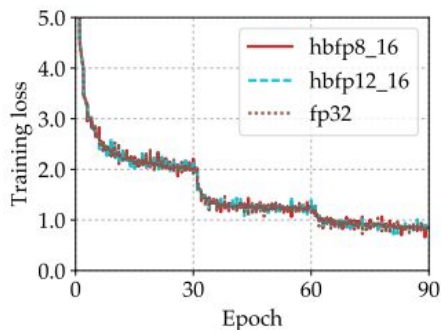


(b) BFP repr. with an exponent per tensor.

- Block floating point format achieves a better hardware efficiency and comparable representation capability than FP.

Training DNNs with Hybrid BFP

- Use BFP in all dot-product-based operations present in DNNs (i.e., convolutions, matrix multiplications, and outer products), and floating-point representations for all other operations (i.e., activations, regularizations, etc).
- To minimize data loss in long-lasting training state, the weights are stored with wider mantissas.



- ResNet-50 trained on ImageNet for 90 epochs.
- 8 bit mantissa, 16 bits weight seems to achieve comparable performance as FP32. A mantissa bitwidth of 12 achieves an even better performance.
- A tile size of 24.
- Weight update is performed in FP32.

Two Copies of Weights

Input data gradient Computation

$$\begin{matrix} C_{out} \\ B \end{matrix} \begin{matrix} \nabla Y \end{matrix} \times_{C_{out}} \begin{matrix} C_{in} \\ W^T \end{matrix} = \begin{matrix} C_{in} \\ B \end{matrix} \begin{matrix} \nabla X \end{matrix}$$

Weight Gradient Computation

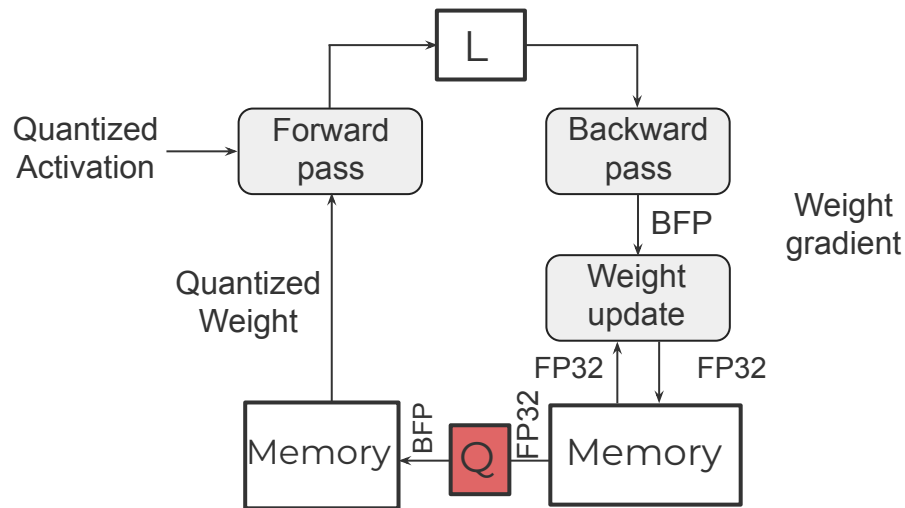
$$\begin{matrix} B \\ C_{in} \end{matrix} \begin{matrix} X^T \end{matrix} \times_B \begin{matrix} C_{out} \\ \nabla Y \end{matrix} = \begin{matrix} C_{out} \\ C_{in} \end{matrix} \begin{matrix} \nabla W \end{matrix}$$

Weight Gradient Updates

$$\begin{matrix} C_{out} \\ C_{in} \end{matrix} \begin{matrix} W \end{matrix} - \eta \times \begin{matrix} \nabla W \end{matrix} = \begin{matrix} W' \end{matrix}$$

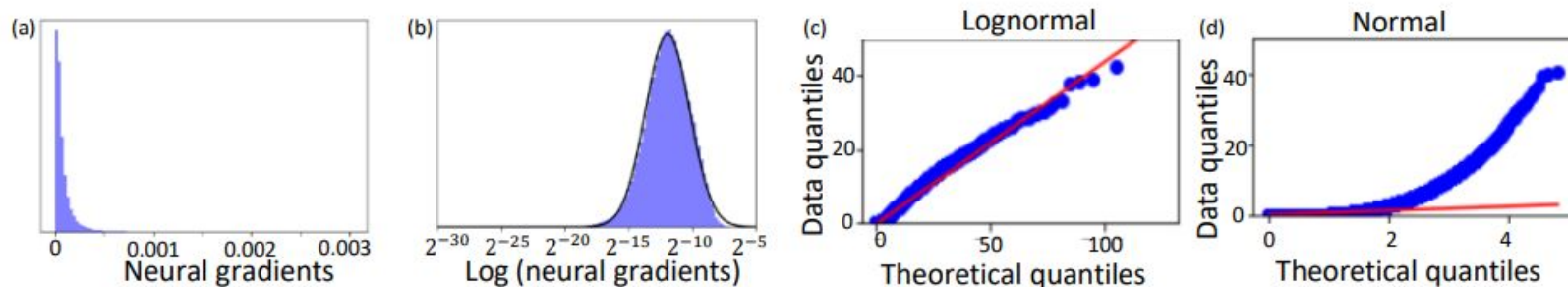
- Gradient and forward propagation are performed using BFP.
- Weights are updated using FP.
- Two copies of weights are used.

Two Copies of Weights



- Two pieces of copies are needed to be kept in the memory.
- The weight updates are usually performed with higher precision (e.g., FP16).

Neural Gradients are Near-Lognormal: Improved Quantized and Sparse Training

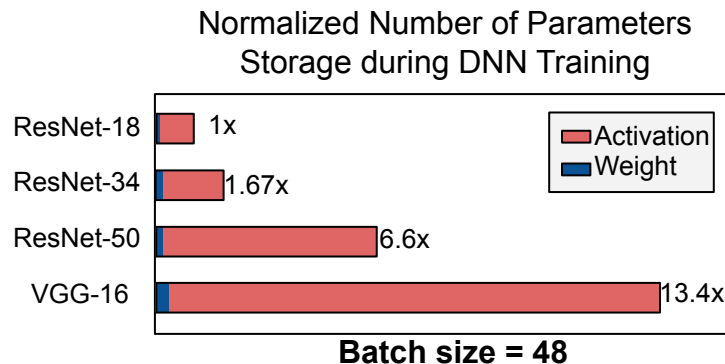
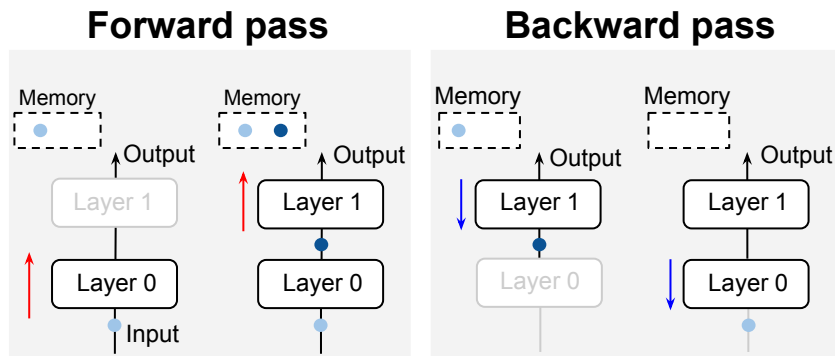


- The distribution of neural gradients is approximately lognormal.
- We can use lognormal regression to determine the optimal quantization setting (e.g., bitwidth, quantization interval).

Topics

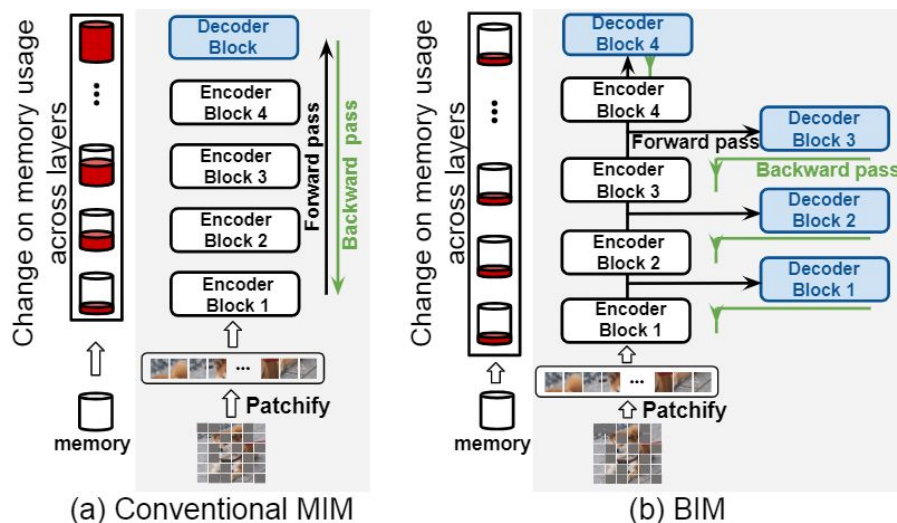
- Efficient training of DNNs
 - Efficient computing
 - **Efficient storage**
- Parameter efficient finetuning
- Federated Learning

Memory Consumption During Training



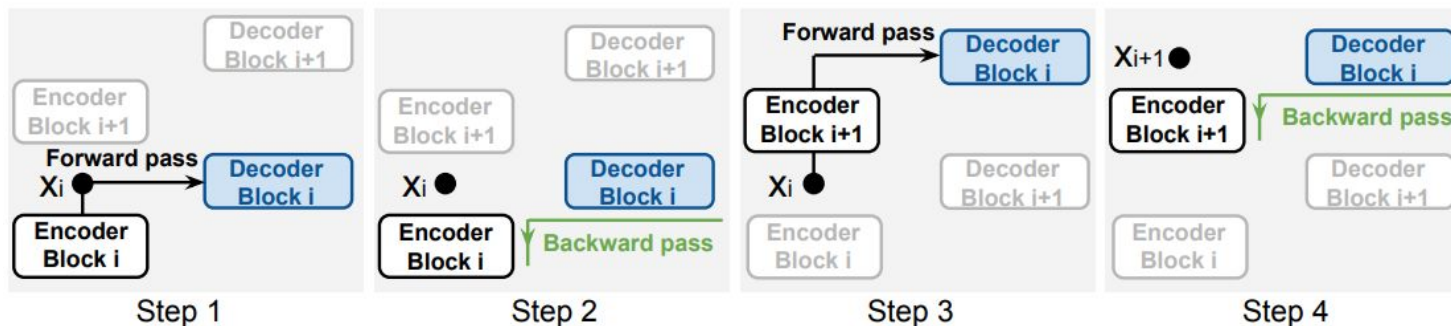
- The memory footprint grows proportional with the layer depth. The activation in the early DNN layers need to be stored for a long time.
- Activations consume most of the memory space, approximately 13 times larger than the weights on average.

BIM: Block-Wise Local Learning with Masked Image Modeling



- Local exist is introduced during the training process.
- The intermediate results can be discarded once the training process for the current layer is complete.

BIM: Block-Wise Local Learning with Masked Image Modeling



- Once the parameter updates in encoder block i and decoder block i are finished, all intermediate features stored in the buffer, except for x_i , can be cleared from memory, preserving them for future use.

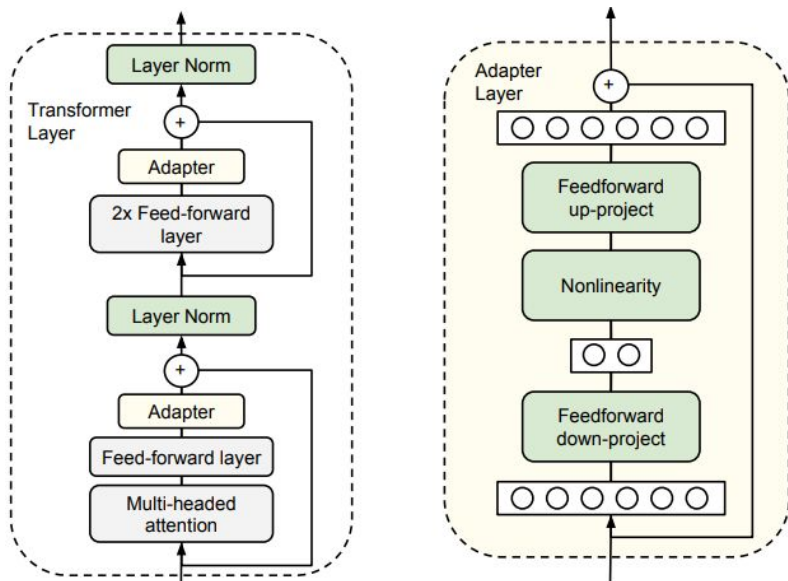
Topics

- Efficient training of DNNs
 - Efficient computing
 - Efficient storage
- **Parameter efficient finetuning**
- Federated learning

Parameter-efficient Finetuning (PEFT)

- Large models (LMs), often consisting of billions of parameters, require vast amounts of computational resources for execution.
- The expansive scale and computational demands pose considerable challenges when customizing them for particular downstream tasks.
- To better adapt the LMs over the downstream tasks, we can finetune a small portion of the LM parameters. This will make LMs achieve great performance over the downstream tasks while minimizing the training cost.
- Some of the popular PEFT Algorithms:
 - LoRA
 - Adapter
 - BitFit

Parameter-Efficient Transfer Learning for NLP



- We add the adapter module twice to each Transformer layer.
- The adapter consists of a bottleneck which contains few parameters relative to the attention and feedforward layers in the original model. The adapter also contains a skip-connection.
- The learnable parameters contributes to around 0.5 – 8% of the parameters of the original model.

BitFit

$$\mathbf{Q}^{m,\ell}(\mathbf{x}) = \mathbf{W}_q^{m,\ell} \mathbf{x} + \mathbf{b}_q^{m,\ell}$$

$$\mathbf{K}^{m,\ell}(\mathbf{x}) = \mathbf{W}_k^{m,\ell} \mathbf{x} + \mathbf{b}_k^{m,\ell}$$

$$\mathbf{V}^{m,\ell}(\mathbf{x}) = \mathbf{W}_v^{m,\ell} \mathbf{x} + \mathbf{b}_v^{m,\ell}$$

$$\mathbf{h}_2^\ell = \text{Dropout}(\mathbf{W}_{m_1}^\ell \cdot \mathbf{h}_1^\ell + \mathbf{b}_{m_1}^\ell)$$

$$\mathbf{h}_3^\ell = \mathbf{g}_{LN_1}^\ell \odot \frac{(\mathbf{h}_2^\ell + \mathbf{x}) - \mu}{\sigma} + \mathbf{b}_{LN_1}^\ell$$

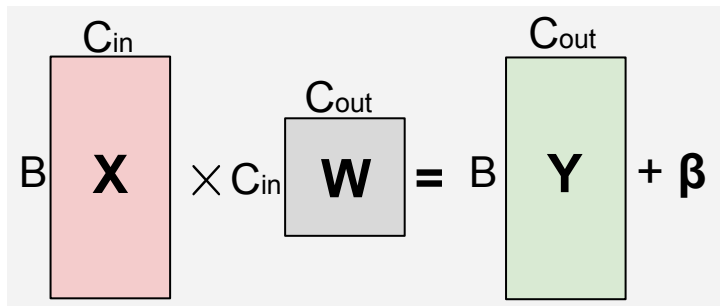
$$\mathbf{h}_4^\ell = \text{GELU}(\mathbf{W}_{m_2}^\ell \cdot \mathbf{h}_3^\ell + \mathbf{b}_{m_2}^\ell)$$

$$\mathbf{h}_5^\ell = \text{Dropout}(\mathbf{W}_{m_3}^\ell \cdot \mathbf{h}_4^\ell + \mathbf{b}_{m_3}^\ell)$$

$$\text{out}^\ell = \mathbf{g}_{LN_2}^\ell \odot \frac{(\mathbf{h}_5^\ell + \mathbf{h}_3^\ell) - \mu}{\sigma} + \mathbf{b}_{LN_2}^\ell$$

- BitFit is a sparse-finetuning method where only the bias-terms of the model are being modified.
- Applying BitFit on pre-trained BERT models is competitive with (and sometimes better than) fine-tuning the entire model.
- Bias parameters make up 0.09% of the total number of parameters in BERT.

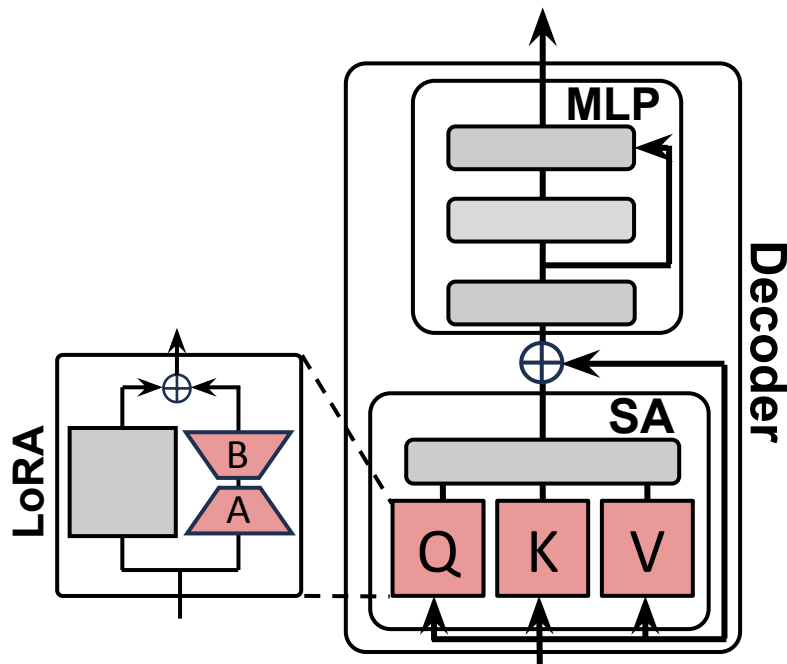
Finetune Bias is Cheap



$$\frac{dL}{d\beta} = \sum_{y \in Y} \frac{dL}{dy}$$

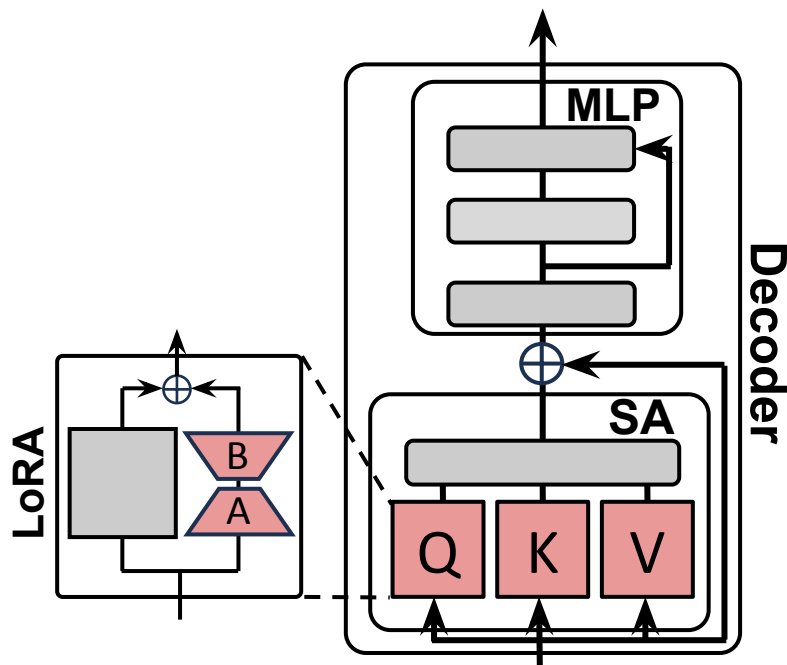
- Updating the bias does not require buffering any intermediate results during the forward pass of DNN training.

Low-rank Adaptation (LoRA)



- **LoRA (Low-Rank Adaptation)** is a **PEFT** method for large pre-trained models. Instead of updating all model weights during fine-tuning, LoRA inserts small trainable low-rank matrices into specific layers (usually the attention projections).
- This dramatically reduces memory and compute requirements while maintaining near full fine-tuning performance.

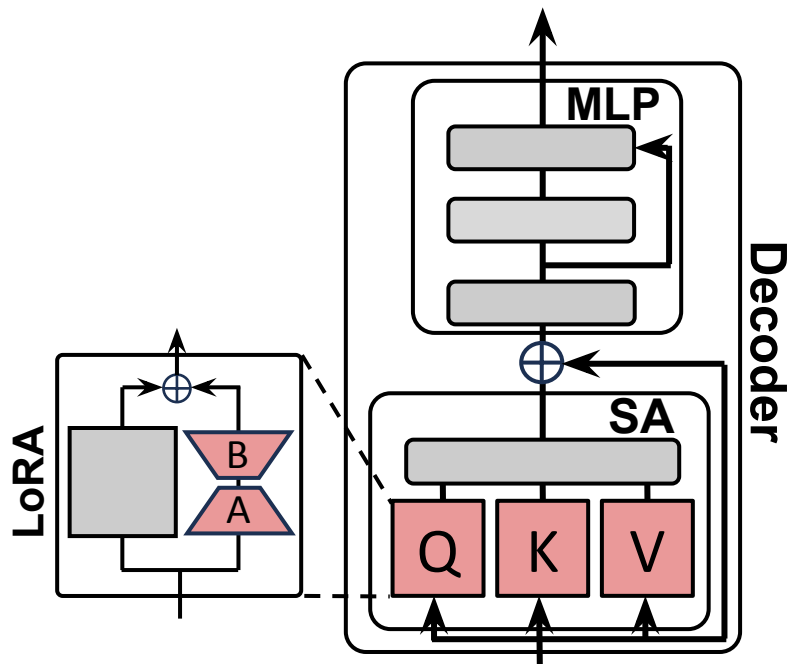
Low-rank Adaptation (LoRA)



$$h = W_0x + \Delta Wx = W_0x + BAx$$

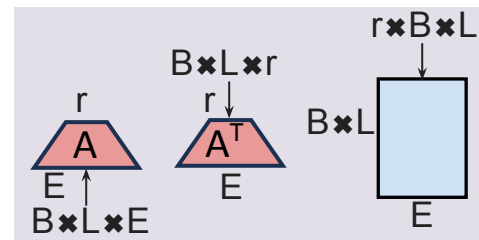
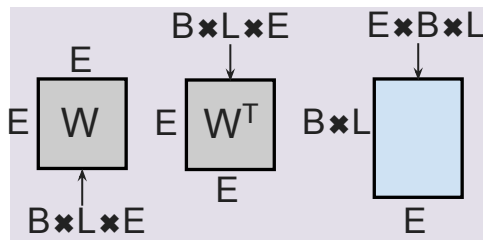
- Only the weights within the red blocks are updated.
- Assume the weight matrix has a dimension of $E \times E$, A and B have a size of $E \times r$ and $r \times E$, where $r \ll k$ (e.g., $r=4$).
- BA can be merged with the original weight W_0 , leading to no additional computational and storage cost.

Low-rank Adaptation (LoRA)



$$h = W_0x + \Delta Wx = W_0x + BAx$$

- Compared with finetuning the entire W_Q , W_K and W_V , this will lead to great compute savings:
 - $3BLE^2$
 - $6BLrE$



Low-rank Adaptation (LoRA)

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB _{base} (FT)*	125.0M	87.6	94.8	90.2	63.6	92.8	91.9	78.7	91.2	86.4
RoB _{base} (BitFit)*	0.1M	84.7	93.7	92.7	62.0	91.8	84.0	81.5	90.8	85.2
RoB _{base} (Adpt ^D)*	0.3M	87.1 $\pm$ 0.0	94.2 $\pm$ 1.1	88.5 $\pm$ 1.1	60.8 $\pm$ 4.4	93.1 $\pm$ 1.1	90.2 $\pm$ 0.0	71.5 $\pm$ 2.7	89.7 $\pm$ 3.3	84.4
RoB _{base} (Adpt ^D)*	0.9M	87.3 $\pm$ 1.1	94.7 $\pm$ 3.3	88.4 $\pm$ 1.1	62.6 $\pm$ 9.9	93.0 $\pm$ 2.2	90.6 $\pm$ 0.0	75.9 $\pm$ 2.2	90.3 $\pm$ 1.1	85.4
RoB _{base} (LoRA)	0.3M	87.5 $\pm$ 3.3	95.1$\pm$2.2	89.7 $\pm$ 7.7	63.4 $\pm$ 1.2	93.3$\pm$3.3	90.8 $\pm$ 1.1	86.6$\pm$7.7	91.5$\pm$2.2	87.2
RoB _{large} (FT)*	355.0M	90.2	96.4	90.9	68.0	94.7	92.2	86.6	92.4	88.9
RoB _{large} (LoRA)	0.8M	90.6$\pm$2.2	96.2 $\pm$ 5.5	90.9$\pm$1.2	68.2$\pm$1.9	94.9$\pm$3.3	91.6 $\pm$ 1.1	87.4$\pm$2.5	92.6$\pm$2.2	89.0
RoB _{large} (Adpt ^P)†	3.0M	90.2 $\pm$ 3.3	96.1 $\pm$ 3.3	90.2 $\pm$ 7.7	68.3$\pm$1.0	94.8$\pm$2.2	91.9$\pm$1.1	83.8 $\pm$ 2.9	92.1 $\pm$ 7.7	88.4
RoB _{large} (Adpt ^P)†	0.8M	90.5$\pm$3.3	96.6$\pm$2.2	89.7 $\pm$ 1.2	67.8 $\pm$ 2.5	94.8$\pm$3.3	91.7 $\pm$ 2.2	80.1 $\pm$ 2.9	91.9 $\pm$ 4.4	87.9
RoB _{large} (Adpt ^H)†	6.0M	89.9 $\pm$ 5.5	96.2 $\pm$ 3.3	88.7 $\pm$ 2.9	66.5 $\pm$ 4.4	94.7 $\pm$ 2.2	92.1 $\pm$ 1.1	83.4 $\pm$ 1.1	91.0 $\pm$ 1.7	87.8
RoB _{large} (Adpt ^H)†	0.8M	90.3 $\pm$ 3.3	96.3 $\pm$ 5.5	87.7 $\pm$ 1.7	66.3 $\pm$ 2.0	94.7 $\pm$ 2.2	91.5 $\pm$ 1.1	72.9 $\pm$ 2.9	91.5 $\pm$ 5.5	86.4
RoB _{large} (LoRA)†	0.8M	90.6$\pm$2.2	96.2 $\pm$ 5.5	90.2$\pm$1.0	68.2 $\pm$ 1.9	94.8$\pm$3.3	91.6 $\pm$ 2.2	85.2$\pm$1.1	92.3$\pm$5.5	88.6
DeB _{XXL} (FT)*	1500.0M	91.8	97.2	92.0	72.0	96.0	92.7	93.9	92.9	91.1
DeB _{XXL} (LoRA)	4.7M	91.9$\pm$2.2	96.9 $\pm$ 2.2	92.6$\pm$6.6	72.4$\pm$1.1	96.0$\pm$1.1	92.9$\pm$1.1	94.9$\pm$4.4	93.0$\pm$2.2	91.3

- LoRA achieves better results than Adapter and BitFit.

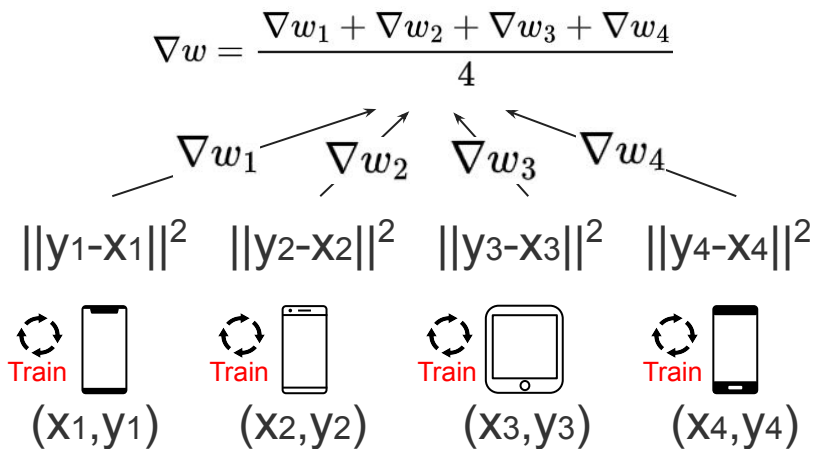
Topics

- Efficient training of DNNs
 - Efficient computing
 - Efficient storage
- Parameter efficient finetuning
- Federated learning

Federated Learning

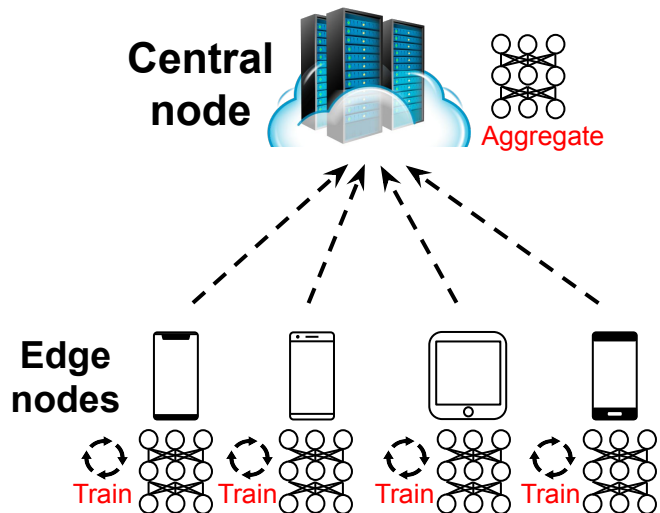
- Training data: $(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$

$$\sum_{i=1}^4 \|y_i - F(x_i)\|^2$$



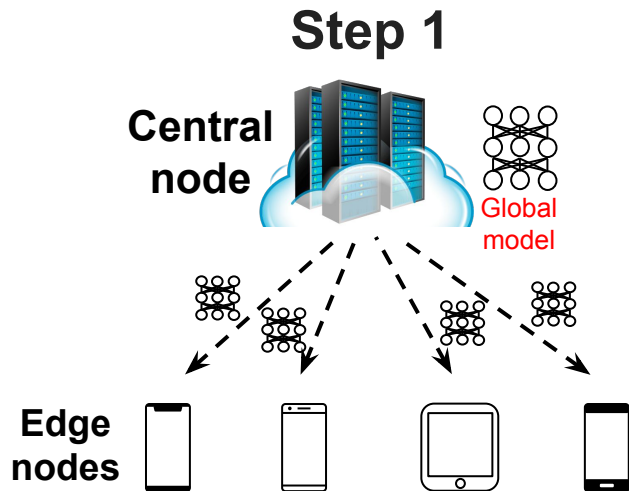
- Non-iid training data distribution
- Heterogeneity among the edge devices
- Communication error

Federated Learning



- Federated learning is a machine learning technique that allows the training of models across multiple decentralized nodes holding local data samples, **without exchanging their data.**
- This approach enhances privacy, user can train the powerful DNN model **without sharing the dataset.**

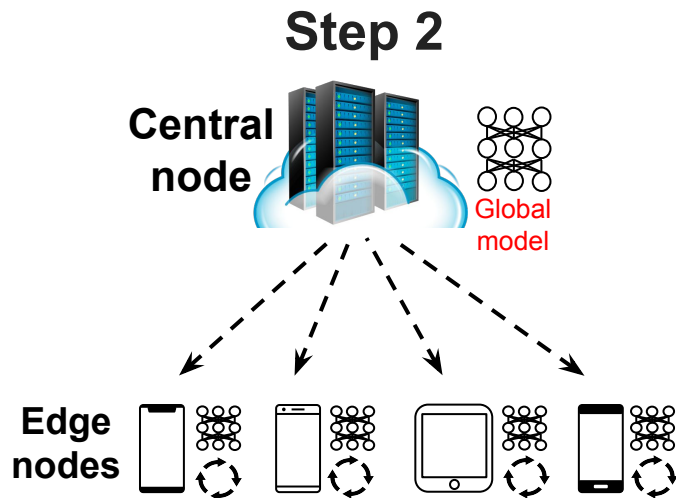
FedAvg



- A global model is initialized on the central node and sent to all participating nodes .

$$w_i = w_{global} \quad \text{For each } i$$

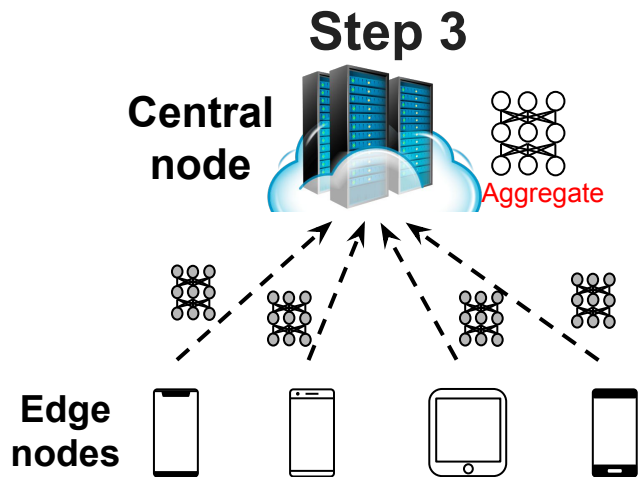
FedAvg



$$w'_i = \min_{w_i} L(F(D_i), Y_i)$$

- Each node i trains the global model locally using its own data for a few epochs.
- The length of local training process may vary.

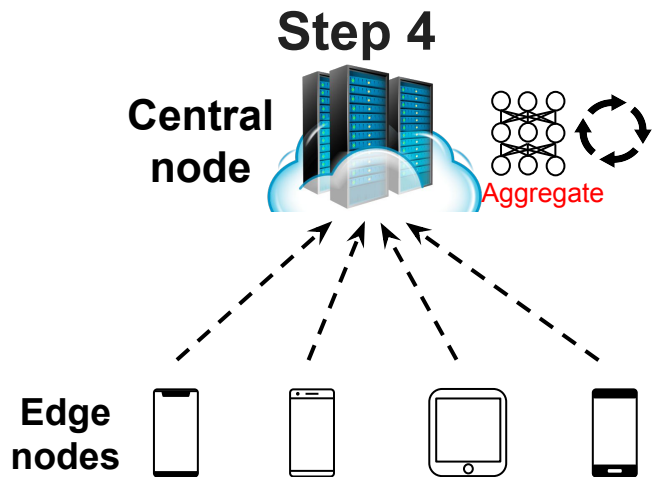
FedAvg



$$\Delta w_i = w - w_i$$

- Local updates are sent from each node to the central node.

FedAvg

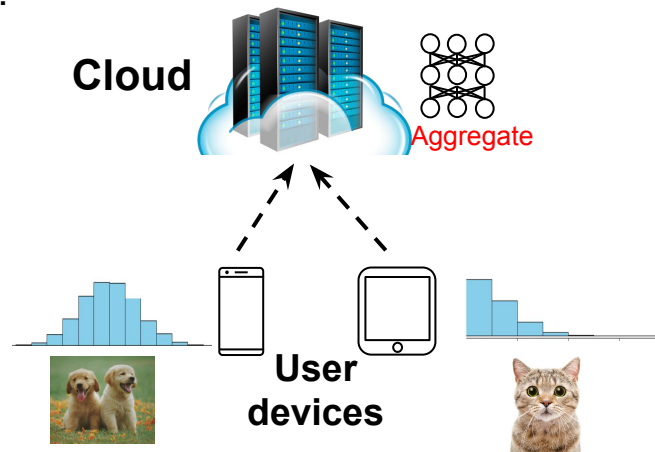


- The central node aggregates the local updates to update the global model.

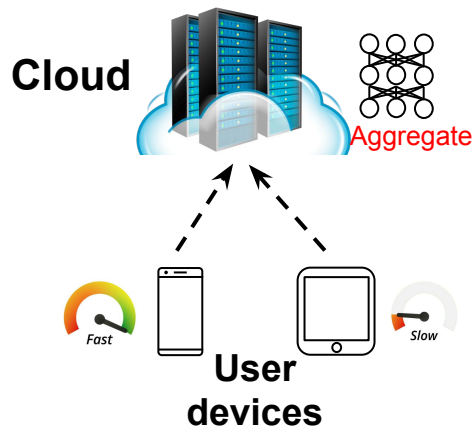
$$w + \frac{1}{N} \sum_i \Delta w_i$$

Federated Learning Problems: Non-IID

- However, in FL, the data distributed across different devices or clients is not drawn from the same statistical distribution.
- Unlike the scenario distributed training, where the training data are randomly distributed. For FL, the data stored in each device is highly biased.
- This may lead to significant accuracy degradation for the global model.

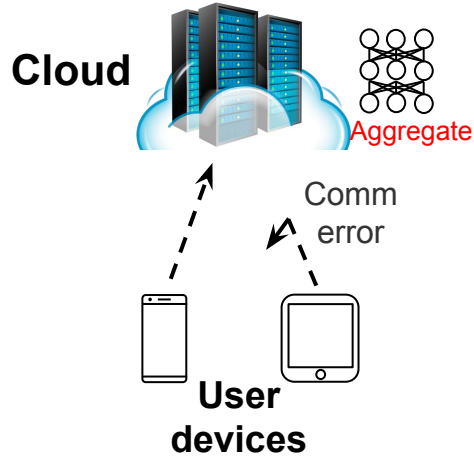


Federated Learning Problems: Heterogeneity



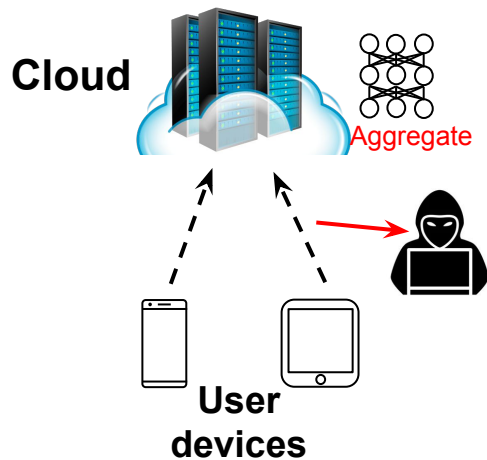
- Different edge device may have different processing speed.
- This will cause the total latency of each training round bottlenecked by the straggler, leading to a slow convergence of the training process.

Federated Learning Problems: Communication



- The communication between edge devices and central cloud may incur transmission loss or error.
- This will impact the training latency and accuracy.

Federated Learning Problems: Privacy



- The attacker can leverage the transmitted gradient to reconstruct the original input training data.
- This will lead to privacy leakage.

