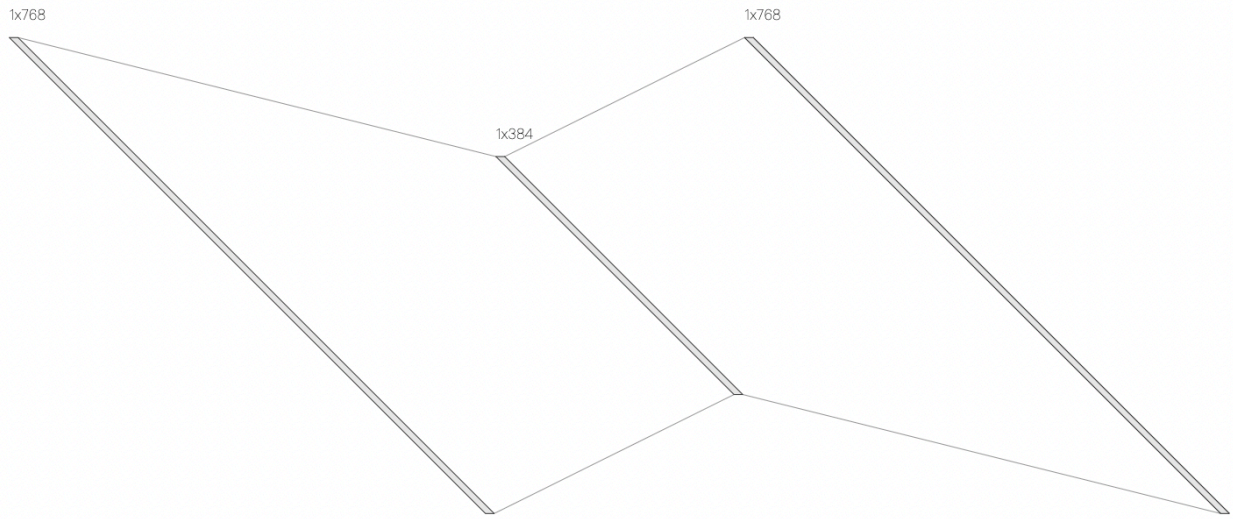
1. InsertLinear



1.1 Decaying linear

Settings

Backend	Roberta-base
Dataset	COLA
Lr	2e-5
Optimizer	AdamW
Scheduler	Polynomial
batch size	32

Partition strategy	Part1(client)	Part2(server)	Part3(client)
Type 1	Self attention+Dense	Fastfeedforward+ Roberta[1:11]	Classifier

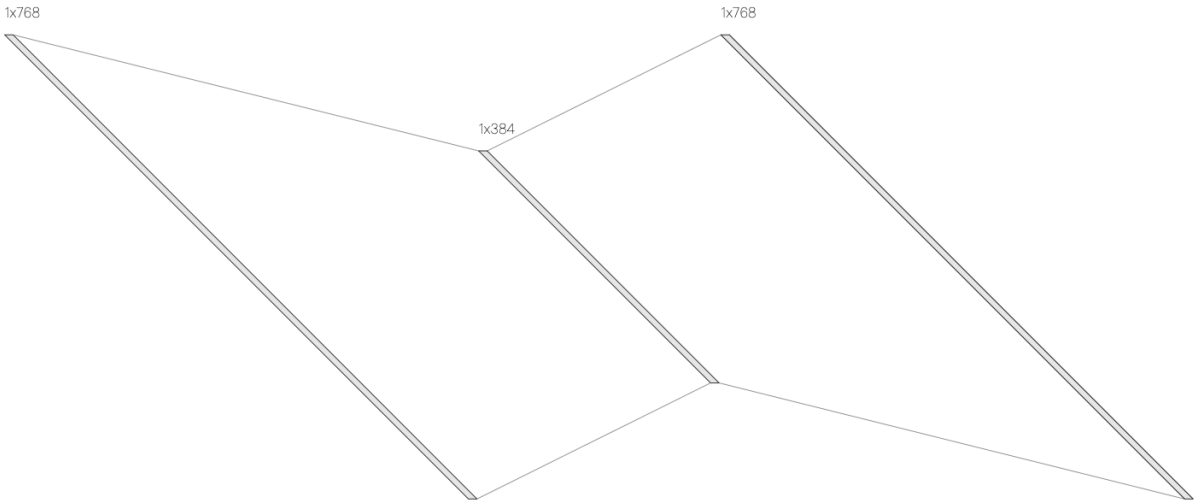
I am testing inserting linear to minimize the activation memory size needed to transfer.

A problem is that when I insert four Random initialize layers with a size of (768,768), The model can not be finetuned to a acceptable performance.

One solution is to embed eye initialization, which creates diagonal matrixs(and then slice them). However, this method can not afford a bigger compression rate. Even compress from 768 to half is not possible.

Here we design two ways.

1. Slowly decay the out_feature of first linear layer and in_feature of second layer.(Decaying Linear)



2. Finetune two linear layer to some big datasets and learn the activation memory.(imitate diagonal matrix)

1.2 Decaying Linear

We decay the `in_feature` and the `out_feature` per constant number of steps with same rate

$$feature\_size = feature\_size\_old * rate$$

In order to make the matrix going to be deleted sparse, we implement L1 regularization to those weights going to be deleted.

Here are the results.

$$compress\_ratio = feature * rate^{iteration} / feature$$

Steps	Rate	Iteration	compress rate	Acc	L1 reg
100	1.0	9	1.0	85.1	No
100	0.9	9	0.387	82.9	No
200	0.9	9	0.387	83.8	No
300	0.9	9	0.387	84.3	No

Longer steps give better results

Steps	Rate	Iteration	compress rate	Acc	L1 reg
500	0.83	5	0.401	83.9	No
300	0.9	9	0.387	84.3	No

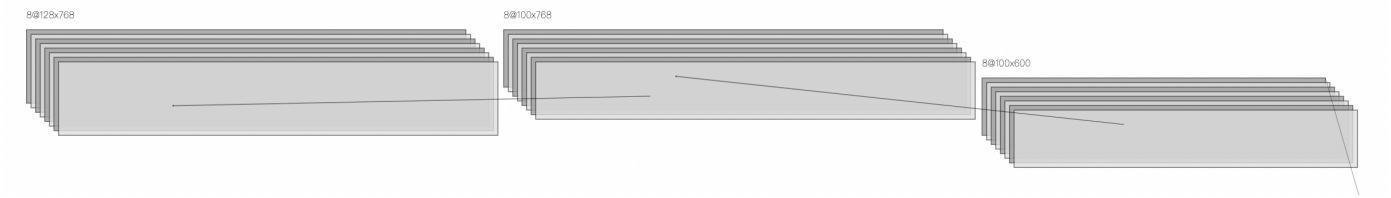
Bigger rates get better results

When using l1 regularization, the result does not get much better. And L1 regularization gets sparse weights but not zero weights.

L1 reg here means the value of lambda

Steps	Rate	Iteration	compress rate	Acc	L1 reg
100	1.0	9	1.0	85.3	1e-2
100	1.0	9	1.0	85.1	1e-4
100	0.9	9	0.387	83.1	1e-4
300	0.9	9	0.387	84.6	1e-4
500	0.83	5	0.394	84.3	1e-4

Then I use two sets of decaying linear inserting on client and server.



$$feature\_size_1 = feature\_size\_old_1 * rate_1(last\ dimention)$$
$$feature\_size_2 = feature\_size\_old_2 * rate_2(second\ dimention)$$

Steps	Rate1	Rate2	Iteration	compress rate	Acc	L1 reg
300	0.9	0.95	9	0.244	83.4	1e-4
500	0.9	0.96	9	0.244	83.8	1e-4
500	0.9	0.96	8	0.285	83.4	1e-4

Here is the implementation of my decaying linear

https://github.com/timmywanttolearn/gpipe_test/blob/master/layer_insertion/NLP/models/decaying_linear.py

```
# Y = XW.T
class DecaylinearFunctionFirst(F):
    @staticmethod
```

```
    def forward(ctx, input, weight, rank, l1):
        ctx.input = input
        ctx.weight = weight
        ctx.rank, ctx.l1 = rank, l1
        output = input @ weight.t()
        return output
    @staticmethod
    def backward(ctx, grad_output):
        input = ctx.input
        weight = ctx.weight
        rank, l1 = ctx.rank, ctx.l1
        grad_input = grad_output @ weight
        grad_weight = grad_output.transpose(-1, -2) @ input
        # do l1 regulization in backward
        l1_reg = -l1 * grad_weight[rank:,:]  / torch.abs(grad_weight[rank:,:])
        l1_reg = torch.nan_to_num(l1_reg,0.0,0.0,0.0)
        grad_weight[rank:,:] += l1_reg
        return grad_input, grad_weight, None, None
```

And also I put a slice of the total matrix in linear Function in order to avoid rebuilding optimizer

```
class DecayLinearFirst(nn.Module):
    def __init__(self, in_features, decay_rate, step, step_stop) -> None:
        super(DecayLinearFirst, self).__init__()
        self.weight = nn.Parameter(torch.eye(in_features))
        self.decay_rate = decay_rate
        self.step = step
        self.step_stop = step_stop
        self.iter = 0
        self.rank = in_features
        self.rank1 = int(in_features * decay_rate) # l1 reg weights
    def forward(self, input):
        if self.training is True:
            self.iter += 1
        if self.iter % self.step == self.step - 1 and self.iter <= self.step_stop:
            self.rank = int(self.rank * self.decay_rate)
            if self.iter < self.step_stop - 30:
                self.rank1 = int(self.rank1 * self.decay_rate)
        return DecaylinearFunctionFirst.apply(
            input, self.weight[: self.rank, :], self.rank1, 1e-2
        )
```

2 Power Iteration

2.1 Settings

Backend	Roberta-base
Dataset	COLA
Lr	2e-5
Optimizer	AdamW
Scheduler	Polynomial
batch size	32

Partition strategy	Part1(client)	Part2(server)	Part3(client)
Type 1	Self attention+Dense	Fastfeedforward+ Roberta[1:11]	Classifier

2.2 Results

Rank	Compress Rate	Acc
128	1.17	85.2
32	0.291	85.6
16	0.146	85.0
12	0.109	84.3

3 Flop counter

Here I detected the macs of mobilenetv2 at residual [224,224] of the CIFAR10 task. And here are the results

3.1 Settings

Backend	MobileNetV2
Dataset	CIFAR10
batch size	64
image size	[3,224,224]
Macs	20.1Gflops

Partition strategy	Part1(client)	Part2(server)	Part3(client)	Compress rate
Type 1	Conv+bn	relu+Residual_blocks[0:]	Classifier	1.0
Type 2	Conv+bn+conv	t_conv+relu+Residual_blocks[0:]+conv	t_conv+Classifier	0.070
Type 3	Conv+bn+conv1+conv2	t_conv1+t_conv2+relu+Residual_blocks[0:]+conv3+conv4	t_conv3+t_conv4+Classifier	0.038

3.2 Results

Partition strategy	Macs Part1	Macs Part2	Macs Part3
Type 1	7.45e8	1.93e10	8.19e7
Type 2	1.25e9	2.53e10	1.53e9
Type 3	1.60e9	3.40e10	1.65e9

4 Pipeline Paralism in two separate machines

4.1 Settings

In here I use 40CPUs to simulate the client and one GTX1080 to simulate the server

Backend	MobileNetV2
Dataset	CIFAR10
Batch size	64
Image size	[3,224,224]
Lr	0.005
Weight decay	0
Optimizer	SGD with momentum
Momentum	0.1
Partition	Client: Conv+bn， Classifier Server: The rest
Chunk	8

4.2 Results

Compress Method	Compress Rate	Acc	Bandwidth	Computation time	Total Time per batch
None	1.0	95.94	737.41MB/s	0.30s	2.07s
Conv Insert	0.097	96.01	69.03MB/s	0.32s	0.38s
Conv Insert	0.070	95.87	47.85MB/s	0.32s	0.37s
Quantization 8bits	0.25	-	184.35MB/s	0.30s	0.62s