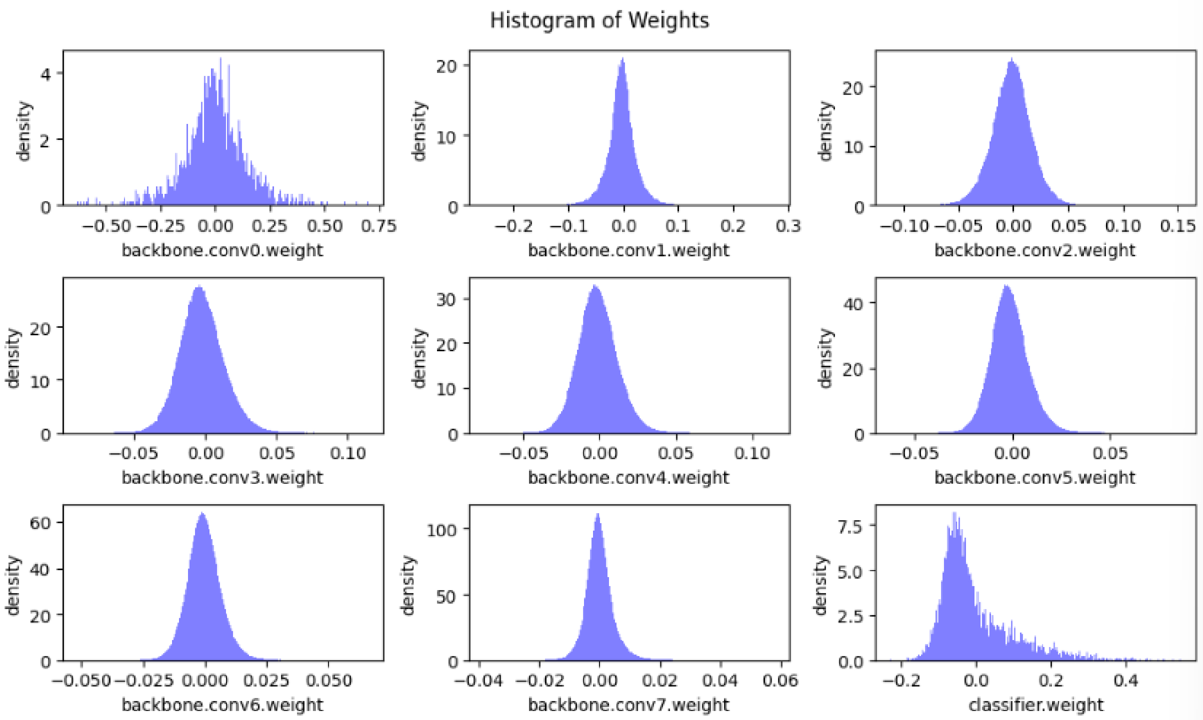
lab1

Here we have loaded a pretrained VGG model for classifying images in CIFAR10 dataset.

Let's first evaluate the accuracy and model size of this model.

```
dense model has accuracy=92.95%
dense model has size=35.20 MiB
```

Before we jump into pruning, let’s see the distribution of weight values in the dense model.



1-1 What are the common characteristics of the weight distribution in the different layers?

N(0,1)을 따르는 경향을 보임. WEIGHT VALUE가 [-0.5,0.5] 사이에 분포함.

1-2 How do these characteristics help pruning?

많은 WEIGHT을 0으로 만들기 쉬움.

Magnitude-based Pruning

▼ code(question2,3)

```
def test_fine_grained_prune(test_tensor,test_mask,target
_sparsity, target_nonzeros):
    def plot_matrix(tensor, ax, title):
        ax.imshow(tensor.cpu().numpy() == 0, vmin=0, vma
x=1, cmap='tab20c')
        ax.set_title(title)
        ax.set_yticklabels([])
        ax.set_xticklabels([])
        for i in range(tensor.shape[1]):
            for j in range(tensor.shape[0]):
                text = ax.text(j, i, f'{tensor[i, j].ite
m():.2f}',
                                ha="center", va="cente
r", color="k")

    test_tensor = test_tensor.clone()
    fig, axes = plt.subplots(1,2, figsize=(6, 10))
    ax_left, ax_right = axes.ravel()
    plot_matrix(test_tensor, ax_left, 'dense tensor')

    sparsity_before_pruning = get_sparsity(test_tensor)
    mask = fine_grained_prune(test_tensor, target_sparsi
ty)
    sparsity_after_pruning = get_sparsity(test_tensor)
    sparsity_of_mask = get_sparsity(mask)

    plot_matrix(test_tensor, ax_right, 'sparse tensor')
    fig.tight_layout()
    plt.show()

    print('* Test fine_grained_prune()')
    print(f'    target sparsity: {target_sparsity:.2f}')
    print(f'        sparsity before pruning: {sparsity_b
efore_pruning:.2f}')
    print(f'        sparsity after pruning: {sparsity_af
ter_pruning:.2f}')
    print(f'        sparsity of pruning mask: {sparsity_
of_mask:.2f}')
```

```
    if target_nonzeros is None:
        if test_mask.equal(mask):
            print('* Test passed.')
        else:
            print('* Test failed.')
    else:
        if mask.count_nonzero() == target_nonzeros:
            print('* Test passed.')
        else:
            print('* Test failed.')
```

```
def fine_grained_prune(tensor: torch.Tensor, sparsity :
float) -> torch.Tensor:
    """
    magnitude-based pruning for single tensor
    :param tensor: torch.(cuda.)Tensor, weight of conv/f
c layer
    :param sparsity: float, pruning sparsity
        sparsity = #zeros / #elements = 1 - #nonzeros /
#elements
    :return:
        torch.(cuda.)Tensor, mask for zeros
    """
    sparsity = min(max(0.0, sparsity), 1.0)
    if sparsity == 1.0:
        tensor.zero_()
        return torch.zeros_like(tensor)
    elif sparsity == 0.0:
        return torch.ones_like(tensor)

    num_elements = tensor.numel()

    ##################### YOUR CODE STARTS HERE ####ㅁ###
##############
    # Step 1: calculate the #zeros (please use round())
    num_zeros = round(num_elements * sparsity)
    print("num zeros는  ",num_zeros)
```

```
    # Step 2: calculate the importance of weight
    importance = torch.abs(tensor)
    print("importance of weight tensor => ", importance)
    # Step 3: calculate the pruning threshold
    threshold = torch.kthvalue(importance.flatten(), num
_zeros)[0]
    print("threshold= ", threshold)
    #imoprtance가 threshold보다 작으면 지우기 위함.
    # Step 4: get binary mask (1 for nonzeros, 0 for zer
os)
    mask = importance > threshold
    ##################### YOUR CODE ENDS HERE ##########
#############

    # Step 5: apply mask to prune the tensor
    tensor.mul_(mask)

    return mask
```

```
class FineGrainedPruner:
    def __init__(self, model, sparsity_dict):
        self.masks = FineGrainedPruner.prune(model, spar
sity_dict)

    @torch.no_grad() #해당 함수가 실행되는 동안 gradient 연산을
막음.
    def apply(self, model):
        for name, param in model.named_parameters():
            if name in self.masks:
                param *= self.masks[name]

    @staticmethod
    @torch.no_grad()
    def prune(model, sparsity_dict):
        masks = dict()
        for name, param in model.named_parameters():
            if param.dim() > 1: # we only prune conv and
fc weights
```

```
                masks[name] = fine_grained_prune(param,
sparsity_dict[name])
        return masks
```

```
##################### YOUR CODE STARTS HERE ############
#########
target_sparsity = (15/25) # please modify the value of t
arget_sparsity
#non-zero가 10개여야하니까
##################### YOUR CODE ENDS HERE ##############
#######
test_fine_grained_prune(target_sparsity=target_sparsity,
target_nonzeros=10)
```

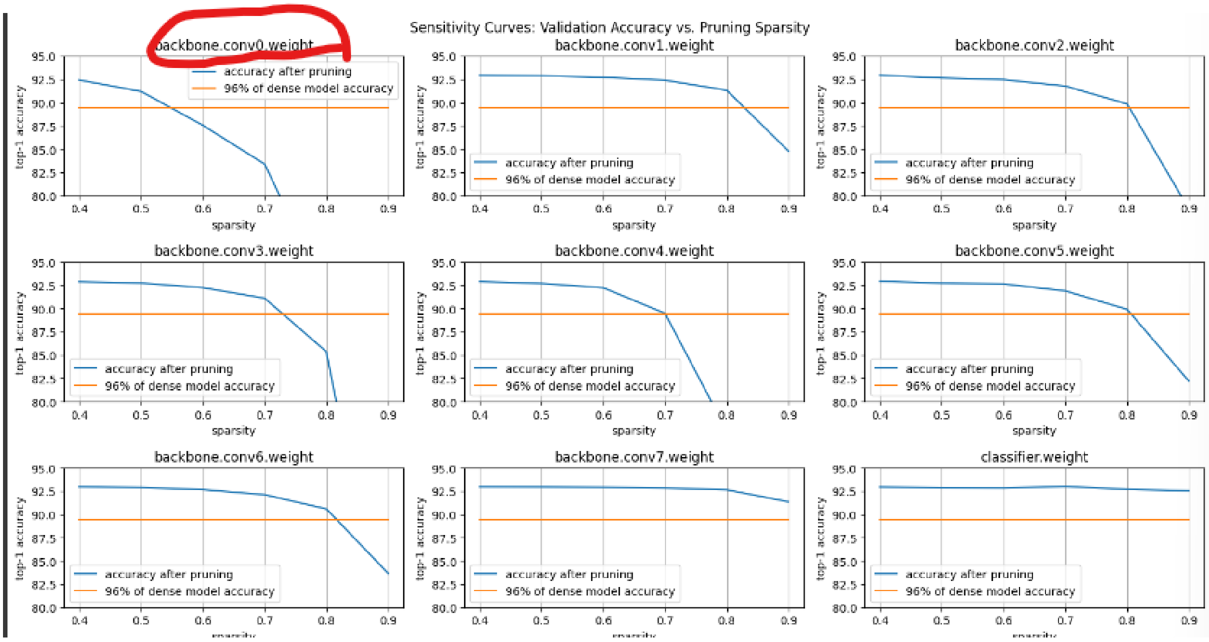
4-1 What's the relationship between pruning sparsity and model accuracy? (*i.e.*, does accuracy increase or decrease when sparsity becomes higher?)

sparsity가 증가할수록 acc의 감소가 커진다.

4-2 Do all the layers have the same sensitivity?

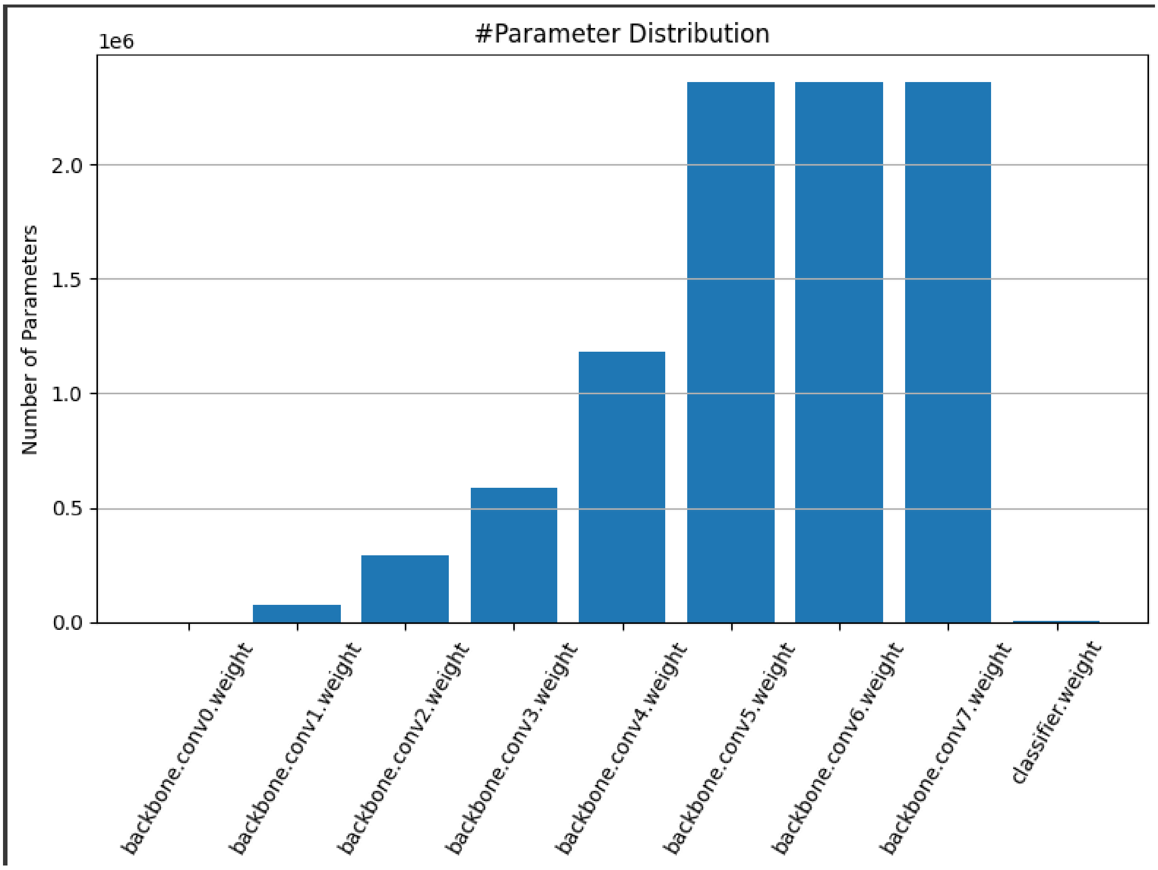
아니다.

4-3 Which layer is the most sensitive to the pruning sparsity?



▼ code(question 5,6)

더 많은 파라미터를 가질수록 sparse해서 pruning할 weight가 크다.



위 그래프에서 보이듯 layer별 민감도에 따라 sparsity가 증가하면 더 급격하게 acc감소가 발생할 것이다.

```
recover_model()

sparsity_dict = {
##################### YOUR CODE STARTS HERE ############
#########
    # please modify the sparsity value of each layer
    # please DO NOT modify the key of sparsity_dict
    'backbone.conv0.weight': 0,
    'backbone.conv1.weight': 0,
    'backbone.conv2.weight': 0,
    'backbone.conv3.weight': 0.8,
    'backbone.conv4.weight': 0.7,
    'backbone.conv5.weight': 0.6,
```

```
    'backbone.conv6.weight': 0.6,
    'backbone.conv7.weight': 0.6,
    'classifier.weight': 0
##################### YOUR CODE ENDS HERE ##############
#########
}
```

Please make sure that after pruning, the sparse model is 25% of the size of the dense model, and validation accuracy is higher than 92.5 after finetuning.

92.5 after finetuning 보다 높게 나와야함.

```
Sparse model has size=14.05 MiB = 39.92% of dense model size
Sparse model has accuracy=92.88% after fintuning
```

Channel pruning removes an entire channel, so that it can achieve inference speed up on existing hardware like GPUs.

we can remove the weights entirely from the tensor in channel pruning.

$$\#\text{out\_channels}_{\text{new}} = \#\text{out\_channels}_{\text{origin}} \cdot (1 - \text{sparsity})$$

6 유지되는 channel 숫자는 1에서 prune ratio를 뺀 값에 곱하면 됨.

```
int(round((1-prune_ratio)*channels))
```

naive하게 30%의 channel을 prune하면 92.5 → 28.14가 됨.

▼ code(question 7)

importance를 측정해서 덜 중요한 channel을 없애는 것이 중요함.

channel 별 importance를 측정하기 위해 Frobenius norm을 활용하여 weight tensor slice의 norm을 계산함.

```
# function to sort the channels from important to non-im
portant
def get_input_channel_importance(weight):
    in_channels = weight.shape[1]
```

```
    importances = []
    # compute the importance for each input channel
    for i_c in range(weight.shape[1]):
        channel_weight = weight.detach()[:, i_c]
        ##################### YOUR CODE STARTS HERE ####
#################
        importance = torch.norm(channel_weight, p='fro')
        ##################### YOUR CODE ENDS HERE ######
###############
        importances.append(importance.view(1))
    return torch.cat(importances)
```

computed importance에 따라 reordering을 진행하여 정렬함.

```
@torch.no_grad()
def apply_channel_sorting(model):
    model = copy.deepcopy(model)  # do not modify the or
iginal model
    # fetch all the conv and bn layers from the backbone
    all_convs = [m for m in model.backbone if isinstance
(m, nn.Conv2d)]
    all_bns = [m for m in model.backbone if isinstance
(m, nn.BatchNorm2d)]
    # iterate through conv layers
    for i_conv in range(len(all_convs) - 1):
        # each channel sorting index, we need to apply i
t to:
        # - the output dimension of the previous conv
        # - the previous BN layer
        # - the input dimension of the next conv (we com
pute importance here)
        prev_conv = all_convs[i_conv]
        prev_bn = all_bns[i_conv]
        next_conv = all_convs[i_conv + 1]
        # note that we always compute the importance acc
ording to input channels
        importance = get_input_channel_importance(next_c
onv.weight)
```

```
        # sorting from large to small
        sort_idx = torch.argsort(importance, descending=
True)

        # apply to previous conv and its following bn
        prev_conv.weight.copy_(torch.index_select(
            prev_conv.weight.detach(), 0, sort_idx))
        for tensor_name in ['weight', 'bias', 'running_m
ean', 'running_var']:
            tensor_to_apply = getattr(prev_bn, tensor_na
me)
            tensor_to_apply.copy_(
                torch.index_select(tensor_to_apply.detac
h(), 0, sort_idx)
            )

        # apply to the next conv input (hint: one line o
f code)
        ##################### YOUR CODE STARTS HERE ####
#################
        next_conv.weight.copy_(torch.index_select(next_c
onv.weight.detach(), 1, sort_idx))
        ##################### YOUR CODE ENDS HERE ######
###############

    return model
```

```
 * Without sorting pruned model has accuracy=28.14%
 * With sorting pruned model has accuracy=36.81% -> 향상되
긴 했지만 여전히 저조함.

 fine-tuning을 시키면 92.28%로 성능이 복구되긴함.
```

	Original	Pruned	Reduction Ratio
Latency (ms)	25.3	14.2	1.8
MACs (M)	606	305	2.0
Param (M)	9.23	5.01	1.8

8.1 Explain why removing 30% of channels roughly leads to 50% computation reduction.

(1 - 30%)^2 = 0.49

8.2 Explain why the latency reduction ratio is slightly smaller than computation reduction.

We should consider data movement as well.

9.1 Advantages and Disadvantages of Fine-Grained Pruning and Channel Pruning

- Fine-Grained Pruning:
    - Pros
        - High Compression Ratio
        - Flexibility
        - Compatibility with Dense Layers
    - Cons
        - Complexity in Hardware Support
        - Latency Issues
        - Implementation Complexity
- Channel Pruning
    - Pros
        - Improved Latency
        - Better Hardware Support
        - Structural Simplicity
    - Cons
        - Lower Compression Ratio
        - Potential Accuracy Loss
        - Limited Flexibility

9.2 On-device에서 어떤 pruning method? why?

[channel pruning], **1. Latency Improvement** (reduces the computational workload, leading to smaller, more regular matrix operations) **2. Hardware Compatibility** (channel pruning maintains the dense structure of the model) **3. Simplicity in Deployment** (remain structurally similar to the original)