감정 추론 딥러닝 모델을 사용한 AI면접용 표정 관리 임베디드 기기

윤수완*

(*) 중앙대학교 AI 학과, suwanive@proton.me

최근 여러 기업들에서 AI면접을 활용한 면접을 진행하고있다. 공기업 뿐만 아니라 사기업에서도 AI면접을 도입해가는 추세이고 이를 이용해 일차적으로 합불 여부를 가르고 있다. 다만 이러한 AI면접에서 평가 기준은 모호하고 베타적인 특성을 가지고 있다. 다만 알려진 평가 기준은 얼굴 표정을 사용하는 것이다. 얼굴 표정을 사용해 면접자가 긴장한 표정을 짓거나, 당황한 표정을 짓는다면 감점 요인이 될 수 있다. 이를 위해서는 자신의 모습을 실시간으로 파악해야하지만, 캠을 주시해야하는 AI면접 특성상 자신의 모습을 실시간으로 확인하기 어렵다. 본 연구에서는 이러한 어려움을 해결하기 위해 임베디드 디바이스를 사용하여 AI면접용 표정 관리 기술을 연구한다. 이를 위해 Nvidia Jetson Nano를 사용하였으며, 임베디드 디바이스의 부족한 연산 능력을 위해 여러 경량화 기법을 연구하였다. 또한 딥러닝 모델은 Kaggle에 있는 모델에 변형을 더한 모델이다.

본 연구에서 제시하는 과정은 Opencv를 이용한 얼굴 검출 후 딥러닝 모델을 이용한 5개의 카테고리로 감정 분류, 그리고 나서 3개의 카테고리로 분류한 후, 적절하지 않은 표정이 검출되었을 때 사용자가 카메라를 보지 않고 인지할 수 있도록 소리를 낸다. 이때 5개의 감정은 fear, happy, neutral, sad, surprise로 분류되고 각각 bad, good, acceptable, bad, acceptable로 Mapping된다. 모델 평가를 위해 5개로 분류하는 task에 대해서 최적화를 수행했다.

자세한 내용을 설명하자면 Jetson nano 환경에서 haarcascade_frontalface_default.xml을 통해 얼굴을 추출해내는데 성공했으며 검출된 image를 42*42로 변형한다. 그리고 이를 모델에 입력하여 5개의 label중 하나를 추출하고 이를 다시 3개의 label으로 맵핑한다. 만약 bad라면 사용자가 직접 녹음한 자신의 목소리를 들을 수 있다. 즉 custom이 가능하며 3개의 label mapping또한 사용자의 선호대로 mapping이 가능하다.

제시하는 모델은 다음과 같다.

단계	레이어 이름	구성 요소	출력 크기 (예시 입력 크기: 3 x 48 x 48)
1	Quantization	QuantStub()	3 x 48 x 48
2	Conv1	Conv2d(3, 32, kernel_size=3, padding=1)	32 x 48 x 48
3	ReLU	F.relu()	32 x 48 x 48
4	Pool1	MaxPool2d(2, 2)	32 x 24 x 24

단계	레이어 이름	구성 요소	출력 크기 (예시 입력 크기: 3 x 48 x 48)
5	Conv2	Conv2d(32, 64, kernel_size=3, padding=1)	64 x 24 x 24
6	ReLU	F.relu()	64 x 24 x 24
7	Pool2	MaxPool2d(2, 2)	64 x 12 x 12
8	Conv3	Conv2d(64, 128, kernel_size=3, padding=1)	128 x 12 x 12
9	ReLU	F.relu()	128 x 12 x 12
10	Dropout1	Dropout(0.5)	128 x 12 x 12
11	Pool3	MaxPool2d(2, 2)	128 x 6 x 6
12	Conv4	Conv2d(128, 128, kernel_size=1, padding=0)	128 x 6 x 6
13	ReLU	F.relu()	128 x 6 x 6
14	Dropout2	Dropout(0.5)	128 x 6 x 6
15	Flatten	Flatten()	4608
16	FC1	Linear(128 * 6 * 6, 512)	512
17	ReLU	F.relu()	512
18	Dropout3	Dropout(0.5)	512
19	FC2	Linear(512, 256)	256
20	ReLU	F.relu()	256
21	Dropout4	Dropout(0.5)	256
22	FC3	Linear(256, num_classes)	num_classes (기본값: 5)
23	Dequantization	DeQuantStub()	num_classes (기본값: 5)

```
EmotionRecognitionModel(
  2.6 M, 100.000% Params, 26.57 MMac, 99.018% MACs,
  (quant): QuantStub(0, 0.000% Params, 0.0 Mac, 0.000% MACs, )
  (dequant): DeQuantStub(0, 0.000% Params, 0.0 Mac, 0.000% MACs, )
  (conv1): Conv2d(896, 0.034% Params, 2.06 MMac, 7.694% MACs, 3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(18.5 k, 0.711% Params, 10.65 MMac, 39.704% MACs, 32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(73.86 k, 2.838% Params, 10.64 MMac, 39.636% MACs, 64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(16.51 k, 0.635% Params, 594.43 KMac, 2.215% MACs, 128, 128, kernel_size=(1, 1), stride=(1, 1))
  (pool): MaxPool2d(0, 0.000% Params, 129.02 KMac, 0.481% MACs, kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (flatten): Flatten(0, 0.000% Params, 0.0 Mac, 0.000% MACs, start_dim=1, end_dim=-1)
  (fc1): Linear(2.36 M, 90.686% Params, 2.36 MMac, 8.795% MACs, in_features=4608, out_features=512, bias=True)
  (fc2): Linear(131.33 k, 5.047% Params, 131.33 KMac, 0.489% MACs, in_features=512, out_features=256, bias=True)
  (fc3): Linear(1.28 k, 0.049% Params, 1.28 KMac, 0.005% MACs, in_features=256, out_features=5, bias=True)
  (dropout): Dropout(0, 0.000% Params, 0.0 Mac, 0.000% MACs, p=0.5, inplace=False)
)
```

모델에 대한 자세한 코드 구현은 appendix에서 확인할 수 있다. 총 파라미터 개수는 2,602,181 개이며 이를 이용해 표정 관리 sequence를 jetson nano에서 구현한 결과는 첨부한 영상에서 확인이 가능하다. 또한 학습 setting은 **Fer dataset**의 변형을 사용했으며 나머지 하이퍼파라미터 부분은 appendix에서 확인 가능하다. Fer dataset을 사용한 이유는 다른 dataset에 비해서 noise가 많아 실제로 사용하기 좋은 dataset이기 때문이다.

이 모델을 jetson nano에서 구동한 결과가 연산량이 많아 실시간 사용에 부족함이 보였기 때문에 몇가지 경량화 기법을 사용했다. 먼저 pruning을 사용했다.

baseline

0.6752293577981652

Prediction Runtime: 0.20 seconds (mps)

Prediction Runtime: 2.87 seconds (mac cpu)

10.41 MB

==========

0.6752293577981652

Prediction Runtime: 19.43 seconds (jetson gpu)

Prediction Runtime: 60.25 seconds (jetson cpu)

2602181

MACs: 26.569221 M

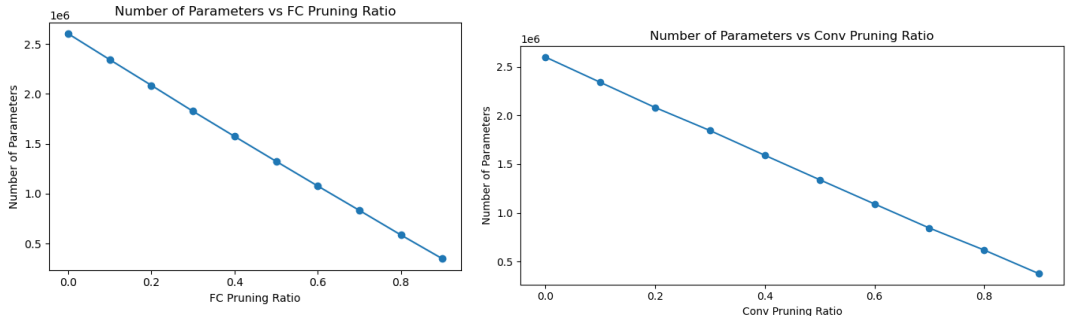
Params: 2.602181 M

10.41 MB

Pruning

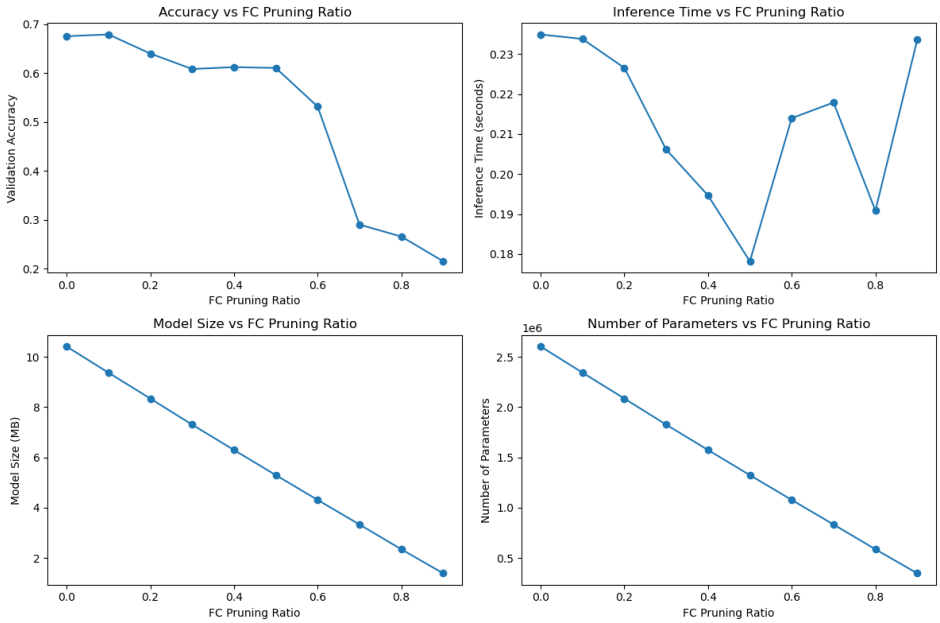
프루닝 기법을 사용한 모델은 정확도와 연산량(연산 시간)의 trade off를 보이는 경향이 있다. 사용한 프루닝 코드는 아래와 같은데, 기존의 구조적 프루닝 기법에 cnn에서는 channel 단위로, Fc layer에서는 뉴런 단위로 프루닝을 했다. 하지만 기존 pytorch의 프루닝 기법은 속도 향상을 기대하기 어려우므로 새로운 프루닝 방식을 하나 추가해서 경량화를 진행하였다.

제시하는 프루닝 기법은 dependency pruning이다. 어떠한 가중치를 제거할 때, 이와 연관된 가중치도 제거하는 방식이다.

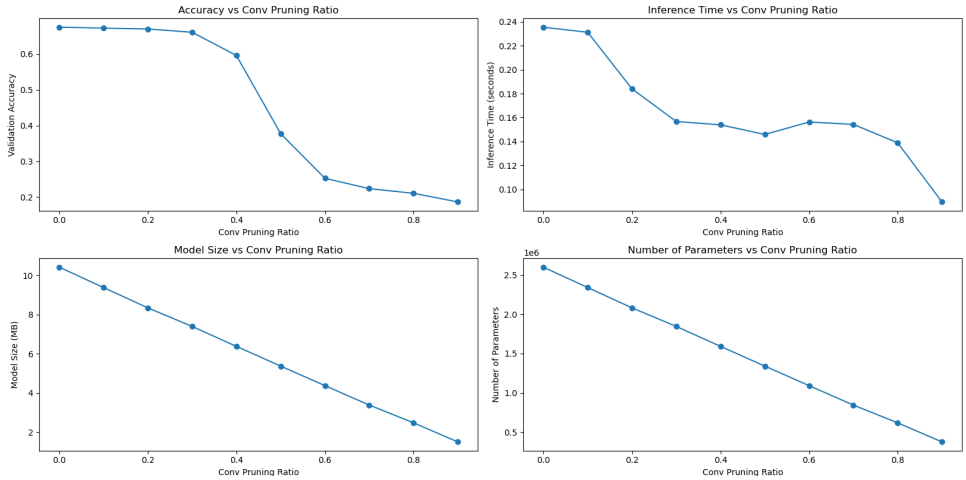


이를 이용해 FC layer만 pruning 해도 Conv layer가 dependency를 갖고있기 때문에 Conv layer도 prune되고 Conv도 역으로 성립한다. 이를 보여주는 것이 위의 figure이다. FC, CONV각각만 보고 prune한 결과가 결국 선형적으로 모델 파라미터가 0으로 향하는 결과를 볼 수 있다.

또한 prune 결과도 상당히 인상적이었다.

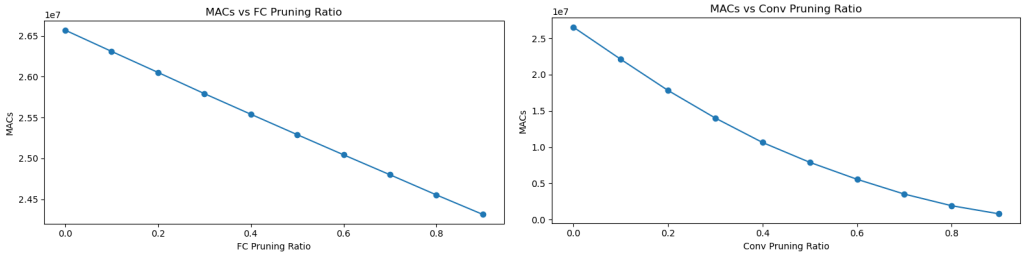


FC레이어를 기준으로 prune된 모델은 비율 0.5 이후에 급격하게 감소하는 경향을 보였다. 그리고 모델 크기와, 파라미터의 개수도 상당히 감소하였으며, inference time 또한 심한 발열으로 인한 쓰로틀링을 감안하더라도 0.5 이전까지는 감소하는 경향을 보였다.



CONV 레이어를 기준으로 prune된 모델은 0.4 이후 급격하게 감소하는 경향이 보였으며 inference time은 쓰로틀링으로 인한 연산 속도 저하가 발생했지만 적절하게 감소하는 경향을 보였다. 만약 적절한 발열 관리가 되었다면 accuracy과 동일한 개형을 보였을 수 있음이 기대된다.

(FC와 CONV 둘 다 유사한 시점에서 쓰로틀링이 발생하였다.)



또한 Flops 연산량을 보았을 때에도 적절하게 감소하는 모양이 보이는데 이때 Conv기준으로 연산량을 보았을 때, 아래로 약간 볼록한 경향성이 보였다. 또한 동시에 prune한 결과도 첨부된 코드에서 확인할 수 있으며 Pruning이 된 model의 구동 영상도 첨부된 영상에서 확인 가능하다(0.8, 0.8으로 Prune). Jetson nano에서 구동 속도 차이에 대한 비교도 첨부된 코드에서 가능하나, 쓰로틀링과 과열로 인해 적절한 비교가 불가능 하였다. 이론적으로 FLOPS연산량이 감소했기 때문에 발열이 해결되었다면 충분한 inference 속도 감소를 기대할 수 있으며 camera와 연결하여 하는 Task에서는 cpu사용량이 현저히 감소하는 경향을 확인할 수 있다. 이는 영상에서 확인 가능하다.

Quantization

이 task에서 가장 많이 시도한 방법이다. 일단 가장 먼저 quantization aware training을 하였을 때에 성능이 하락하였으며 해당 방법은 jetson nano에서 구동할 방법이 없어 포기하였으며 quantization 또한 시도하지 않았다.

quantization aware training (no quan)

0.6308590492076731

Prediction Runtime: 3.83 seconds -> mac cpu

10.45 MB

Static Quantization

Pytorch에서 기본적으로 제공하는 프루닝 기법으로 **conv, linear**에 모두 적용하였다. 이때 qint8로 quantization되었으며 모델 용량이 10.41mb에서 2.61mb로 감소하는 결과를 보였다. 다만 정확도가 상당히 낮아졌으며 Runtime 또한 약간 감소하는 경향이 보였으나 크게는 감소하는 경향이 보이지 않았다. 또한 이 방법이 jetson nano에서 구동이 불가능 하므로 더이상의 분석은 진행하지 않았다.

0.3474562135112594

Prediction Runtime: 2.24 seconds

2.61 MB

(static) – mac cpu

Dynamic Quantization (jetson cuda에서 연산은 float16, float32만 가능함)

Dynamic Quantization은 두가지 방법으로 진행하였다. 첫번째로는 torch에서 제공하는 모듈을 사용했으며 Convolution layer는 지원하지 않으므로 **fc layer**만 양자화 되었다. 따라서 모델 크기가 2.94mb로 static quantization보다 약간 큰 모습을 보였다. 다만 정확도는 baseline 모델에 비해 결과를 보였다. 다만 inference time은 감소하는 경향이 보이지 않았으며 이 또한 jetson nano에서 구동이 불가능했다. 이 또한 qint8로 양자화 되었기 때문에 jetson nano에서는 불가능했다. 또한 float16 양자화는 지원되지 않았다. 또한 파라미터 개수를 줄이는 방법이 아니기 때문에 파라미터의 수가 감소하지는 않았으나 학습 가능한 파라미터가 감소하는 것이 보였다.

0.6763969974979149

Prediction Runtime: 2.84 seconds (Mac cpu)

109,760 total parameters.

2.94 MB (dynamic qint8)

Float16

추가적으로 Quantization이라고 하기 어렵지만, Float32 를 Float16으로 바꾸는 방법을 사용해보았다. Dynamic quantization을 기반으로 **Fc layer**만 변환을 하는 코드를 작성하였으며 float16 형변환을 진행했다.

0.613511259382819

Prediction Runtime: 123.69 seconds

109,760 total parameters.

MACs: 24.0768 M

Params: 0.10976 M

Jetson gpu

5.17mb

결과값은 위와 같다. 확실히 모델 용량이 절반으로 계산과 비슷하게 감소했음을 볼 수 있으나 cpu에서 구동이 불가능하나, cuda를 명확하게 지원하지 않아 연산시간이 매우 길어 실제로 사용하기 어렵다는 판단이 들었다. 또한 실제 사용에서도 두두둑 끊기면서 예측되는 문제가 발생하였다. 이 실험를 진행한 이유를 조금 더 자세히 설명하자면, float16으로 변환하는 것이 Quantization보다 성능이 낮음을 보이기 위함이다.

Symmetric Quantization

그리고 최종적으로 **Conv, Linear 모두** Symmetric하게 Quantization하는 코드를 찾아내어 작성하였다. 출처는 코드에 남겨두었다. 다만 이 또한 CPU에서 구동되었다. 코드는

하단에 첨부한다. 그리고 아래는 구동 결과이다.

Accuracy (int4, int8, int16, int32)
[0.6445371142618849, 0.6748957464553795, 0.6752293577981652, 0.6752293577981652]

Inference time (int4, int8, int16, int32) (mac cpu)
[0.225108163356781, 0.2165154528617859, 0.252600314617157, 0.33066543340682986]

Model size (int4, int8, int16, int32)
[2.4834861755371094, 2.4834861755371094, 4.964076995849609, 9.92519760131836]

눈에 띄는 점은 model size 가 눈에 띄게 줄어드는 것이며 Accuray 도 눈에 띄게 증가했다. 또한 inference time 또한 점차 감소하는 경향을 보였다. 실제 jetson nano 환경에서도 baseline 에 비해 cpu 점유율이 상당히 낮았으며 prune 과 같을 정도로 점유율이 낮아졌다. Int4 에 대한 시연 결과는 영상에서 찾아볼 수 있으며 확실히 baseline 보다는 성능이 좋지 않음을 보였다만 prune 보다 월등히 좋은 성능을 보였다.

결론적으로 목표로 했던 모든 task에 성공했으며 실제 사용에도 적합할 정도로 연산속도를 향상시키는데 성공했다.
다만 conv prun에서 FLOPS연산이 곡선으로 보이는 모습과 FC와 CONV의 프루닝 비율에 따른 정확도 하강 그래프에 대해서 연구해볼 필요가 있으며 dataset을 변경하는 방법도 고려해볼 수 있다. 예를 들어 CK+와 같은 dataset은 정확도가 높게 나오는 잘 정제된 dataset이므로 이를 사용한다면 Pruning기법의 효과를 더 잘 보여줄 수 있을 것으로 보이지만, CK+는 FER에 비해 잘 정제되어있는 데이터로 실생활에 사용하기 어렵다는 단점이 있어서 이 두 데이터셋을 섞는 방식도 고려해 볼 필요가 있다.

Appendix

Model

```python
from torch.quantization import QuantStub, DeQuantStub

class EmotionRecognitionModel(nn.Module):
    def __init__(self, num_classes=5):
        super(EmotionRecognitionModel, self).__init__()
        self.quant = QuantStub()  # Quantization module
        self.dequant = DeQuantStub()  # Dequantization module

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=1, padding=0)
        self.pool = nn.MaxPool2d(2, 2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 6 * 6, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.quant(x)  # Quantize input
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.dropout(x)
        x = self.pool(x)
        x = F.relu(self.conv4(x))
        x = self.dropout(x)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        x = self.dequant(x)  # Dequantize output
        return x
model = EmotionRecognitionModel().to('mps')
```

Pruning code

```
# %%
import warnings
warnings.filterwarnings('ignore')  # Ignore warnings

import torch_pruning as tp
import time
from torch.quantization import QuantStub, DeQuantStub

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
import os
import numpy as np
import cv2
from ptflops import get_model_complexity_info


# %%
class EmotionRecognitionModel(nn.Module):
    def __init__(self, num_classes=5):
        super(EmotionRecognitionModel, self).__init__()
        self.quant = QuantStub()
        self.dequant = DeQuantStub()

        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=1, padding=0)
        self.pool = nn.MaxPool2d(2, 2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 6 * 6, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.quant(x)
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.dropout(x)
        x = self.pool(x)
        x = F.relu(self.conv4(x))
        x = self.dropout(x)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        x = self.dequant(x)
        return x


# %%
```

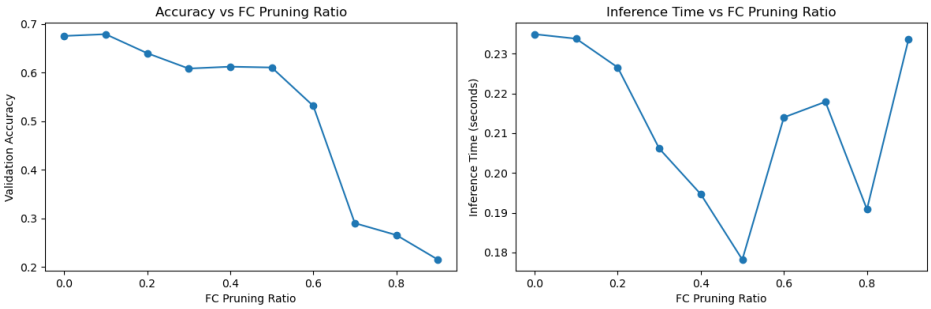
```
model = EmotionRecognitionModel()
model.load_state_dict(torch.load('MyModelFaceRecogD3.pth',
map_location=torch.device('cuda')))
# %%
val_path = 'validation/'
print(os.listdir(val_path))
# Loading validation data
x_val = []
y_val = []
# dataset load
for class_idx, class_name in enumerate(['happy', 'sad', 'fear', 'surprise',
'neutral']):
    class_folder = os.path.join(val_path, class_name)
    for img_file in os.listdir(class_folder):
        img = cv2.imread(os.path.join(class_folder, img_file))
        img = cv2.resize(img, (48, 48))
        x_val.append(img)
        y_val.append(class_idx)

x_val = np.array(x_val)
y_val = np.array(y_val)

y_val_tensor = torch.tensor(y_val, dtype=torch.long)
x_val_tensor = torch.tensor(x_val, dtype=torch.float32).permute(0, 3, 1, 2)
/ 255.0

val_dataset = TensorDataset(x_val_tensor, y_val_tensor)
val_loader = DataLoader(val_dataset, batch_size=100, shuffle=False)
# %%

example_inputs = torch.randn(1, 3, 48, 48)

# %%
# pruning 적용 함수
import torch.nn.utils.prune as prune
#prun rate define
conv_prune = 0.8
fc_prune = 0.8
model = EmotionRecognitionModel()
model.load_state_dict(torch.load('MyModelFaceRecogD3.pth',
map_location=torch.device('cuda')))

#use torch.nn.utils.prun, find where to prune
def apply_pruning(model, conv_prune=conv_prune, fc_prune=fc_prune):
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            prune.ln_structured(module, name='weight', amount=conv_prune,
n=2, dim=0)  # 필터의 20%를 pruning
        elif isinstance(module, nn.Linear):
            prune.ln_structured(module, name='weight', amount=fc_prune, n=2,
dim=0)  # FC 레이어의 뉴런 40%를 pruning


# pruning 완료 후 가중치에서 마스크 제거
def finalize_pruning(model):
    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, nn.Linear)):
            prune.remove(module, 'weight')
```

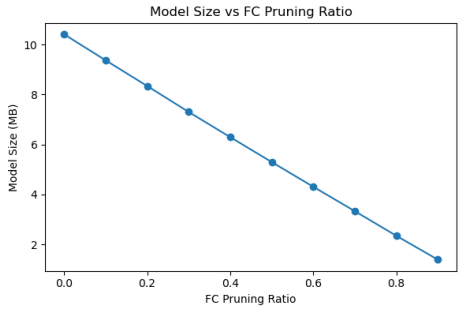
```
apply_pruning(model)
finalize_pruning(model)


# 가중치 값이 0인 인덱스 찾는 함수
def find_zero_weight_indices(model):
    zero_indices = {}
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            weight = module.weight.data
            # 출력 채널별로 가중치의 절대값 합 계산
            weight_abs_sum = weight.abs().view(weight.shape[0], -
1).sum(dim=1)
            # 합이 0인 경우 (완전히 pruning된 필터)
            zero_channels = (weight_abs_sum ==
0).nonzero(as_tuple=False).squeeze().tolist()
            zero_indices[name] = zero_channels
        elif isinstance(module, nn.Linear):
            weight = module.weight.data
            # 각 뉴런의 입력 가중치 절대값 합 계산
            weight_abs_sum = weight.abs().sum(dim=1)
            # 합이 0인 경우 (완전히 pruning된 뉴런)
            zero_neurons = (weight_abs_sum ==
0).nonzero(as_tuple=False).squeeze().tolist()
            zero_indices[name] = zero_neurons
    return zero_indices


zero_weight_indices = find_zero_weight_indices(model)
# %%
import copy

model = EmotionRecognitionModel()
model.load_state_dict(torch.load('MyModelFaceRecogD3.pth',
map_location=torch.device('cuda')))
net = copy.deepcopy(model)
net.to('cuda')
example_inputs = torch.randn(1,3, 48, 48)
example_inputs.to('cuda')
#do prune with dependency
DG = tp.DependencyGraph().build_dependency(net.to('cuda'),
example_inputs=example_inputs.to('cuda'))

pruning_idxs = zero_weight_indices['conv1']
pruning_group = DG.get_pruning_group(net.conv1, tp.prune_conv_out_channels,
idxs=pruning_idxs)

if DG.check_pruning_group(pruning_group):
    pruning_group.prune()

pruning_idxs = zero_weight_indices['conv2']
pruning_group = DG.get_pruning_group(net.conv2, tp.prune_conv_out_channels,
idxs=pruning_idxs)

if DG.check_pruning_group(pruning_group):
    pruning_group.prune()

pruning_idxs = zero_weight_indices['conv3']
```

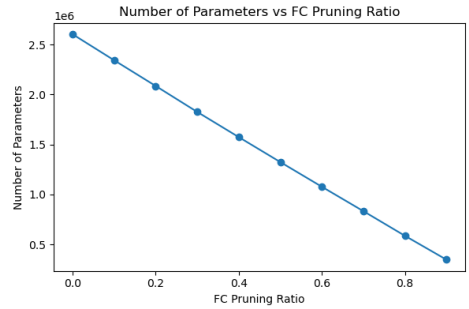
```
pruning_group = DG.get_pruning_group(net.conv3, tp.prune_conv_out_channels,
idxs=pruning_idxs)

if DG.check_pruning_group(pruning_group):
    pruning_group.prune()

pruning_idxs = zero_weight_indices['conv4']
pruning_group = DG.get_pruning_group(net.conv4, tp.prune_conv_out_channels,
idxs=pruning_idxs)

if DG.check_pruning_group(pruning_group):
    pruning_group.prune()

pruning_idxs = zero_weight_indices['fc1']
pruning_group = DG.get_pruning_group(net.fc1, tp.prune_linear_out_channels,
idxs=pruning_idxs)

pruning_idxs = zero_weight_indices['fc2']
pruning_group = DG.get_pruning_group(net.fc2, tp.prune_linear_out_channels,
idxs=pruning_idxs)

if DG.check_pruning_group(pruning_group):
    pruning_group.prune()

# 3. prune all grouped layer that is coupled with model.conv1
if DG.check_pruning_group(pruning_group):
    pruning_group.prune()

base_macs, base_nparams = tp.utils.count_ops_and_params(model.to('cuda'),
example_inputs.to('cuda'))
print('======================')
print('Before')
print(f'\tMACs: {base_macs / 1e6} M')
print(f'\tParams: {base_nparams / 1e6} M')
prune_macs, prune_nparams = tp.utils.count_ops_and_params(net.to('cuda'),
example_inputs.to('cuda'))
print('After')
print(f'\tMACs: {prune_macs / 1e6} M')
print(f'\tParams: {prune_nparams / 1e6} M')
print('======================')


# %%
def print_model_size(mdl):
    torch.save(mdl.state_dict(), "tmp.pt")
    print("%.2f MB" % (os.path.getsize("tmp.pt") / 1e6))
    os.remove('tmp.pt')


# %%
correct_val = 0
total_val = 0

start_time = time.time()

net.eval()
#do test
with torch.no_grad():
    for inputs, labels in val_loader:
        outputs = net(inputs.to('cuda'))
```

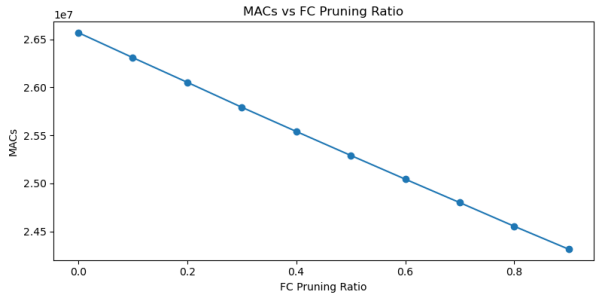
```
        _, predicted_val = torch.max(outputs, 1)
        total_val += labels.size(0)
        correct_val += (predicted_val.to('cuda') ==
labels.to('cuda')).sum().item()

    val_acc = correct_val / total_val  # Validation accuracy for this epoch
    end_time = time.time()
    print(val_acc)

    runtime = end_time - start_time
    print(f'Prediction Runtime: {runtime:.2f} seconds')
    total_params = sum(p.numel() for p in net.parameters())
    print(total_params)
    prune_macs, prune_nparams =
tp.utils.count_ops_and_params(net.to('cuda'), example_inputs.to('cuda'))
    print(f'\tMACs: {prune_macs / 1e6} M')
    print(f'\tParams: {prune_nparams / 1e6} M')
    print_model_size(model)
    flops, params = get_model_complexity_info(net, (3,48,48),
as_strings=True, print_per_layer_stat=True)
    print("flops : ", flops)
```

Pruning (fc) (쓰로틀링 발생 -> 그럼에도 충분히 속도 증가가 보임)

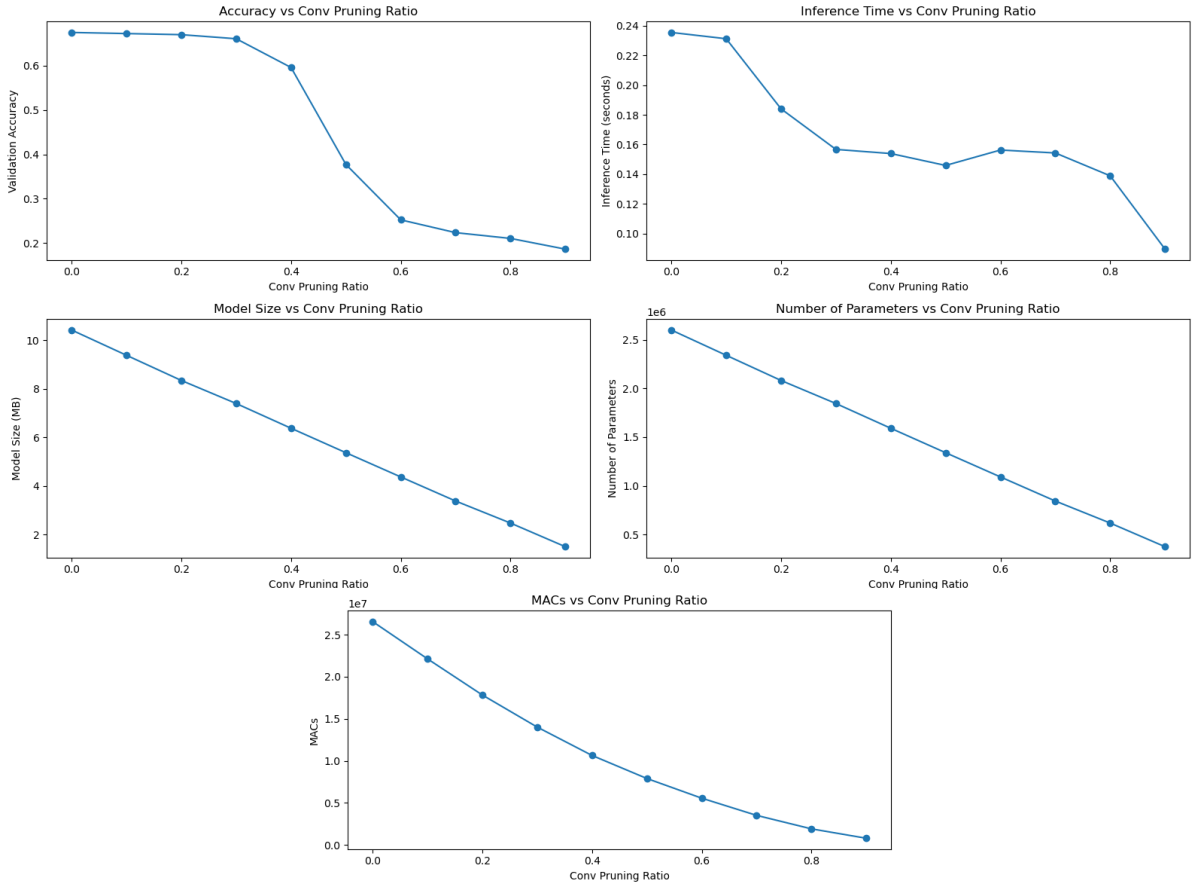








Pruning (conv) -> 쓰로틀링 발생



Quantization(static)

```
model.to('cpu')
backend = "qnnpack"
model.qconfig = torch.quantization.get_default_qconfig(backend)
torch.backends.quantized.engine = backend
model_static_quantized = torch.quantization.prepare(model, inplace=False)
model_static_quantized = torch.quantization.convert(model_static_quantized, inplace=False)

import os
import torch

def print_model_size(mdl):
    torch.save(mdl.state_dict(), "tmp.pt")
    print("%.2f MB" %(os.path.getsize("tmp.pt")/1e6))
    os.remove('tmp.pt')
print_model_size(model_static_quantized)
print_model_size(model)
```

Executed at 2024.12.03 21:48:46 in 139ms

```
2.61 MB
10.41 MB
```

0.3474562135112594

Prediction Runtime: 2.24 seconds

2.61 MB

(static) - cpu (conv)

==========

baseline

0.6752293577981652

Prediction Runtime: 0.20 seconds (mps)

Prediction Runtime: 2.87 seconds (cpu)

10.41 MB

==========

quantization aware training

0.6308590492076731

Prediction Runtime: 3.83 seconds -> cpu에서 사용

10.45 MB

-> Quantization 인지 학습이 성능이 잘 나오지 않음.

==========

코드 설명

엔진을 설정하고, prepare후 quanization. Conv도 프루닝 됨.

Quantization (qint8, dynamic)

```
import torch
import os
import time

# Dynamic quantization
model_dynamic_quantized = torch.quantization.quantize_dynamic(
    model,  # Pass the pre-trained model here
    {torch.nn.Linear},  # Specify layers to quantize (e.g., Linear layers)
    dtype=torch.qint8  # Use 8-bit integers for weights
)


# Function to calculate model size
def print_model_size(mdl):
    torch.save(mdl.state_dict(), "tmp.pt")
    print("%.2f MB" % (os.path.getsize("tmp.pt") / 1e6))
    os.remove('tmp.pt')

# Validation Accuracy and Runtime
model_dynamic_quantized.to('cpu')
model_dynamic_quantized.eval()

correct_val = 0
total_val = 0

start_time = time.time()

with torch.no_grad():
    for inputs, labels in val_loader:
        outputs = model_dynamic_quantized(inputs.to('cpu'))
        _, predicted_val = torch.max(outputs, 1)
        total_val += labels.size(0)
        correct_val += (predicted_val.to('cpu') ==
labels.to('cpu')).sum().item()

    val_acc = correct_val / total_val  # Validation accuracy for this epoch
    end_time = time.time()
    print(val_acc)

    runtime = end_time - start_time
    print(f'Prediction Runtime: {runtime:.2f} seconds')
    total_params = sum(p.numel() for p in
model_dynamic_quantized.parameters())
    print(f'{total_params:,} total parameters.')
    print_model_size(model_dynamic_quantized)
```

0.6763969974979149

Prediction Runtime: 2.84 seconds (cpu), does not support mps

109,760 total parameters.

2.94 MB

형변환 (f16 dynamic)

```python
class FP16Linear(nn.Module):
    def __init__(self, linear_layer):
        super(FP16Linear, self).__init__()
        assert isinstance(linear_layer, nn.Linear), "Only nn.Linear is supported"
        self.in_features = linear_layer.in_features
        self.out_features = linear_layer.out_features
        self.bias = linear_layer.bias is not None

        # Convert weights and bias to float16
        self.weight = linear_layer.weight.detach().clone().to(torch.float16)
        self.bias_data = linear_layer.bias.detach().clone().to(torch.float16) if self.bias else None

    def forward(self, x):
        # Move weights and biases to the same device as the input
        weight = self.weight.to(x.device)
        bias = self.bias_data.to(x.device) if self.bias else None

        # Ensure input is also in FP16
        x = x.to(torch.float16)

        # Perform the linear transformation
        output = torch.matmul(x, weight.t())
        if bias is not None:
            output += bias
        return output


def apply_dynamic_fp16_quantization(model, target_layers):
    for name, module in model.named_children():
        # Replace target layers
        if type(module) in target_layers:
            if isinstance(module, nn.Linear):
                setattr(model, name, FP16Linear(module))
        else:
            # Recursively apply to submodules
            apply_dynamic_fp16_quantization(module, target_layers)
    return model
```

코드 설명,

10,11번 줄에서 torch.float16으로 변환하고 layer재구성

0.613511259382819
Prediction Runtime: 1.54 seconds

109,760 total parameters.

MACs: 24.0768 M

Params: 0.10976 M

(dynamic, float16) -mps (linear)

==========

0.613511259382819
Prediction Runtime: 2.39 seconds

109,760 total parameters.

MACs: 24.0768 M

Params: 0.10976 M

(dynamic, float16) -Mac cpu (linear)

==========

0.613511259382819
Prediction Runtime: 123.69 seconds
109,760 total parameters.
MACs: 24.0768 M
Params: 0.10976 M
Jetson gpu

Final Quantization

```
class WeightQuantizer:
    def __init__(self, num_bits):
        self.num_bits = num_bits
        if num_bits == 32:
            self.qmin, self.qmax = float('-inf'), float('inf')
        else:
            self.qmin, self.qmax = -(2 ** (num_bits - 1)), (2 ** (num_bits -
1)) - 1
        self.scale = None
        self.zero_point = 0  # Symmetric quantization uses 0

    def calibrate(self, tensor):
        """Compute quantization parameters based on the given tensor."""
        if self.num_bits == 32:
            self.scale = 1.0
            return
        max_val = tensor.max().item()
        min_val = tensor.min().item()
        max_abs = max(abs(max_val), abs(min_val))
        self.scale = max_abs / (self.qmax)
        self.scale = max(self.scale, 1e-8)  # Prevent zero-scale

    def quantize(self, tensor):
        """Quantize the given tensor using computed parameters."""
        if self.scale is None:
            self.calibrate(tensor)

        if self.num_bits == 32:
            return tensor

        quantized = torch.round(tensor / self.scale).clamp(self.qmin,
self.qmax)
        dtype_map = {4: torch.int8, 8: torch.int8, 16: torch.int16, 32:
torch.float32}
        return quantized.to(dtype_map[self.num_bits])

    def dequantize(self, quantized_tensor):
        """Dequantize the tensor back to its original range."""
        if self.num_bits == 32:
            return quantized_tensor
        return quantized_tensor.float() * self.scale


class QuantizedLayer:
    def __init__(self, weight_tensor, num_bits):
        self.quantizer = WeightQuantizer(num_bits)
        self.quantized_weight = self.quantizer.quantize(weight_tensor)

    def get_dequantized_weight(self):
        """Retrieve the dequantized weight."""
        return self.quantizer.dequantize(self.quantized_weight)


class QuantizedModel(nn.Module):
    def __init__(self, original_model, num_bits):
        super().__init__()
        self.original_model = original_model
        self.num_bits = num_bits
```

```
        self.quantized_layers = {}

        self._quantize_model()

    def _quantize_model(self):
        """Apply quantization to all layers with weights."""
        for name, module in self.original_model.named_modules():
            if isinstance(module, (nn.Conv2d, nn.Linear)):
                quantized_layer = QuantizedLayer(module.weight.data.clone(),
self.num_bits)
                self.quantized_layers[name] = quantized_layer

module.weight.data.copy_(quantized_layer.get_dequantized_weight())

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

    def estimated_size(self):
        """Estimate the memory size of the quantized model."""
        param_count = sum(
            module.weight.numel()
            for module in self.original_model.modules()
            if isinstance(module, (nn.Conv2d, nn.Linear))
        )
        param_size = 1 if self.num_bits <= 8 else 2
        return param_count * param_size + 1024

    def save_model(self, filepath):
        """Save the quantized model and parameters."""
        save_data = {
            'num_bits': self.num_bits,
            'state_dict': {},
            'quantization_params': {}
        }

        for name, qlayer in self.quantized_layers.items():
            save_data['state_dict'][name] = qlayer.quantized_weight
            save_data['quantization_params'][name] = {
                'scale': qlayer.quantizer.scale,
                'zero_point': qlayer.quantizer.zero_point
            }

        torch.save(save_data, filepath)
```

Learning setup

```
# Model parameters
num_classes = 5
input_shape = (48, 48, 3)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=0.0009, weight_decay=0.01)

train_dataset = TensorDataset(images_tensor, labels_tensor)
train_loader = DataLoader(train_dataset, batch_size=100, shuffle=True)
val_dataset = TensorDataset(x_val_tensor, y_val_tensor)
val_loader = DataLoader(val_dataset, batch_size=100, shuffle=False)

epochs = 200
patience = 25
best_val_acc = 0
```