Assignment 4: CNN for Food Image Classification

Overview

이번 과제는 `pytorch` 를 이용해 음식 사진 classification model을 만드는 과제입니다. `Dataset`, `Module` class 를 이해하고 training loop을 직접 작성해주세요. code의 틀은 제공되니 명세서의 조건에 맞춰 구현해주세요! **과제 전반에 걸쳐 pdb를 적극적으로 이용해주세요**

Objective

1. Custom Dataset class 의 __getitem__()을 구현한다.
2. Custom CNN model의 __forward__()를 구현한다.
3. Training loop을 작성한다.
4. CLI 에서 python 파일을 실행시킨다.
5. model의 구조를 수정하고, pretrained 모델을 불러와 성능을 비교한다.

Environment

이번 과제는 GPU가 사용 가능한 환경에서 실행해주시기 바랍니다. 가용한 서버나 클라우드 리소스(Vessl 등)가 있는 분은 활용해 주세요. GPU 사용이 어려우신 분들은 github에 코드를 올리고 코랩에서 실행해주시기 바랍니다.

우선 프로젝트를 시작할 폴더에 repo를 clone해줍니다.

```
git clone https://github.com/gdvstd/week04-TrainCNN.git
```

1. GPU가 사용 가능한 환경이 있는 경우

   프로젝트에 사용할 가상환경을 파줍니다.

   ```
   conda create -n week04_assignment python=3.11
   conda activate week04_assignment
   ```

requirements를 다운받아줍니다.

```
cd ./week04-TrainCNN/week-04-Pytorch1
pip install -r requirements.txt
```

이제 아래와 같이 main.py를 실행시킬 수 있습니다.

```
python main.py -m 'CNN1' -e 50 -b 32 -lr 1e-4
```

nvidia GPU를 사용하는 경우 아래 명령어로 GPU 사용 현황을 모니터 할 수 있습니다.

```
nvidia-smi
```

CUDA_VISIBLE_DEVICES 환경변수를 설정하면 특정 GPU에 올려 실험할 수도 있습니다.

```
# 1번 GPU (2번째)를 사용해 main.py를 실행시킴
CUDA_VISIBLE_DEVICES=1 python main.py -m 'CNN1' -e 50 -b 32
```

mac 내장 GPU(mps)를 사용하는 경우 activity monitor > View > GPU Processes 로 GPU 프로세스를 모니터하고 activity monitor > Window > GPU History 로 가용 GPU 메모리를 모니터할 수 있습니다.

nohup을 이용하면 background에서 process를 실행시킬 수 있습니다. 이 경우 컴퓨터 화면을 닫아도 프로세스가 멈추지 않습니다. (아래 명령어는 main.py를 background에서 실행시키고 output을 log.txt로 redirect합니다.)

```
nohup python main.py -m 'CNN1' -e 50 -b 32 -lr 1e-4 > log.txt
```

2. Colab에서 실행하는 경우

   로컬에서 coding하고 본인 github repo에 프로젝트를 올려줍니다. **Notebook 환경에서는 디버깅 하기가 매우 까다로우므로 로컬 CPU에서 batch_size를 작게 지정해 1epoch이 돌아가는 것을 확인 한 후 Colab에서 실험할 것을 강력히 권장합니다.**

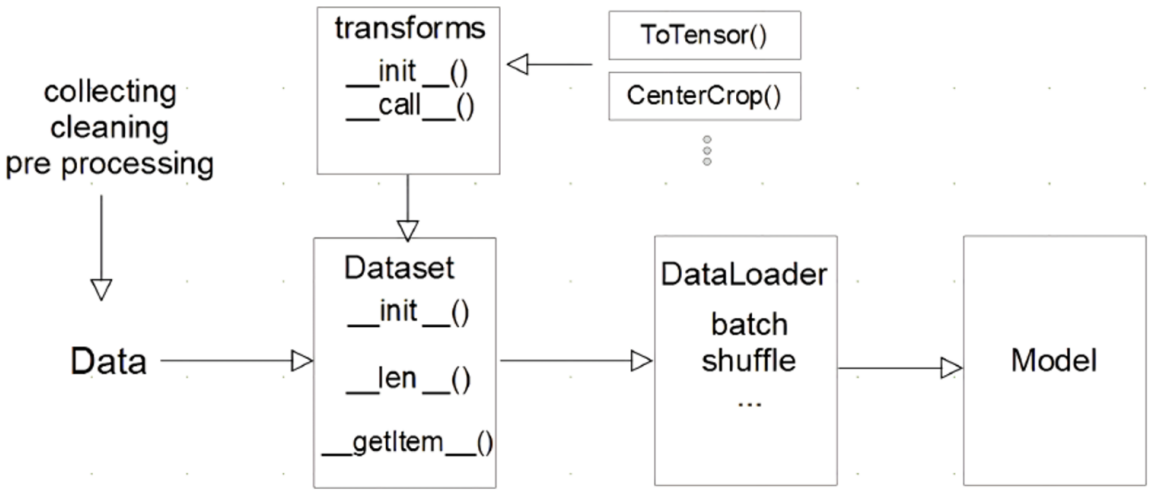
   프로젝트 github에 올리는 방법은 다음 튜토리얼을 참고해주세요. (https://soda-dev.tistory.com/12)

Colab에서 github repo를 clone 해 main.py를 실행시키는 방법은 week04.ipynb 파일을 참고해주세요.

Code Explanation

1. `main.py` : model의 직접적인 training이 수행되는 파일입니다. `dataset.py`에서 정의한 dataset class를 가져오고 `model.py`에서 정의한 model 객체를 가져와 사용합니다. 이후 실험시 직접 실행시킬 파일이므로 argument parsing도 `main.py`에서 진행해줍니다.
2. `dataset.py` : dataset class가 정의된 파일입니다. 현재 custom dataset인 `FoodDataset`의 생성자가 정의되어있습니다.
3. `model.py`: 사용할 model class가 정의된 파일입니다. 현재 직접 구현해서 사용할 `vanillaCNN`와 `vanillaCNN2`의 생성자가 정의되어있고, `torchvision`에서 pretrained model인 VGG19를 가져와 classifier head를 교체해 사용하는 `VGG19`가 정의되어 있습니다.

What is Dataset, Dataloader?



Dataset과 Dataloader는 파이토치에서 지원하는 data handling class 입니다. dataset이 data를 담아두고 하나씩 가져오는 객체라면, dataloader는 dataset에서 N개의 data를 가져와서 묶어 제공해주는 객체라고 볼 수 있습니다. 우리는 우리가 가진 data의 형식에 맞춰 dataset class를 정의하고 dataloader에 넘겨줘 data를 batch 단위로 받아올 수 있습니다.

그럼 우리만의 dataset class은 어떻게 정의할 수 있을까요? Dataset class를 상속해서 __init__, __len__, __getitem__ 메소드를 구현하면 됩니다. 이는 각각 생성자, 데이터셋의 길

이를 반환하는 함수, data 1쌍을 반환하는 함수로 동작합니다.

이제 정의한 dataset 객체를 dataloader에게 전달하면 batch단위로 data를 가져오는 iterable이 생성됩니다.

```
train_dataset = FoodDataset("./data", "train", transforms=transforms)
train_loader = DataLoader(train_dataset, batch_size=args.batch, shuffle=True)
val_dataset = FoodDataset("./data", "val", transforms=transforms)
val_loader = DataLoader(val_dataset, batch_size=args.batch, shuffle=True)
```

이때 batch는 무엇이고 iterable은 무엇일까요? 우리는 모델을 학습시킬 때 dataset 전체를 모델에 때려넣고 학습(parameter update)시키지 않습니다. 그런 학습은 매우 비효율적이고 느리기 때문입니다. 대신 dataset의 일부를 선택해 모델에 통과시키고 학습(parameter update), 다른 일부를 통과시키고 학습(parameter update) .. 하는 과정을 반복합니다. 이때 모델이 보는 dataset의 일부를 batch라고 합니다. (참고: SGD)

iterable이란 for loop으로 순회할 수 있는 객체를 의미합니다. 예를들어 우리가 아는 python list, set, dict, np.array등이 모두 iterable입니다.

즉 딥러닝의 관점에서 batch란 모델이 한번의 파라미터 update를 겪을 때 보는 dataset의 일부입니다. 그리고 batch 단위로 data를 가져올 수 있게 해주는 iterable이 dataloader객체입니다. dataloader는 대표적으로 dataset 객체, batch_size, shuffle이라는 argument를 받아 생성됩니다.

dataloader는 한번 data를 제공할 때 batch_size 번 __getitem__()을 call 해 batch_size 개의 data sample을 가져오고, 이를 batch로 묶어 제공합니다. 이때 shuffle이 True라면 0번째부터 순서대로 batch_size개의 data를 뽑지 않고 random하게 batch_size의 data를 뽑아옵니다.

```
data_list = [1,2,3,4,5]
for data in data_list:
        # 어떤 처리
```

위 반복문이 data_list의 모든 요소를 순회하면 종료되듯이

```
from torch.utils.data import DataLoader

data_loader = DataLoader(dataset, batch_size=5, shuffle=True)
```

```
for data in data_loader:
        # 모델 학습 ( 파라미터 update )
```

위 반복문도 dataset의 모든 요소를 순회하면 종료됩니다. 예를 들어 batch_size가 5고 len(dataset)이 30이라면 6번의 반복 즉 6번의 parameter update가, 70이라면 14번의 반복 즉 14번의 parameter update가 일어나는 것입니다.

이렇게 dataset을 1번 순회하며 학습 하면 모델이 1 epoch 학습했다고 표현합니다. 5 epoch 학습한다는 것은 그러면 위 반복문을 5번 실행한다는 말이 되겠죠.

1. Dataset.py

`pytorch` Dataset class를 상속받아 우리가 학습에 사용할 `FoodDataset`이 정의되어 있습니다. 생성자에서 `self.prepare_dataset()`을 불러 (image 경로, class index) pair list를 self.data에 저장합니다. `__len__()` method는 self.data의 길이를 반환합니다.

이미지 데이터를 처리할 때 관습적으로 dataset의 생성자는 data root 와 transforms를 받습니다. 이미지 데이터는 용량이 커 생성자에서 모두 로딩하면 CPU memory의 부담이 매우 커집니다. 따라서 생성자에서 (image path, label)를 준비해두고 `__getitem__()`이 불려질 때 이미지를 가져와 transform(또는 augmentation)을 적용하고 return하는 것이 보편적입니다.

TODO : __getitem__() 메소드 구현

getitem은 index를 parameter로 받아 index에 해당하는 data pair(input - target)를 반환해주는 메소드입니다. 생성자에서 self.data에 image path와 class를 받아놓았으므로 다음 처리를 해주어야 합니다.

1. self.data를 인덱스로 참조해 image path와 class index를 받아온다.
2. image path에 해당하는 image를 받아온다.

   `PIL.Image.open` 을 사용할 수 있습니다.
3. transform을 적용한다.

   transform이란 이미지 데이터 증강을 의미합니다. 대부분의 transform은 callable한 object로 제공됩니다. 다른 말로 아래와 같은 간단한 형태로 transform을 적용할 수 있습니다.

   ```
   # transform은 transform 객체, image는 image 객체
   ```

```
augmented_image = transform(image)
```

이번 과제에서는 `torchvision.transforms`에서 제공하는 transform을 사용합니다. Compose는 transform 객체를 순서대로 거치도록 묶는 class 입니다. 아래 transforms은 이미지를 (227, 227) 크기로 resizing하고 수직, 수평 방향으로 0.5의 확률로 flip하는 augmentation을 정의합니다.

```
transforms = T.Compose([
    T.Resize((227,227), interpolation=T.InterpolationMode.BILINEAR),
    T.RandomVerticalFlip(0.5),
    T.RandomHorizontalFlip(0.5),
])
```

4. tensor로 변환

   self.totensor를 이용해 transform을 적용한 이미지를 tensor로 바꿀 수 있습니다.

5. dict 형태로 반환

   __getitem__()의 return 값은 dataloader에 의해 batch화 됩니다. readability를 위해 관습적으로 __getitem__()의 출력은 dict로 구성합니다. 그러면 이후 batch에서도 같은 field로 접근할 수 있습니다.

```
# dataset class의 getitem 선언
def __getitem__(self, idx):
    # 이런저런 처리
    return {
       'input': image, # shape : (3, 227, 227)
       'target': label # shape : (1, )
    }

# training loop에서 같은 field로 배치화된 data를 얻을 수 있
for batch, data in enumerate(dataloader):
    input = data['input'] # shape : (N, 3, 227, 227)
    target = data['target'] # shape : (N, )
```

2. Model.py

`pytorch` `nn.Module` class 를 상속받아 `vanillaCNN` 과 `VGG19`가 정의되어 있습니다. `vanillaCNN` 은 (N, 3, 227, 227) shape의 RGB 이미지를 받아 (N, 20) shape의 logit tensor를 만드는 단순한 CNN 모델입니다. (N은 배치 크기를, logit이란 softmax를 적용했을 때 class 확률분포가 되는 값을 의미합니다.)

`VGG19`는 ImageNet으로 pretrain된 vggnet19을 불러옵니다. ImageNet은 1000개의 class를 가지는 데이터셋이기 때문에 pretrained vggnet은 (N, 1000)의 output shape을 갖습니다. 따라서 (N, 20)의 output shape을 가지는 classifier로 생성자에서 교체해줍니다.

TODO-1: vanillaCNN forward() 메소드 구현

`nn.Module`은 callable한 객체를 정의합니다. forward()는 객체가 call될 때 실행될 동작을 정의합니다.

```
class vanillaCNN(nn.Module):
    def __init__(self):
        super().__init__()

        self.cv1 = nn.Conv2d(in_channels=3, out_channels=96, kernel_size=11, stride=4)
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.cv2 = nn.Conv2d(in_channels=96, out_channels=256, kernel_size=5, padding=2)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.cv3 = nn.Conv2d(in_channels=256, out_channels=384, kernel_size=3, padding=1)
        self.cv4 = nn.Conv2d(in_channels=384, out_channels=384, kernel_size=3, padding=1)
        self.cv5 = nn.Conv2d(in_channels=384, out_channels=256, kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.relu = nn.ReLU()

        self.dropout = nn.Dropout()
        self.head = nn.Linear(in_features=9216, out_features=20)
```

vanillaCNN의 구성요소는 위와같이 이루어져 있습니다. forward가 input을 Parameter로 받고

cv1 → relu → pool1 → cv2 → relu → pool2 → cv3 → relu → cv4 → relu → cv5 → relu → pool3 → dropout → head

순서로 통과시켜 return 하도록 구현해주세요. 추가로 fc layer에 진입하기 전에 flatten 하는 과정이 필요합니다.

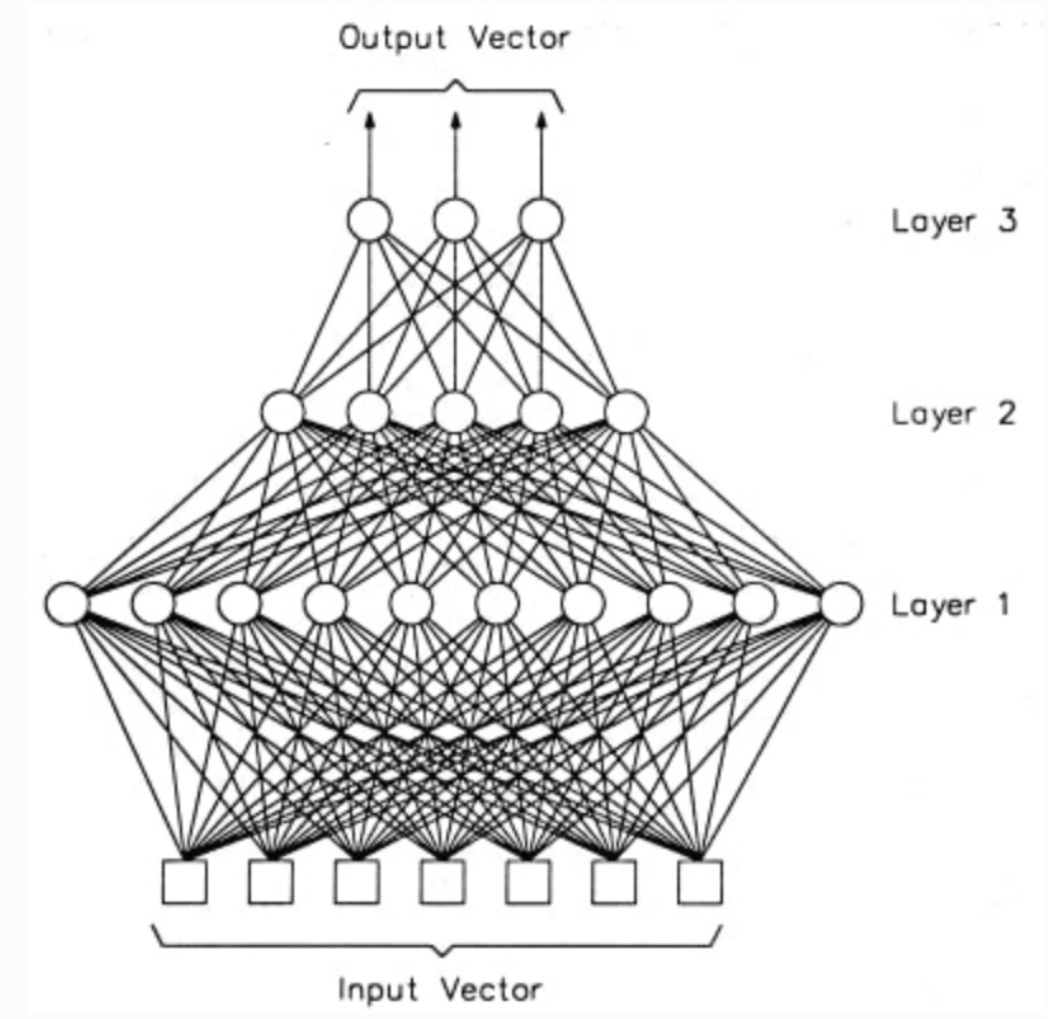
shape: (N, 256, 6, 6) → (N, 9216)

순서가 잘못되거나 flatten이 잘못되면 shape 오류가 뜹니다. 각 구성요소의 input shape, output shape같은 경우는 docs를 참조하시면 알 수 있습니다 [Conv2d docs].

TODO-2: vanillaCNN2 구현

vanillaCNN의 head는 단순한 linear projection head를 사용하기 때문에 분류성능이 좋지 않습니다. vanillaCNN의 head를 다음과 같은 구성요소를 가지는 MLP로 교체해 주세요.

nn.Linear → nn.ReLU → nn.Dropout → nn.Linear → nn.ReLU → nn.Dropout → nn.Linear



사진으로 표현하면 위와 같습니다. 이때 input vector는 convolution layer를 거치고 flatten된 (N, 9216) shape의 tensor입니다. layer1은 2048개, layer2도 2048개, layer3는 20개의 node를 갖도록 구성해주세요.

`nn.Sequential` 을 사용하면 여러 모듈을 순서대로 묶을 수 있습니다. (VGG19 구현 참고)

(self.head를 잘 정의하면 forward는 vanillaCNN과 똑같이 사용할 수 있습니다.)

3. main.py

`main.py` 에서는 크게 7 개의 과정이 일어납니다.

1. 실험 조건을 argument로 받음
2. dataset, dataloader를 정의
3. model을 정의
4. optimizer와 loss function을 정의
5. model을 지정된 device에 올림
6. Train 및 성능 logging
7. model 저장

위 과정 중 1, 2, 3번은 제공되어 있습니다.

TODO: Training loop 작성하기

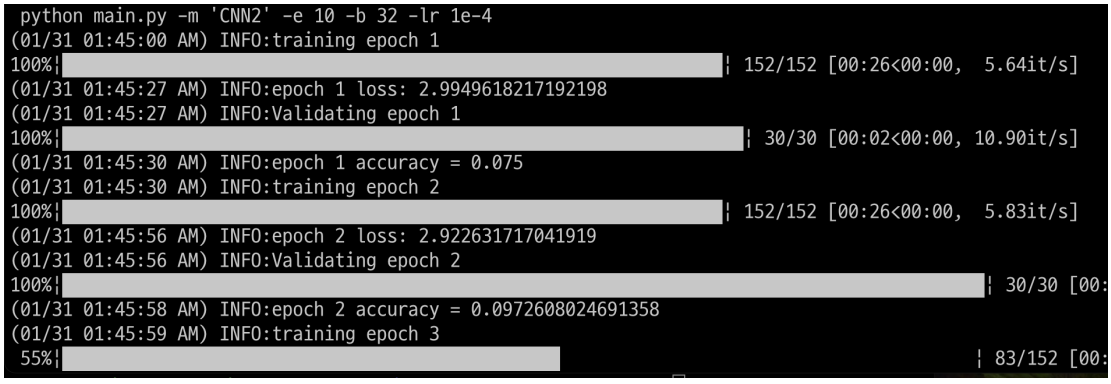
optimizer로는 Adam optimizer, loss function으로는 CrossEntropyLoss를 사용해주세요.

model train 과정에서는 다음과 같은 처리가 이루어져야 합니다.

1. epoch 마다 train_loader를 이용해 train set을 순회하며 parameter를 update 한다.
    a. progress bar가 보여야 합니다.
        i. hint: tqdm을 이용해 순회하면 progress bar를 띄워줍니다.
    b. model이 training mode로 전환되어야 합니다.
        i. 전환되지 않으면 dropout layer가 잘못 적용됩니다.
    c. step마다 step loss가 logging 되어야 합니다. (logging level : DEBUG)
        i. log message format : “Step [step index] loss : [loss값]“
        ii. hint : enumerate를 사용해 순회하면 step index를 쉽게 얻을 수 있습니다.

2. epoch 마다 평균 Loss를 logging한다. (logging level : INFO)
    a. log message format : “Epoch [epoch] loss : [평균 loss 값]”
3. epoch마다 val_loader를 이용해 validation set을 순회하면 validation accuracy를 측정한다.
    a. progress bar가 보여야 합니다.
        i. hint: tqdm을 이용해 순회하면 progress bar를 띄워줍니다.
    b. model이 evaluation mode로 전환되어야 합니다.
        i. 전환되지 않으면 dropout layer가 잘못 적용됩니다.
    c. torch.no_grad()를 이용해 autograd engine의 gradient 추적을 막아주세요.
4. epoch 마다 validation score를 logging한다. (logging level : INFO)
    a. log message format : “Epoch [epoch] accuracy = [평균 validation accuracy]”
5. epoch마다 model checkpoint를 저장한다.
    a. save dir : ./save/[args.model]_[args.epoch]_[args.batch]_[args.learning_rate]/
    b. file name: [epoch]_score:[소수점 3자리 까지 rounding한 accuracy].pth

Log가 잘 찍히면 아래와 같이 나타납니다. (default logging level = INFO)



4. Let’s Train

모든 구현을 마쳤다면 아래 세팅에서 모델을 학습시키고 간단한 보고서를 작성해주세요. 보고서는 결과 보고 형식으로 1페이지 이내로 작성하여 (week04_[이름].pdf) github에 함께 제출해주시면 됩니다.

< 실험 세팅 >

1. model : vanillaCNN(CNN1), epoch : 50, batch_size : 32, lr : 1e-4
2. model : vanillaCNN2(CNN2), epoch : 50, batch_size : 32, lr : 1e-4
3. model : VGG19(VGG), epoch : 50, batch_size : 32, lr : 1e-4

< 보고서에 포함되어야 하는 내용 >

1. 모델 별로 가장 잘 나온 validation score는 얼마이고, 성능이 가장 좋았던 epoch은 언제인지 간단한 표로 정리해주세요.
2. 어떤 모델의 성능이 가장 좋았고 어떤 모델이 가장 안 좋았나요? 왜 그렇다고 생각하나요?
3. Pytorch를 써야 하는 이유에 대해 간단히 서술해주세요 (2~3 문장이면 충분합니다.)

Submission Format

아래 형식으로 본인 github에 upload해주세요.

```
week04-TrainCNN
|-- README.md
|-- week-04-Pytorch1
        |-- requirements.txt
        |-- class_info.json
        |-- dataset.py
        |-- model.py
        |-- main.py
        |-- week04.ipynb
        |-- week04_남세현.pdf
        |-- data
                |-- train
```

```
                    |-- val
                    |-- test
```