DNN_01_03. MLP_Pytorch

생성일	@2022년 6월 13일 오후 6:00
유형	머신러닝/딥러닝
작성자	동훈 오

구글 드라이브 세팅은 텐서플로우와 동일하다.

```
from google.drive import drive
drive.mount("/content/drive")

import os
import sys
sys.path.append("/content/drive/Mydrive/source")
!pip install -r "/content/drive/Mydrive/source/requirements.txt"
```

라이브러리 세팅

```
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

import torch
from torch import nn
import torch.nn.functional as F

from torch import optim
from torch_optimizer import RAdam
from torch_optimizer import AdamP

from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import random_split

from torchvision.datasets import FashionMNIST

import wandb

from data_utils import dataset_split
```

(pytorch 는 GPU 를 활용할 CUDA 언어를 지원 한다.) 사용할 수 있는 GPU 탐색은 다음의 코드를 실행하면 된다.

```
!nvidia-smi
>>>
-------------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   73C    P8    37W / 149W |      0MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
```

data preprocessing & defining dataset

```
# dataset을 저장할 data_root 를 미리 지정한다.
# 코랩은 클라우드 서버를 사용하므로 데이터를 임시로 저장하면 된다고 생각.
data_root = os.path.join(os.getcwd(), "data")

# preprocessing & defining datset
transform = transforms.Compose(
  [
    transforms.ToTensor(),
    transforms.Normalize([0.5],[0.5]), # mean, std
  ]
)

# data download
fashion_mnist_dataset = FashionMNIST(data_root, download=True,
                                     train=True, transform=transform)
```

dataloader 정의

```
datasets = dataset_split(fashion_mnist_dataset, splilt=[0.9, 0.1]

train_dataset = datasets["train"]
val_dataset = datasets["val"]

train_batch_size = 100
val_batch_size = 10

train_dataloader = torch.utils.data.DataLoader(
    train_dataset, batch_size=train_batch_size, shuffle=True, num_workers=1
)
val_dataloader = torch.utils.data.DataLoader(
    val_dataset, batch_size=val_batch_size, shuffle=True, num_workers=1
)
```

위에서 사용된 dataset_split() 은 data_util 파일로 따로 만들어두었다.

```
def dataset_split(
  dataset: torch.utils.data.Dataset,
  split: List[float] = [0.9, 0.1],
  random_train_val_split: bool=Fasle,
) -> Dict[str, torch.utils.data.dataset.Subset]:
"""
  Returns:
    Dict[str, torch.utils.data.dataset.Subset]:
      return subset of datasets as dictionaries.
      i.e. {
              "train": train_dataset,
              "val": val_dataset,
              "test": test_datset,
            }
"""
# 방어
assert len(split) in [2,3]
assert sum(split) == 1.0
for frac in split:
  assert frac >= 0.0

indices = list(range(len(dataset)))
modes = ["train","val","test"][: len(split)]
sizes = np.array(np.cumsum([0] + list(split)) * len(dataset), dtype=int)
# sizes = [0, train_size, train_size+val_size, len(datasets)]

if random_train_val_split:
  train_and_val_idx = indices[: sizes[2]]
  random.shuffle(train_and_val_idx)
  indices = train_and_val_idx + indices[sizes[2] :]

datasets = {
```

```
          mode: torch.utils.data.Subset(
              dataset, indices[sizes[i] : sizes[i + 1]]
          )
          for i, mode in enumerate(modes)
  }

  return datasets
```

모델 정의 : MLP → MLPWithDropout

파이토치에서 서브클래싱 방식의 모델 정의는 텐서플로우와 유사하다. layer 순서를 쌓는 메서드가 텐서플로우에서는 call 이었다면, 파이토치에서는 forward 이다.

```
class MLP(nn.Module):
  def __init__(self, in_dim: int, h1_dim: int, h2_dim: int, out_dim: int):
    super().__init__()
    self.linear1 = nn.Linear(in_dim, h1_dim)
    self.linear2 = nn.Linear(h1_dim, h2_dim)
    self.linear3 = nn.Linear(h2_dim, out_dim)
    self.relu = F.relu
    pass

  def forward(self, input):
    x = torch.flatten(input, start_dim=1)
    x = self.relu(self.linear1(x))
    x = self.relu(self.linear2(x))
    out = self.linear3(x)
    return out


class MLPWithDropout(MLP):
  def __init__(self, in_dim: int, h1_dim: int, h2_dim: int, out_dim: int, dropout_prob: float):
      super().__init__( in_dim, h1_dim, h2_dim, out_dim)
      self.dropout1 = nn.Dropout(dropout_prob)
      self.dropout2 = nn.Dropout(dropout_prob)

  def forward(self, input):
      x = torch.flatten(input, start_dim=1)
      x = self.relu(self.linear1(x))
      x = self.dropout1(x)   # 활성함수를 거친 다음, dropout
      x = self.relu(self.linear2(x))
      x = self.dropout2(x)
      out = self.linear3(x)
      # out = F.softmax(out)
      return out
```

모델 선언 및 손실 함수, 최적화 정의

텐서플로우로 만든 모델과 달리, 파이토치에서는 learning rate scheduler 는 제외했다.

```
# define model
model = MLPWithDropout(28*28, 128, 64, 10, dropout_prob=0.3)
model_name = type(model).__name__

# define loss function
loss_function = nn.CrossEntropyLoss()

# define optimizer
lr=1e-3
optimizer = torch.optim.Adam(model.prameters(), lr=lr)
optimizer_name = type(optimizer).__name__

max_epoch = 15

# define tensorboard logger
run_name = f"{datetime.now().isoformat(timespec='seconds')}-{model_name}-{optimizer_name}_optim_{lr}"
log_dir = f"runs/{run_name}"
```

```
writer = SummaryWriter(log_dir=log_dir)
log_interval = 100

# define wandb
project_name = "torch_fashion_mnist_tutorials"
run_tags = [project_name]
wandb.init(
    project=project_name,
    name=run_name,
    tags=run_tags,
    config={"lr": lr, "model_name": model_name, "optimizer_name": optimizer_name},
    reinit=True,
)

# setting for saving model path
log_model_path = os.path.join(log_dir, "models")
os.makedirs(log_model_path, exist_ok=True)
```

학습진행: train step & validation step

```
%load_ext tensorboard
%tensorboard --logdir runs/

# 훈련스텝 초기화
train_step = 0

for epoch in range(1, max_epoch+1):
  # train step 보다 validation step 을 먼저 두면 디버깅할 때 편리하다.
  # validation data에서는 optimizer가 업데이트를 하지 않는다.

  # valid step
  with torch.no_grad():
    # 검증 손실, 검증 정확도 초기화
    val_loss = 0.0
    val_corrects = 0
    model.eval()

    for val_batch_idx, (val_images, val_labels) in enumerate(
      tqdm(val_dataloader, position=0, leave=True, desc="validation")
    ):
      # forward
      val_outputs = model(val_images)
      _, val_preds = torch.max(val_outputs, 1)

      # loss & accuracy
      # 누적된 loss & acc 각각 batch size 로 나누어 평균을 구한다.
      val_loss += loss_function(val_outputs, val_labels) / val_outputs.shape[0]
      val_corrects += torch.sum(val_preds == val_labels.data) / val_outputs.shape[0]

# valid step logging
val_epoch_loss = val_loss / len(val_dataloader)
val_epoch_acc = val_corrects / len(val_dataloader)

print(
   f"{epoch} epoch, {train_step} step: val_loss: {val_epoch_loss}, val_acc: {val_epoch_acc}"
)

# tensorboard log
writer.add_scalar("Loss/val", val_epoch_loss, train_step)
writer.add_scalar("Acc/val", val_epoch_acc, train_step)
writer.add_images("images/val", val_images, train_step)

# wandb log
wandb.log({
  "Loss/val": val_epoch_loss,
  "Acc/val": val_epoch_acc,
  "Images/val": wandb.Image(val_images),
  "Outputs/val": wandb.Histogram(val_outputs.detach().numpy()),
  "Preds/val": wandb.Histogram(val_preds.detach().numpy()),
  "Labels/val": wandb.Histogram(val_labels.data.detach().numpy()),
}, step=train_step)

######################################################################

# train step
current_loss = 0
current_corrects = 0
```

```
model.train()

for batch_idx, (images, labels) in enumerate(
    tqdm(train_dataloader, position=0, leave=True, desc="training")
):

    current_loss = 0
    current_corrects = 0

    # forward
    # get prediction
    outputs = model(images)
    _, preds = torch.max(outputs, 1)  # '1' 배치는 유지하고 index 출력

    # loss
    loss = loss_function(outputs, labels)

    # backpropagation
    # optimizer 초기화
    optimizer.zero_grad()

    # 최적화 수행
    optimizer.step()

    # 누적된 loss, accuracy 계산
    current_loss += loss.item()
    current_corrects += torch.sum(preds == labels.data)

  if train_step % log_interval == 0:
      train_loss = current_loss / log_interval
      train_acc = current_corrects / log_interval
      print(
        f"{train_step}: train_loss: {train_loss}, train_acc: {train_acc}"
      )

      # tensorboard log
      writer.add_scalar("Loss/train", train_loss, train_step)   #tensorboard에 추가.
      writer.add_scalar("Acc/train",train_acc, train_step)
      writer.add_images("Images/train", images, train_step)
      writer.add_graph(model, images)

      # wandb log
      wandb.log({
        "Loss/train": train_loss,
        "Acc/train": train_acc,
        "Images/train": wandb.Image(images),
        "Outputs/train": wandb.Histogram(outputs.detach().numpy()),
        "Preds/train": wandb.Histogram(preds.detach().numpy()),
        "Labels/train": wandb.Histogram(labels.data.detach().numpy()),
      }, step=train_step)

      current_loss = 0
      current_corrects = 0

  train_step += 1
```

- with torch.no_grad()

  no_grad() 감싸게 되면, 파이토치는 autograd engine 을 꺼버린다. 더 이상 자동으로 gradient 를 트래킹하지 않는다. 어차피 validation step 에서는 가중치 업데이트가 일어나지 않으므로 gradient가 계산되지 않는다. 따라서 autograd 를 끔으로써 메모리 사용량을 줄이고 연산속도를 높일 수 있다.

- model.eval()

  eval() 함수는 해당 모델의 모든 레이어가 evaluate mode 에 들어가게 해준다. 학습할 때만 필요한 dropout, batchnorm 등의 기능을 자동으로 비활성화시킨다.

- tqdm

  tqdm 은 python 진행률 프로세스바 이다.

  사용법 예시:

  ```
  from tqdm import tqdm
  import time
  ```

```
for i in tqdm(range(100, desc="tqdm example", mininterval=0.01)):
  print(i)
```

- iterable : 반복하는 객체
- desc : 진행바 앞에 텍스트 출력
- totla: int, 전체 반복량
- leave: bool, default로 True(진행상태 잔상이 남음.)
- mininterval : 업데이트 주기

model 저장

```
torch.save(model, os.path.join(log_model_path, "model.ckpt"))
```

softmax 함수

```
def softmax(x, axis=0):
  max = np.max(x, axis=axis, keepdims=True)
  e_x = np.exp(x - max)
  sum = np.sum(e_x, axis=axis, keepdims=True)
  f_x = e_x/sum
  return f_x
```

model test

모델을 테스트 하는 과정은 텐서플로우에서 진행한 것과 유사하다.

```
test_batch_size = 10
# 테스트셋을 미리 만들어두지 않아서 새롭게 추가한다.
test_dataset = FashionMNIST(
  data_root, download=True, train=False, transform=transforms.ToTensor()
)
test_dataloader = torch.utils.data.DataLoader(
  test_dataset, batch_size=test_batch_size, shuffle=False, num_workers=1
)

test_labels_list = []
test_preds_list = []
test_outputs_list = []

for i, (test_images, test_labels) in enumerate(
  tqdm(test_dataloader, position=0, leave=True, desc="testing")
):
  # 미리 훈련시킨 모델을 가져온다.
  test_outputs = loaded_model(test_images)
  _, preds = torch.max(test_outputs, 1)

  final_outs = softmax(test_outputs.detach().numpy(), axis=1)
  test_outputs_list.extend(final_outs)
  test_preds_list.extend(test_preds.detach().numpy())
  test_labels_list.extend(test_labels.detach().numpy())

test_preds_list = np.array(test_preds_list)
test_labels_list = np.array(test_labels_list)

print(f"acc: {np.mean(test_preds_list == test_labels_list)*100}")
```