


DNN_02_2. CNN_TF

🕒 생성일	@2022년 6월 10일 오후 1:49
📁 유형	머신러닝/딥러닝
👤 작성자	 동훈 오

데이터는 그대로 fashion_mnist 를 사용할 것이기에 데이터 전처리 및 dataloader 부분까지의 부분은 동일하다.

이제부터는 모델에 관여하는 파라미터 튜닝을 편리하게 하기 위해 configuraton 관리를 하기 시작할 것이다. 따라서 ‘추가 라이브러리 설치’ 부분에서 몇 개의 라이브러리가 새로 추가 되었다.

개발에 필요한 패키지들을 txt 파일로 작성해두었다. /content/drive/Mydrive/source 위치에 “requirements.txt” 파일명으로 저장.

```
pytorch-lightning==1.3.8
torch-optimizer==0.1.0
hydra-core==1.1
wandb==0.11.1
torchtext==0.10.0
spacy==2.2.4
efficientnet_pytorch==0.7.1
tensorflow-addons==0.14.0
```

위의 패키지를 설치하기 위해 구글 드라이브 접근 및 시스템 path 설정 → !pip 를 통해 패키지 설치

```
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"
```

추가 라이브러리 설치

코랩에서 제공하는 머신러닝 관련 라이브러리를 import.

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

from omegaconf import OmegaConf
from omegaconf import DictConfig # type checking

import tensorflow as tf
import tensorflow_addons as tfa
import wandb
```

하드웨어 체크 및 GPU 설정.

코랩에서는 48 시간까지 연속으로 GPU 를 사용할 수 있다. GPU 를 사용하기 위한 세팅은 다음과 같다.

```
tf.config.list_physical_devices()
# define GPUs strategy
mirrored_strategy = tf.distribute.MirroredStrategy() # gpu 병렬 처리할 수 있다.
```

data : normalization / split / dataloader

```
with mirrored_strategy.scope():
    # 사용할 데이터셋은 fashion_mnist 이다.
    fashion_mnist = tf.keras.datasets.fashion_mnist
    (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

    # 이미지 데이터 -> 255 로 나누어서 0-1 사이로 정규화
    x_train = x_train / 255.0 # float type
    x_test = x_test / 255.0

    # split 비율 미리 지정
    train_size = int(len(x_train) * 0.9)
    val_size = len(x_train) - train_size

    # dataset 을 만들기 위해 tensor 객체로 변환 후 이미지와 라벨을 함께 전달.
    dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(buffer_size=1024)
    test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test)).shuffle(buffer_size=1024)

    # dataset split 과정
    train_dataset = dataset.take(train_size)
    val_dataset = dataset.skip(train_size)

    # dataloader 정의

    train_batch_size = 100
    val_batch_size = 10
    test_batch_size = 100

    train_dataloader = train_dataset.batch(train_batch_size, drop_remainder = True)
    val_dataloader = val_dataset.batch(val_batch_size, drop_remainder = True)
    test_dataloader = test_dataset.batch(test_batch_size, drop_remainder = True)
```

CNN모델 : configuration

생성할 CNN 모델은 (합성곱 층 + 풀링층)1 + (합성곱 층 + 풀링층)2 + dense layer 3개로 구성될 계획이다. configuration 부분을 미리 작성한다.

```
_cnn_cfg: dict = {
    "layer_1": {
        "conv2d_filters" : 32,
        "conv2d_kernel_size" : [3, 3],
        "conv2d_strides" : [1, 1],
        "conv2d_padding" : "same",
        "maxpool2d_pool_size" : [2, 2],
        "maxpool2d_strides" : [2, 2],
        "maxpool2d_padding" : "valid",
    },
    "layer_2": {
        "conv2d_filters" : 64,
        "conv2d_kernel_size" : [3, 3],
        "conv2d_strides" : [1, 1],
        "conv2d_padding" : "valid",
        "maxpool2d_pool_size" : [2, 2],
        "maxpool2d_strides" : [1, 1],
        "maxpool2d_padding" : "valid",
    },
    "fc_1" : {"units" : 512},
    "fc_2" : {"units" : 128},
    "fc_3" : {"units" : 10},
    "dropout_prob" : 0.25,
}
```

```
# OmegaConf 모듈로 파일을 읽어온다. yaml, json 형식을 파일을 지원한다.
_cnn_cfg = OmegaConf.create(_cnn_cfg)
```

CNN 모델 : Convolutional layer + Batch Normalization + MaxPooling lyaer.

앞서 만들어둔 configuration을 이용해, 합성곱 층과 배치 정규화 기능 + max pooling layer 를 call 해주는 클래스를 생성해보자. (모델 안에 들어가는 레이어 층을 따로 만들때는 Layer 클래스를 상속받는 것이 일반적이다.)

```
class ConvBatchNormMaxPool(tf.keras.layers.Layer):
    def __init__(
        self,
        conv2d_filters,
        conv2d_kernel_size,
        conv2d_strides,
        conv2d_padding,
        maxpool2d_pool_size,
        maxpool2d_strides,
        maxpool2d_padding
    ):
        super().__init__()
        # 합성곱 층 정의
        self.conv2d = tf.keras.layers.Conv2D(
            filters=conv2d_filters,
            kernel_size=conv2d_kernel_size,
            strides=conv2d_strides,
            padding=conv2d_padding,
        )
        # 배치 정규화 정의
        self.batchnorm = tf.keras.layers.BatchNormalization()
        # 풀링 층 정의
        self.maxpool2d = tf.keras.layers.MaxPool2D(
            pool_size=maxpool2d_pool_size,
            strides=maxpool2d_strides,
            padding=maxpool2d_padding
        )

    def call(self, input):
        """ Conv2D --> BatchNormalization --> Activation --> MaxPooling"""
        x = self.conv2d(input)
        x = self.batchnorm(x)
        x = tf.keras.activation.relu(x)
        out = self.maxpool2d(x)
        return out
```

CNN 모델 : 하나의 클래스에서 완성.

서브 클래싱 방식으로 모델을 커스텀 할 때, configuration managing tool 을 이용해 layer 를 쌓는 방식은 모델이 복잡해질수록 작업에 효과적이다.

지금까지 만들어둔 CNN 관련 부분을 하나의 클래스에서 결합시켜준다. 그리고 train step 과 test step 까지 하나의 클래스에서 설정해준다. train step & test step 은 이전에 만들어 둔 MLP 에서 가져와서 그대로 재사용한다.

```
class CNN(tf.keras.Model):
    def __init__(self, cfg: DictConfig = _cnn_cfg):
        super().__init__()
        self.layer1 = ConvBatchNormMaxPool(
            cfg.layer_1.conv2d_filters,
            cfg.layer_1.conv2d_kernel_size,
            cfg.layer_1.conv2d_strides,
            cfg.layer_1.conv2d_padding,
            cfg.layer_1.maxpool2d_pool_size,
            cfg.layer_1.maxpool2d_strides,
            cfg.layer_1.maxpool2d_padding
        )
```

```

)
self.layer2 = ConvBatchNormMaxPool(
    cfg.layer_2.conv2d_filters,
    cfg.layer_2.conv2d_kernel_size,
    cfg.layer_2.conv2d_strides,
    cfg.layer_2.conv2d_padding,
    cfg.layer_2.maxpool2d_pool_size,
    cfg.layer_2.maxpool2d_strides,
    cfg.layer_2.maxpool2d_padding
)

# 1. Global pooling --> dense + 바로 softmax 적용
# 2. Flatten 하는 방법 --> Dense (여기서 적용)
self.flatten = tf.keras.layers.Flatten()

self.fc1 = tf.keras.layers.Dense(cfg.fc_1.units)
self.fc2 = tf.keras.layers.Dense(cfg.fc_2.units)
self.fc3 = tf.keras.layers.Dense(cfg.fc_3.units)

self.dropout = tf.keras.layers.Dropout(cfg.dropout_prob)

def call(self, input, training=False):
    # 2차원 이미지를 3차원으로 변환시켜준다. 끝에 한 차원을 더 추가시킨다.
    input = tf.expand_dims(input, -1)
    # 합성곱 층 - 풀링 층
    x = self.layer1(input)
    x = self.layer2(x)
    # fully-connected 층
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.dropout(x, training=training)
    x = self.fc2(x)
    out = self.fc3(x)
    out = tf.nn.softmax(out)
    return out

def train_step(self, data):
    # 손실 함수를 통과시켜서 손실 계산하고 그레디언트 업데이트 하는 과정.
    images, labels = data

    with tf.GradientTape() as tape:
        outputs = self(images, training=True)
        preds = tf.argmax(outputs, 1)

        loss = self.compiled_loss(labels, outputs)

    # compute gradients
    trainable_vars = self.trainable_variables
    gradients = tape.gradient(loss, trainable_vars)

    # update weights
    self.optimizer.apply_gradients(zip(gradients, trainable_vars))

    # update the metrics
    self.compiled_metrics.update_state(labels, preds)

    # return a dict mapping metrics names to current values
    logs = {m.name: m.result() for m in self.metrics}
    logs.update({"loss": loss})
    return logs

# test_step 의 경우 일단 train_step 과 동일.
def test_step(self, data):
    images, labels = data
    outputs = self(images, training=True) # call 함수를 불러온다.
    preds = tf.argmax(outputs, 1) # 예측확률 계산, 가장 높은 값 불러오기

    loss = self.compiled_loss(labels, outputs)

    # update the metrics
    self.compiled_metrics.update_state(labels, preds)

    # return a dict mapping metrics names to current values
    logs = {m.name: m.result() for m in self.metrics}
    logs.update({"test_loss": loss})
    return logs

```

compile setting

컴파일 세팅 부분도 MLP 와 동일하게 한다.

```
n_class = 10
max_epoch = 50

with mirrored_strategy.scope():
    model = CNN()
    moel_name = type(mdoel).__name__

# define loss function
loss_funtion = tf.losses.SparseCategoricalCrossentropy()

# define learning rate
# if learning_scheduler exists, use it.
lr = 1e-3
scheduler = LinearWarmupLRScheduler(lr, 1500)
scheduler_name = type(scheduler).__name__ if scheduler is not None else "no_scheduler"
if scheduler is None:
    scheduler = lr

# define optimizer
optimizer = tf.optimizers.Adam(learning_rate=scheduler)
optimizer_name = type(optimizer).__name__

# model compile
model.compile(
    loss=loss_function,
    optimizer=optimizer,
    metrics=[tf.keras.metrics.Accuracy()],
)

model.build((1,28, 28))
model.summary()
```

>>>
Model: "cnn_1"

Layer (type)	Output Shape	Param #
conv_batch_norm_max_pool_1 (ConvBatchNormMaxPool)	multiple	448
conv_batch_norm_max_pool_2 (ConvBatchNormMaxPool)	multiple	18752
flatten_4 (Flatten)	multiple	0
dense_12 (Dense)	multiple	3965440
dense_13 (Dense)	multiple	65664
dense_14 (Dense)	multiple	1290
dropout_4 (Dropout)	multiple	0

=====
Total params: 4,051,594
Trainable params: 4,051,402
Non-trainable params: 192
=====

logging & callbacks

```
from datetime import datetime
drive_project_root = "/content/drive/Mydrive/source"
# 사실 위의 project root 설정은 sys.path 에서 다뤄야 한다.
# 모델 저장/관리를 위해 다뤄야 한다.

log_interval = 100

# 아래 제시된 run_name 은 optimizr 에 따라, learning rate 에 따른 실험을 위해 만든 이름 형식이다.
run_name = f"{datetime.now()}-{model_name}-{optimizer_name}-optim-{lr}-lr"
run_dirname = "CNN-tutorial-fashion-mnist-runs-tf"

log_dir = os.path.join(drive_project_root, "runs", run_dirname, run_name)
```

```
# callbacks - Tensorboard, Earlystopping
tb_callback = tf.keras.callbacks.TensorBoard(log_dir,
                                             update_freq=log_interval)

early_stop_callback = tf.keras.callbacks.EarlyStopping(patience=5, verbose=True)
```

```
# wandb 연동 전에 물론 wandb 로그인은 되어 있어야 한다.
# wandb setup
project_name = "fastcampus_fashion_mnist_tutorials_tf"
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,
    sync_tensorboard=True
)
```

```
%load_ext tensorboard
%tensorboard --logdir /content/drive/Mydrive/source/runs

model.fit(
    train_data_loader,
    validation_data=val_data_loader,
    epochs=max_epoch,
    callbacks=[tb_callback, early_stop_callback]
)
```

model testing(수정중)

```
# 모델 평가
model.evaluate(test_data_loader)

# calculate accuracy
test_labels_list = []
test_preds_list = []
test_outputs_list = []

for i, (test_images, test_labels) in enumerate(tqdm(test_data_loader, position=0, leave=True, desc='testing')):
    with mirrored_strategy.scope():
        test_outputs = model(test_images)
        test_preds = tf.argmax(test_outputs, 1)

        final_outs = test_outputs.numpy()
        test_outputs_list.extend(final_outs)
        test_preds_list.extend(test_preds.numpy())
        test_labels_list.extend(test_labels.numpy())

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

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