

Nama : Nabilah Salwa

NIM : 1103204060

UAS Machine Learning

05. PyTorch Going Modular

1. Turn the code to get the data (from section 1. Get Data) into a Python script, such as `get_data.py`.

```
%%writefile get_data.py
import os
import zipfile

from pathlib import Path

import requests

# Setup path to data folder
data_path = Path("data/")
image_path = data_path / "pizza_steak_sushi"

# If the image folder doesn't exist, download it and prepare it...
if image_path.is_dir():
    print(f"{image_path} directory exists.")
else:
    print(f"Did not find {image_path} directory, creating one...")
    image_path.mkdir(parents=True, exist_ok=True)

# Download pizza, steak, sushi data
with open(data_path / "pizza_steak_sushi.zip", "wb") as f:
    request = requests.get("https://github.com/mrdbourke/pytorch-deep-learning/raw/main/data/pizza_steak_sushi.zip")
    print("Downloading pizza, steak, sushi data...")
    f.write(request.content)

# Unzip pizza, steak, sushi data
with zipfile.ZipFile(data_path / "pizza_steak_sushi.zip", "r") as zip_ref:
    print("Unzipping pizza, steak, sushi data...")
    zip_ref.extractall(image_path)

# Remove zip file
os.remove(data_path / "pizza_steak_sushi.zip")
```

Kode di atas adalah script Python yang disimpan dalam file "get_data.py". Fungsinya adalah untuk mengunduh dataset gambar Pizza, Steak, dan Sushi dari GitHub. Jika folder yang ditentukan untuk menyimpan dataset sudah ada, program akan mencetak bahwa direktori tersebut sudah ada. Jika tidak, maka program akan membuat direktori tersebut. Selanjutnya, script mengunduh file zip dataset, mengekstraknya ke dalam folder yang telah disiapkan, dan menghapus file zip setelah selesai. Hal ini memudahkan pengguna untuk mengelola dataset secara otomatis dengan menjalankan script ini.

```
!python get_data.py
```

Perintah `!python get_data.py` digunakan untuk menjalankan script Python yang disimpan dalam file "get_data.py" di dalam lingkungan shell atau terminal. Script tersebut bertujuan

untuk mengunduh dataset gambar Pizza, Steak, dan Sushi dari GitHub, membuat struktur direktori jika belum ada, mengekstrak dataset ke dalam direktori tersebut, dan menghapus file zip setelah diekstrak. Dengan menjalankan perintah ini, pengguna dapat secara otomatis mendapatkan dan mempersiapkan dataset dengan mengautomatisasi proses tersebut melalui script Python.

2. Use Python's argparse module to be able to send the train.py custom hyperparameter values for training procedures.

```
%%writefile data_setup.py
"""
Contains functionality for creating PyTorch DataLoaders for
image classification data.
"""
import os

from torchvision import datasets, transforms
from torch.utils.data import DataLoader

NUM_WORKERS = os.cpu_count()

def create_dataloaders(
    train_dir: str,
    test_dir: str,
    transform: transforms.Compose,
    batch_size: int,
    num_workers: int=NUM_WORKERS
):
    """Creates training and testing DataLoaders.
    Takes in a training directory and testing directory path and turns
    them into PyTorch Datasets and then into PyTorch DataLoaders.
    Args:
        train_dir: Path to training directory.
        test_dir: Path to testing directory.
        transform: torchvision transforms to perform on training and testing data.
        batch_size: Number of samples per batch in each of the DataLoaders.
        num_workers: An integer for number of workers per DataLoader.
```

```

Returns:
    A tuple of (train_dataloader, test_dataloader, class_names).
    Where class_names is a list of the target classes.
Example usage:
    train_dataloader, test_dataloader, class_names = \
        = create_dataloaders(train_dir=path/to/train_dir,
                             test_dir=path/to/test_dir,
                             transform=some_transform,
                             batch_size=32,
                             num_workers=4)

"""
# Use ImageFolder to create dataset(s)
train_data = datasets.ImageFolder(train_dir, transform=transform)
test_data = datasets.ImageFolder(test_dir, transform=transform)

# Get class names
class_names = train_data.classes

# Turn images into data loaders
train_dataloader = DataLoader(
    train_data,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    pin_memory=True,
)
test_dataloader = DataLoader(
    test_data,
    batch_size=batch_size,
    shuffle=False,
    num_workers=num_workers,
    pin_memory=True,
)

return train_dataloader, test_dataloader, class_names

```

Script `data_setup.py` menyediakan fungsionalitas untuk membuat DataLoader PyTorch dalam konteks klasifikasi gambar. Fungsi `create_dataloaders` dalam script ini menerima jalur direktori pelatihan dan pengujian, transformasi torchvision untuk diterapkan pada data pelatihan dan pengujian, ukuran batch, serta jumlah pekerja yang diinginkan untuk DataLoader. Fungsi ini menghasilkan dua DataLoader untuk pelatihan dan pengujian, bersama dengan daftar nama kelas target. Contoh penggunaan script ini adalah menghasilkan DataLoader untuk proses pelatihan dan pengujian dalam tugas klasifikasi gambar dengan PyTorch.

```

%%writefile engine.py
"""
Contains functions for training and testing a PyTorch model.
"""
import torch

from tqdm.auto import tqdm
from typing import Dict, List, Tuple

def train_step(model: torch.nn.Module,
               dataloader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               device: torch.device) -> Tuple[float, float]:
    """Trains a PyTorch model for a single epoch.
    Turns a target PyTorch model to training mode and then
    runs through all of the required training steps (forward
    pass, loss calculation, optimizer step).
    Args:
    model: A PyTorch model to be trained.
    dataloader: A DataLoader instance for the model to be trained on.
    loss_fn: A PyTorch loss function to minimize.
    optimizer: A PyTorch optimizer to help minimize the loss function.
    device: A target device to compute on (e.g. "cuda" or "cpu").
    Returns:
    A tuple of training loss and training accuracy metrics.
    In the form (train_loss, train_accuracy). For example:
    (0.1112, 0.8743)
    """
    # Put model in train mode
    model.train()

    # Setup train loss and train accuracy values
    train_loss, train_acc = 0, 0

    # Loop through data loader data batches
    for batch, (X, y) in enumerate(dataloader):
        # Send data to target device
        X, y = X.to(device), y.to(device)

        # 1. Forward pass
        y_pred = model(X)

        # 2. Calculate and accumulate loss
        loss = loss_fn(y_pred, y)
        train_loss += loss.item()

        # 3. Optimizer zero grad
        optimizer.zero_grad()

        # 4. Loss backward
        loss.backward()

        # 5. Optimizer step
        optimizer.step()

        # Calculate and accumulate accuracy metric across all batches
        y_pred_class = torch.argmax(torch.softmax(y_pred, dim=1), dim=1)
        train_acc += (y_pred_class == y).sum().item()/len(y_pred)

    # Adjust metrics to get average loss and accuracy per batch
    train_loss = train_loss / len(dataloader)
    train_acc = train_acc / len(dataloader)
    return train_loss, train_acc

```

```

def test_step(model: torch.nn.Module,
              dataloader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              device: torch.device) -> Tuple[float, float]:
    """Tests a PyTorch model for a single epoch.
    Turns a target PyTorch model to "eval" mode and then performs
    a forward pass on a testing dataset.
    Args:
    model: A PyTorch model to be tested.
    dataloader: A DataLoader instance for the model to be tested on.
    loss_fn: A PyTorch loss function to calculate loss on the test data.
    device: A target device to compute on (e.g. "cuda" or "cpu").
    Returns:
    A tuple of testing loss and testing accuracy metrics.
    In the form (test_loss, test_accuracy). For example:
    (0.0223, 0.8985)
    """

    # Put model in eval mode
    model.eval()

    # Setup test loss and test accuracy values
    test_loss, test_acc = 0, 0

    # Turn on inference context manager
    with torch.inference_mode():
        # Loop through DataLoader batches
        for batch, (X, y) in enumerate(dataloader):
            # Send data to target device
            X, y = X.to(device), y.to(device)

            # 1. Forward pass
            test_pred_logits = model(X)

            # 2. Calculate and accumulate loss
            loss = loss_fn(test_pred_logits, y)
            test_loss += loss.item()

            # Calculate and accumulate accuracy
            test_pred_labels = test_pred_logits.argmax(dim=1)
            test_acc += ((test_pred_labels == y).sum().item() / len(test_pred_labels))

    # Adjust metrics to get average loss and accuracy per batch
    test_loss = test_loss / len(dataloader)
    test_acc = test_acc / len(dataloader)
    return test_loss, test_acc

def train(model: torch.nn.Module,
          train_dataloader: torch.utils.data.DataLoader,
          test_dataloader: torch.utils.data.DataLoader,
          optimizer: torch.optim.Optimizer,
          loss_fn: torch.nn.Module,
          epochs: int,
          device: torch.device) -> Dict[str, List]:
    """Trains and tests a PyTorch model.
    Passes a target PyTorch models through train_step() and test_step()
    functions for a number of epochs, training and testing the model
    in the same epoch loop.
    Calculates, prints and stores evaluation metrics throughout.

```

```

Args:
model: A PyTorch model to be trained and tested.
train_dataloader: A DataLoader instance for the model to be trained on.
test_dataloader: A DataLoader instance for the model to be tested on.
optimizer: A PyTorch optimizer to help minimize the loss function.
loss_fn: A PyTorch loss function to calculate loss on both datasets.
epochs: An integer indicating how many epochs to train for.
device: A target device to compute on (e.g. "cuda" or "cpu").
Returns:
A dictionary of training and testing loss as well as training and
testing accuracy metrics. Each metric has a value in a list for
each epoch.
In the form: {train_loss: [...],
               train_acc: [...],
               test_loss: [...],
               test_acc: [...]}
For example if training for epochs=2:
{train_loss: [2.0616, 1.0537],
 train_acc: [0.3945, 0.3945],
 test_loss: [1.2641, 1.5706],
 test_acc: [0.3400, 0.2973]}
"""
# Create empty results dictionary
results = {"train_loss": [],
           "train_acc": [],
           "test_loss": [],
           "test_acc": []
}

```

```

# Loop through training and testing steps for a number of epochs
for epoch in tqdm(range(epochs)):
    train_loss, train_acc = train_step(model=model,
                                       dataloader=train_dataloader,
                                       loss_fn=loss_fn,
                                       optimizer=optimizer,
                                       device=device)

    test_loss, test_acc = test_step(model=model,
                                    dataloader=test_dataloader,
                                    loss_fn=loss_fn,
                                    device=device)

    # Print out what's happening
    print(
        f"Epoch: {epoch+1} | "
        f"train_loss: {train_loss:.4f} | "
        f"train_acc: {train_acc:.4f} | "
        f"test_loss: {test_loss:.4f} | "
        f"test_acc: {test_acc:.4f}"
    )

    # Update results dictionary
    results["train_loss"].append(train_loss)
    results["train_acc"].append(train_acc)
    results["test_loss"].append(test_loss)
    results["test_acc"].append(test_acc)

# Return the filled results at the end of the epochs
return results

```

Script `engine.py` menyediakan fungsi-fungsi untuk melatih dan menguji model PyTorch. Fungsi `train_step` dan `test_step` masing-masing menangani langkah-langkah pelatihan dan pengujian model dalam satu epoch, termasuk perhitungan loss, optimasi, dan akurasi. Fungsi `train` mengkoordinasikan pelatihan dan pengujian model untuk sejumlah epoch yang ditentukan, dengan hasil berupa metrik pelatihan dan pengujian seperti loss dan akurasi untuk setiap epoch, yang dicetak dan disimpan dalam sebuah dictionary.

```
%%writefile model_builder.py
"""
Contains PyTorch model code to instantiate a TinyVGG model.
"""
import torch
from torch import nn

class TinyVGG(nn.Module):
    """Creates the TinyVGG architecture.
    Replicates the TinyVGG architecture from the CNN explainer website in PyTorch.
    See the original architecture here: https://poloclub.github.io/cnn-explainer/
    Args:
        input_shape: An integer indicating number of input channels.
        hidden_units: An integer indicating number of hidden units between layers.
        output_shape: An integer indicating number of output units.
    """
    def __init__(self, input_shape: int, hidden_units: int, output_shape: int) -> None:
        super().__init__()
        self.conv_block_1 = nn.Sequential(
            nn.Conv2d(in_channels=input_shape,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=0),
            nn.ReLU(),
            nn.Conv2d(in_channels=hidden_units,
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=0),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,
                         stride=2)
        )
```

```

self.conv_block_2 = nn.Sequential(
    nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
    nn.ReLU(),
    nn.Conv2d(hidden_units, hidden_units, kernel_size=3, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
self.classifier = nn.Sequential([
    nn.Flatten(),
    # Where did this in_features shape come from?
    # It's because each layer of our network compresses and changes the shape of our inputs data.
    nn.Linear(in_features=hidden_units*13*13,
              out_features=output_shape)
])

def forward(self, x: torch.Tensor):
    x = self.conv_block_1(x)
    x = self.conv_block_2(x)
    x = self.classifier(x)
    return x
# return self.classifier(self.block_2(self.block_1(x))) # <- leverage the benefits of operator fusion

```

`model_builder.py` berisi definisi kelas `TinyVGG` yang mengimplementasikan arsitektur TinyVGG dalam PyTorch. Arsitektur ini terdiri dari dua blok konvolusi, diikuti oleh lapisan pengklasifikasi yang mencakup operasi konvolusi, aktivasi ReLU, dan max pooling. Kelas ini menerima parameter seperti `input_shape`, `hidden_units`, dan `output_shape`, yang memungkinkan pengguna untuk menyesuaikan ukuran dan kompleksitas model sesuai kebutuhan. Metode `forward` mendefinisikan alur maju (forward pass) melalui model.


```

%%writefile utils.py
"""
Contains various utility functions for PyTorch model training and saving.
"""
import torch
from pathlib import Path

def save_model(model: torch.nn.Module,
               target_dir: str,
               model_name: str):
    """Saves a PyTorch model to a target directory.
    Args:
        model: A target PyTorch model to save.
        target_dir: A directory for saving the model to.
        model_name: A filename for the saved model. Should include
            either ".pth" or ".pt" as the file extension.
    Example usage:
        save_model(model=model_0,
                   target_dir="models",
                   model_name="05_going_modular_tingvgg_model.pth")
    """
    # Create target directory
    target_dir_path = Path(target_dir)
    target_dir_path.mkdir(parents=True,
                          exist_ok=True)

    # Create model save path
    assert model_name.endswith(".pth") or model_name.endswith(".pt"), "model_name should end with '.pt' or '.pth'"
    model_save_path = target_dir_path / model_name

    # Save the model state_dict()
    print(f"[INFO] Saving model to: {model_save_path}")
    torch.save(obj=model.state_dict(),
               f=model_save_path)

```

`utils.py` berisi fungsi utilitas untuk menyimpan model PyTorch. Fungsi `save_model` menerima model, direktori target, dan nama model sebagai parameter, kemudian membuat direktori target jika belum ada, dan menyimpan state_dict dari model ke file dengan ekstensi ".pt" atau ".pth". Fungsi ini berguna untuk menyimpan kemajuan pelatihan model untuk penggunaan atau evaluasi selanjutnya.

```

%%writefile train.py
"""
Trains a PyTorch image classification model using device-agnostic code.
"""

import os
import argparse

import torch

from torchvision import transforms

import data setup, engine, model builder, utils

# Create a parser
parser = argparse.ArgumentParser(description="Get some hyperparameters.")

# Get an arg for num_epochs
parser.add_argument("--num_epochs",
                    default=10,
                    type=int,
                    help="the number of epochs to train for")

# Get an arg for batch_size
parser.add_argument("--batch_size",
                    default=32,
                    type=int,
                    help="number of samples per batch")

# Get an arg for hidden_units
parser.add_argument("--hidden_units",
                    default=10,
                    type=int,
                    help="number of hidden units in hidden layers")

# Get an arg for learning_rate
parser.add_argument("--learning_rate",
                    default=0.001,
                    type=float,
                    help="learning rate to use for model")

# Create an arg for training directory
parser.add_argument("--train_dir",
                    default="data/pizza_steak_sushi/train",
                    type=str,
                    help="directory file path to training data in standard image classification format")

# Create an arg for test directory
parser.add_argument("--test_dir",
                    default="data/pizza_steak_sushi/test",
                    type=str,
                    help="directory file path to testing data in standard image classification format")

# Get our arguments from the parser
args = parser.parse_args()

# Setup hyperparameters
NUM_EPOCHS = args.num_epochs
BATCH_SIZE = args.batch_size
HIDDEN_UNITS = args.hidden_units
LEARNING_RATE = args.learning_rate
print(f"[INFO] Training a model for {NUM_EPOCHS} epochs with batch size {BATCH_SIZE} using {HIDDEN_UNITS} hidden units and a learning rate of {LEARNING_RATE}")

```

```

# Setup directories
train_dir = args.train_dir
test_dir = args.test_dir
print(f"[INFO] Training data file: {train_dir}")
print(f"[INFO] Testing data file: {test_dir}")

# Setup target device
device = "cuda" if torch.cuda.is_available() else "cpu"

# Create transforms
data_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor()
])

# Create DataLoaders with help from data_setup.py
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(
    train_dir=train_dir,
    test_dir=test_dir,
    transform=data_transform,
    batch_size=BATCH_SIZE
)

# Create model with help from model_builder.py
model = model_builder.TinyVGG(
    input_shape=3,
    hidden_units=HIDDEN_UNITS,
    output_shape=len(class_names)
).to(device)

# Set loss and optimizer
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),
                               lr=LEARNING_RATE)

# Start training with help from engine.py
engine.train(model=model,
             train_dataloader=train_dataloader,
             test_dataloader=test_dataloader,
             loss_fn=loss_fn,
             optimizer=optimizer,
             epochs=NUM_EPOCHS,
             device=device)

# Save the model with help from utils.py
utils.save_model(model=model,
                 target_dir="models",
                 model_name="05_going_modular_script_mode_tinyvgg_model.pth")

```

`train.py` adalah skrip untuk melatih model klasifikasi gambar menggunakan PyTorch dengan pendekatan yang dapat digunakan di berbagai perangkat. Skrip ini menggunakan argumen baris perintah untuk mengatur hyperparameter seperti jumlah epoch, ukuran batch, jumlah unit tersembunyi, dan tingkat pembelajaran. Data pelatihan dan pengujian dimuat menggunakan `data_setup.py`, model dibangun dengan arsitektur TinyVGG dari `model_builder.py`, dan pelatihan dilakukan menggunakan fungsi `train` dari `engine.py`. Selain itu, model yang dilatih disimpan menggunakan fungsi `save_model` dari `utils.py`.

```
!python train.py --num_epochs 5 --batch_size 128 --hidden_units 128 --learning_rate 0.0003
```

Perintah `!python train.py --num_epochs 5 --batch_size 128 --hidden_units 128 --learning_rate 0.0003` menjalankan skrip pelatihan (`train.py`) untuk melatih model klasifikasi gambar dengan hyperparameter yang ditentukan melalui argumen baris perintah. Hyperparameter tersebut mencakup jumlah epoch (5), ukuran batch (128), jumlah unit tersembunyi (128), dan tingkat pembelajaran (0.0003). Skrip menggunakan TinyVGG sebagai arsitektur model, dan hasil pelatihan disimpan dalam direktori "models" dengan nama file "05_going_modular_script_mode_tinyvgg_model.pth".

3. Create a Python script to predict (such as `predict.py`) on a target image given a file path with a saved model.

```
%%writefile predict.py
import torch
import torchvision
import argparse

import model_builder

# Creating a parser
parser = argparse.ArgumentParser()

# Get an image path
parser.add_argument("--image",
                    help="target image filepath to predict on")

# Get a model path
parser.add_argument("--model_path",
                    default="models/05_going_modular_script_mode_tinyvgg_model.pth",
                    type=str,
                    help="target model to use for prediction filepath")

args = parser.parse_args()

# Setup class names
class_names = ["pizza", "steak", "sushi"]

# Setup device
device = "cuda" if torch.cuda.is_available() else "cpu"

# Get the image path
IMG_PATH = args.image
print(f"[INFO] Predicting on {IMG_PATH}")
```

```

# Function to load in the model
def load_model(filepath=args.model_path):
    # Need to use same hyperparameters as saved model
    model = model_builder.TinyVGG(input_shape=3,
                                   hidden_units=128,
                                   output_shape=3).to(device)

    print(f"[INFO] Loading in model from: {filepath}")
    # Load in the saved model state dictionary from file
    model.load_state_dict(torch.load(filepath))

    return model

# Function to load in model + predict on select image
def predict_on_image(image_path=IMG_PATH, filepath=args.model_path):
    # Load the model
    model = load_model(filepath)

    # Load in the image and turn it into torch.float32 (same type as model)
    image = torchvision.io.read_image(str(IMG_PATH)).type(torch.float32)

    # Preprocess the image to get it between 0 and 1
    image = image / 255.

    # Resize the image to be the same size as the model
    transform = torchvision.transforms.Resize(size=(64, 64))
    image = transform(image)

    # Predict on image
    model.eval()
    with torch.inference_mode():
        # Put image to target device
        image = image.to(device)

    # Get pred logits
    pred_logits = model(image.unsqueeze(dim=0)) # make sure image has batch dimension (shape: [batch_size, height, width, color_channels])

    # Get pred probs
    pred_prob = torch.softmax(pred_logits, dim=1)

    # Get pred labels
    pred_label = torch.argmax(pred_prob, dim=1)
    pred_label_class = class_names[pred_label]

    print(f"[INFO] Pred class: {pred_label_class}, Pred prob: {pred_prob.max():.3f}")

if __name__ == "__main__":
    predict_on_image()

```

Skrip `predict.py` digunakan untuk melakukan prediksi kelas pada suatu gambar menggunakan model TinyVGG yang telah dilatih. Dengan menggunakan argumen baris perintah, kita dapat menyediakan path gambar target (`--image`) dan path model yang akan digunakan (`--model_path`). Gambar akan di-preprocess, diprediksi oleh model, dan hasil prediksi berupa label kelas dan probabilitasnya akan dicetak ke konsol. Model yang digunakan di-load dari path yang disediakan pada argumen atau menggunakan path default jika tidak disediakan.

```
!python predict.py --image data/pizza_steak_sushi/test/sushi/175783.jpg
```

Panggilan skrip `predict.py` dijalankan untuk melakukan prediksi kelas pada gambar sushi yang diberikan (`175783.jpg`) menggunakan model TinyVGG yang telah dilatih. Skrip memuat model dari path default (`models/05_going_modular_script_mode_tinyvgg_model.pth`), meng-preprocess gambar, dan memberikan hasil prediksi berupa label kelas dan probabilitasnya pada konsol. Pada contoh ini, hasil prediksi untuk gambar sushi tersebut dicetak ke layar.