mlr

make ___Task( )

make Learner ( )

make ______ wrapper(), resample(), tune Param(), SetHype
Impute/filter/tune

setHyper Pars( )
↳ lrn= , par.vals= tp$x

impute Learner( )

make ParamSet (
make Numeric Param( ))

make Resample Desc( )

make Tune control Grid( )

```
from feature_engine.wrappers import SklearnTransformerWrapper
from sklearn.preprocessing import OneHotEncoder
from feature_engine.imputation import CategoricalImputer
from feature_engine.selection import SelectBySingleFeaturePerformance
import numpy as np
import pandas as pd
import matplotlib as mpl
import seaborn as sns
import missingno as miss
import sklearn as sklearn
import feature_engine
```

```
from sklearn.model_selection import StratifiedKFold,GridSearchCV,RandomizedSearchCV,cross_val_score
```

{Train( ), predict( ), performance

Scikit learn

```
imdb_data=imdb_encode.drop(["averageRating"],axis=1)
imdb_target=imdb_encode["averageRating"]
```

data and target split korba

Lrn= (এ model ই select করো

from feature_engine.wrapper import SklearnTransformerWrapper.

SkLearn Transformer Wrapper ( ) ↳transformer= function( )), variable=[ ] cross_val_score ( ) ↳x,y,cv

gridsearch.best_params_ , grid search. best_estimator_( ) ↳ best model

PS= { "x": [1,2,3]}

Stratified Kfold ( )

Grid Search CV ( ) = grid search

best_model.fit(X,y) best_model. Predict( ), accuracy_score

1) PCA: prcomp → pca$x[1], pca$x[2]

2) tsne/umap: library (tsne) | library(umap)

Rtsne( )

preplexity = width বা probability তে convert করে (5-50)

Rtsne$Y[1], Rtsne$Y[2]

theta = trade off between speed and accuracy

umap$layout[1]
umap$layout[2]

eta = learning rate

max_iter = মোট iteration এর সংখ্যা

? Rtsne.defaults

df %>% as.matrix( ) %>% ump( )

n_neighbours = মোট near value manifold এ দিবে,

min_dist = মিনিমাম case এ মোট min দূরত্ব.

metric = euclidean

n_epochs = number of iteration

? umap.defaults

3) som / lle: library (kohonen) | library(Rdimtools)

? somGrid ( )
xdim =
ydim =
topo = "hexagonal"
neighbourhood = fct নির্বাচন করার method এ

So Som

? supersom ( )
alpha = learningrate = c(min, max)
rlen = iteration number.
grid = somGrid
plot(som, type = , shape)

do.lle (df%matrix( ), ndim=2)

df কে matrix

ndim = মোট dimension এ convert করে.

lle$Y[1], lle$Y[2]

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Assume `X` is your data matrix
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Plotting
plt.scatter(X_pca[:, 0], X_pca[:, 1])
plt.title("PCA")
plt.show()
```

```
import umap.umap_ as umap

# UMAP reduction
umap_reducer = umap.UMAP(n_components=2)
X_umap = umap_reducer.fit_transform(X)
```

```
from sklearn.manifold import TSNE

# t-SNE reduction
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X)
```

```
from sklearn.manifold import LocallyLinearEmbedding

# LLE reduction
lle = LocallyLinearEmbedding(n_components=2)
X_lle = lle.fit_transform(X)
```

```
from minisom import MiniSom

# SOM parameters
som = MiniSom(x=10, y=10,
input_len=X.shape[1], sigma=1.0,
learning_rate=0.5)
som.random_weights_init(X)
som.train_random(X, 100)
```

Clusten

– analysis –

```
librarary(clvalid)
library(cluster)
library(factoextra)
library(mclust);library(fpc//dbscan)
```

```
clvalid (df, nClust = 2:6; FunClus = "agnes",
                           method = "euclidean"
                           metric = "internal")
```

clValid(up.scale.df,nClust = 3:6,clMethods = "agnes",validation = "internal",method = "complete",metric = "euclidean")

↳ summary (clvalid)

↳ optimal → use.

(facto Extra)

fviz_nbclust( ) → "wss", "Silhouette",

agnes/ dian

fviz_dend ( )

fvin_ cluster ( )

(nclust)

mclust ( ) → automates

dbscan
fpc

```
map(2:10, function (x){
  fpc::dbscan (df, eps= kNNdistplot ( ),
                   minpts= x)
```

↓ same }}

```
x= hdbscan (df, minpts= ) → eps
x → cbind. noise
cbind ( pca.data , cluster= cluster$ x $ cluster)
                    cprcomp(pca[,1], pca[,2])
```

DL work flow:

① train test val dataset তৈরি করা,

keras.utils.image_dataset_from_directory()
keras.utils.timeseries_dataset_from_array()
keras.utils.text_dataset_from_directory()

② Imoptrt tensorflow as tf.

③ FROM tensorflow import keras

FROM keras import layers

```
input= keras.input (shape= depends)

x = layers.Dense(16, activation="relu")
                                (input)

for shape in (2,6):
  residual= x
  x= layers.Dense(shape,..., activation="relu"
                                (x)
  x = layers.Dropout(0.5) (x)
  x = layers.add ([x, residual])
x = Layer.Activation("relu")(x)

output = Layer.Dense (1, activation = "linear")
                                (feature)

model = keras.Model(input, output)
```

⑬ model.summary()

ekhane mone rakhbo residual er shape & dense layer er shape same thakbe and residual network prottek shape er layer jonno layer jonno create korbe

```
sequence_length = 168
num_features=17
inputs=keras.Input(shape=(sequence_length,num_features))
x=layers.LSTM(128,recurrent_dropout=0.3,return_sequences=True,unroll=True)(inputs)
residual=x
x=layers.Dropout(0.5)(x)
x=layers.Dense(128,activation="relu")(x)
x=layers.add([x,residual])
x=layers.LSTM(256,recurrent_dropout=0.3,return_sequences=True,unroll=True)(inputs)
residual=x
x=layers.Dropout(0.5)(x)
x=layers.Dense(256,activation="relu")(x)
x=layers.add([x,residual])
x=layers.GlobalAveragePooling1D()(x)
output=layers.Dense(1,activation="sigmoid")(x)

model=keras.Model(inputs,output)
```

residual & x এর shape same

residual & x এর shape same

Prottek Shape এর জন্য আলাদা residual network

```
dataset = timeseries_dataset_from_array(data=data, targets=targets,
                                        sequence_length=sequence_length,
                                        batch_size=batch_size)
```

```
xtest = np.concatenate([x[0].numpy() for x in test_dataset], axis=0)
y_true=np.concatenate([x[1].numpy() for x in test_dataset], axis=0)
lstm_model.fit(x_train, y_train, epochs=10, batch_size=batch_size, verbose=1)
```

④ model.compile(optimizer="rmsprop",
loss="mse",
metrics=["mae"])

callbacks=[keras.callbacks.ModelCheckpoint
("best.keras",save_best_only=True)
callbacks=keras.callbacks.EarlyStopping
(monitor="val_loss",
patience=20)

⑤ ~~history=keras.~~
history=model.fit(X_train,y_train,
callbacks=["callbacks"],
validation_data=(X_val,y_val),
epochs=100,
batch_size=50)

best_model_mae=keras.Model.load_model
("best.keras")
val,mae=best_model_mae.evaluate(
X_test,
y_test)

```
encoder_inputs = Input(shape=(sequence_length, features))
x = LSTM(32)(encoder_inputs)
latent = Dense(latent_dim)(x)

# Decoder
x = RepeatVector(sequence_length)(latent)
x = LSTM(32, return_sequences=True)(x)
decoder_outputs = TimeDistributed(Dense(features))(x)
```

The TimeDistributed layer in Keras is used to apply a layer independently to each time step of a sequence. This is particularly useful in sequence modeling tasks, such as time series analysis or sequence-to-sequence models, where we need to process each timestep in the sequence in a similar manner (e.g., applying a dense layer or convolution to each timestep).

The RepeatVector layer in Keras is used to replicate or "repeat" the input data across a specified number of timesteps. It's often used in sequence generation tasks, particularly in models like VAEs and seq2seq, where we want to take a single latent vector (output from an encoder or bottleneck layer) and expand it back

— revision —

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import from torch.utils.data import
                    DataLoader,
                    random_split
from torchvision.dataset import ImageFolder
from torchvision import transforms.

transform = transform.Compose([
        transform.Resize((128,128)),
        transforms.ToTensor(),
        transforms.Normalize((0.5,),
                             (0.5,))
    ])
dataset = ImageFolder(root=data.path,
                      transform=
                          transform)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(
        full_dataset, [train_size,
                       val_size])
```

x_train, [img[0] for img in train_dataset]
y_train: [img[1] for img in train_dataset]

train_loader: DataLoader (train_dataset,
batch_size=32,
shuffle=TRUE)

test_loader= DataLoader(test_dataset,
batch_size=32,
shuffle=False)

val_loader = DataLoader(val_dataset,
batch_size=32,
shuffle=False)

---

class ResidualNetwork(nn.Module):
def __init__(self, input_size=784,
hidden_size=128,
num_classes=10):
super(ResidualNetwork, self).__init__()
self.fc1 = nn.Linear(input_size,
hidden_size)
self.fc2= nn.Linear(hidden_size,
hidden_size)
self.fc3= nn.Linear(hidden_size,
num_classes)

parameters of layers or layer to define

models ensemble

```
def forward(self, x):
    x = x.view(-1, 784)  # flatten the input
    # first layer.
    out = self.fc1(x)
    out = F.relu(out)

    # residual connection:
    residual = out
    out = self.fc2(out)
    out = F.relu(out)
    out += residual  [out = out + residual (2nd output size same)]
    out = self.fc3(out)
    return out
```

input pass : functional api : layer style structure

```
# Instantiate the model, define loss and optimizer:
model = ResidualNetwork()
model = ResidualNetwork().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learningrate)
```

```
inputs=layers.Input(shape=(sequence_length,3))
y1=model(inputs)
y2=model_lstm(inputs)
output = 0.3*y1 + 0.7*y2

ensemble_model = keras.Model(inputs = inputs, outputs = output)
ensemble_model.compile(optimizer="rmsprop",
        loss="mse",
        metrics=[
            keras.metrics.RootMeanSquaredError(name="rmse"),  # RMSE
            keras.metrics.MeanAbsoluteError(name="mae"),  # MAE
        ]
        )
ensemble_model.fit(train_dataset,validation_data=val_dataset,epochs=200,callbacks=
[callbacks3,early],batch_size=256)
```

train loops:

```
def train_model(model, train_loader,
                val_loader,
                criterion,
                optimizer,
                num_epochs):
    for epochs in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
```

https://github.com/mrdbourke/pytorch-deep-learning/blob/main/05_pytorch_going_modular.md

```
        for image, labels in train_
                              loader:
            image, labels = imges.to(device),
                            labels.to(devie)
```

gradient zero করলাম loop এর জন্য

```
            optimizer.zero_grad()
```

model এ input image

```
            output = model(images)
```

loss calculate

```
            loss = criterion(output, labels)
```

loss for এর calculate

```
            loss.backward()
```

optimizer এর step বাড়াবে

```
            optimizer.step()
            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels)
                       .sum().item()
```

```
train_loss = running_loss/len(train_loader)
train_acc = 100.0 * connect /total

val_loss, val$_acc = evaluate_model(model,
                                    val_loader,
                                    criterion,
print(f"                    ")
```

```
def evaluate_model(model, data_loader,
                                    criterion):

    model.eval()
    running_loss = 0.0
     connect = 0
    total  = 0
    with torch.no_grad():
        for images, labels in data_loader:
            images, labels = images.to(device)
                            ,labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
```

```
        connect += predicted.eq(
                                labels).
                                    sum().
                                    item()

    avg_loss = runng_loss / len(data_load
                                    er)

    accuracy = 100.0 * connect/total

    return  avg_loss, accuracy

train_model()

evaluate_model()

torch.save(model.state_dict(),
                                "xyz.pth")

# Load model:

    load = Residual Network().to(
                                device)

    load.load_state_dict(torch.load
                            ("xyz.pth"))
```

```
model.eval()  # Set model to evaluation mode

with torch.inference_mode():
    outputs = model(inputs)  # Forward pass
    loss = criterion(outputs, targets)  # Calculate the loss if needed

print(f"Loss: {loss.item():.4f}")
```

```
# Forward pass with training=False to set the
model in inference mode
predictions = model(inputs, training=False)
loss = loss_fn(targets, predictions)
print(f"Loss: {loss.numpy():.4f}")
```

train test song

```
#train
for epoch in range
go model.train()
do the forword pass
  [ypred = model(x_train)
calculate the loss
  loss_fn(y_pred, y_train)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

```
#test
go model.eval()
with torch.inference
                _mode():
do the forword pass:
  [ test_pred = model(x_test)
calculate the loss
  test_loss = loss_fn(
          test_pred,
              y_test)
print out whats happeni
```

choosing loss_fn:

mutiClass_classification : CrossEntropy Loss

Binary_classification : BCE Loss

Regression : MSE loss

Muti_Label classification : MultiLabelSoft Margin loss

```
for epoch in range(num_epochs):
    model.train()
    outputs = model(inputs)
    loss =loss_fn(outputs, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
for epoch in range(num_epochs):
    with tf.GradientTape() as tape:
        predictions = model(inputs, training=True)
        loss = loss_fn(targets, predictions)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

The forward pass (model(inputs)) computes predictions.
The loss function is calculated directly.
optimizer.zero_grad() in PyTorch corresponds to starting a GradientTape in Keras, which tracks gradients for the backward pass.
loss.backward() in PyTorch corresponds to tape.gradient in Keras.
optimizer.step() in PyTorch is equivalent to optimizer.apply_gradients in Keras.

```
import os
from IPython.display import Image, display
from keras.utils import load_img
from PIL import ImageOps
import keras
import numpy as np
from tensorflow import data as tf_data
from tensorflow import image as tf_image
from tensorflow import io as tf_io
import random

# Directories
input_dir = "images/"
target_dir = "annotations/trimaps/"
img_size = (160, 160)
num_classes = 3
batch_size = 32

# Collect input and target file paths
input_img_paths = sorted(
    [
        os.path.join(input_dir, fname)
        for fname in os.listdir(input_dir)
        if fname.endswith(".jpg")
    ]
)
target_img_paths = sorted(
    [
        os.path.join(target_dir, fname)
        for fname in os.listdir(target_dir)
        if fname.endswith(".png") and not fname.startswith(".")
    ]
)

print("Number of samples:", len(input_img_paths))

# Display some input-target path pairs
for input_path, target_path in zip(input_img_paths[:10],
target_img_paths[:10]):
    print(input_path, "|", target_path)

# Display an example input image and its corresponding target
display(Image(filename=input_img_paths[9]))
img = ImageOps.autocontrast(load_img(target_img_paths[9]))
display(img)

# Dataset preparation function
def get_dataset(batch_size, img_size, input_img_paths,
target_img_paths, max_dataset_len=None):
    """
    Returns a TensorFlow Dataset for image segmentation.
    """
```

dataset.batch(batch_size):
Batches the dataset into groups of batch_size. This is important for model training, where each batch will be processed by the model during training.

dataset = dataset.map(load_img_masks, num_parallel_calls=tf_data.AUTOTUNE):
Applies the load_img_masks function to each pair of input and target image paths in the dataset. This will load and preprocess the images and masks. The num_parallel_calls=tf_data.AUTOTUNE optimizes the number of parallel calls for faster processing.

```
    def load_img_masks(input_img_path, target_img_path):
        # Load and preprocess input image
        input_img = tf_io.read_file(input_img_path)
        input_img = tf_io.decode_png(input_img, channels=3)
        input_img = tf_image.resize(input_img, img_size)
        input_img = tf_image.convert_image_dtype(input_img,
"float32")

        # Load and preprocess target mask
        target_img = tf_io.read_file(target_img_path)
        target_img = tf_io.decode_png(target_img, channels=1)
        target_img = tf_image.resize(target_img, img_size,
method="nearest")
        target_img = tf_image.convert_image_dtype(target_img,
"uint8")

        # Adjust target mask to start from 0
        target_img -= 1

        return input_img, target_img

    # Limit dataset size for faster debugging
    if max_dataset_len:
        input_img_paths = input_img_paths[:max_dataset_len]
        target_img_paths = target_img_paths[:max_dataset_len]

    # Create TensorFlow Dataset
    dataset = tf_data.Dataset.from_tensor_slices((input_img_paths,
target_img_paths))
    dataset = dataset.map(load_img_masks,
num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)

# Shuffle and split the dataset into training and validation sets
val_samples = 1000
random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(target_img_paths)

train_input_img_paths = input_img_paths[:-val_samples]
train_target_img_paths = target_img_paths[:-val_samples]
val_input_img_paths = input_img_paths[-val_samples:]
val_target_img_paths = target_img_paths[-val_samples:]

# Create training and validation datasets
train_dataset = get_dataset(
    batch_size, img_size, train_input_img_paths,
train_target_img_paths, max_dataset_len=1000
)
valid_dataset = get_dataset(batch_size, img_size,
val_input_img_paths, val_target_img_paths)
```

dataset = tf_data.Dataset.from_tensor_slices((input_img_paths, target_img_paths)):
Converts the lists of image paths and target paths into a TensorFlow Dataset. Each item in the dataset will contain a pair of image path and mask path.

COCO Dataset Processing Code

```
import os
import json
import numpy as np
from pycocotools.coco import COCO
from pycocotools.mask import decode
from tensorflow import data as tf_data
from tensorflow import image as tf_image
from tensorflow import io as tf_io

# Paths
input_dir = "images/"  # Directory containing image files
annotation_dir = "annotations/json/"  # Directory containing multiple
annotation JSON files

img_size = (160, 160)
batch_size = 32

# Load COCO Annotations from Multiple Files
def load_coco_annotations(annotation_dir):
    """Load annotations from multiple COCO JSON files."""
    coco_objects = []
    for filename in os.listdir(annotation_dir):
        if filename.endswith(".json"):
            annotation_path = os.path.join(annotation_dir, filename)
            with open(annotation_path, 'r') as f:
                _ = json.load(f)  # Load and validate JSON
            coco = COCO(annotation_path)
            coco_objects.append(coco)
    return coco_objects

# Extract image IDs and their paths from all annotation files
def get_image_paths_and_ids(coco_objects, input_dir):
    """Extract image paths and their IDs from multiple COCO objects."""
    image_ids = []
    input_img_paths = []
    for coco in coco_objects:
        ids = coco.getImgIds()
        image_ids.extend(ids)
        paths = [
            os.path.join(input_dir,
coco.loadImgs(img_id)[0]['file_name'])
            for img_id in ids
        ]
        input_img_paths.extend(paths)
    return image_ids, input_img_paths
```

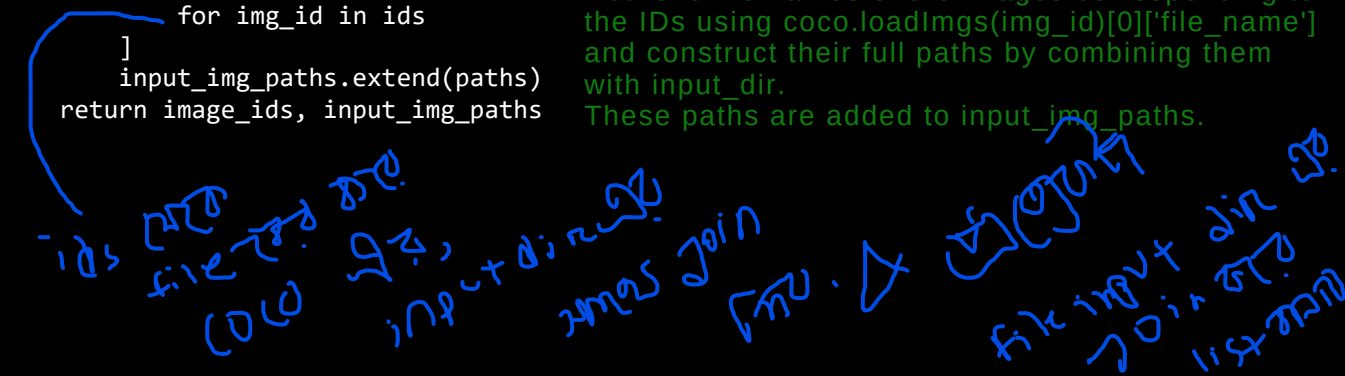
list1 = [1, 2, 3]
tuple1 = (4, 5)

list1.extend(tuple1)
print(list1)
# Output: [1, 2, 3, 4, 5]

os.path.join(input_dir, fname)
for fname in os.listdir(input_dir)
if fname.endswith(".jpg")

Extract all image IDs using coco.getImgIds() and add them to image_ids using .extend().
This ensures the list contains all image IDs from all COCO objects.

Retrieve file names of the images corresponding to the IDs using coco.loadImgs(img_id)[0]['file_name'] and construct their full paths by combining them with input_dir.
These paths are added to input_img_paths.



```
# Preprocessing Functions
def preprocess_image(img_path):
    """Loads and resizes an image."""
    input_img = tf_io.read_file(img_path)
    input_img = tf_io.decode_jpeg(input_img, channels=3)
    input_img = tf_image.resize(input_img, img_size)
    input_img = tf_image.convert_image_dtype(input_img, "float32")
    return input_img

def generate_segmentation_mask(coco, image_id, img_height, img_width):
    """Generates a binary mask from COCO segmentation annotations."""
    ann_ids = coco.getAnnIds(imgIds=image_id, iscrowd=False)
    annotations = coco.loadAnns(ann_ids)
    mask = np.zeros((img_height, img_width), dtype=np.uint8)
    for ann in annotations:
        if 'segmentation' in ann:
            rle = coco.annToRLE(ann)
            mask += decode(rle)  # Decode RLE to a binary mask
    return tf_image.resize(mask[..., None], img_size, method="nearest")

def preprocess_image_and_mask(img_path, image_id):
    """Preprocesses the image and generates a corresponding segmentation
mask."""
    # Load image
    input_img = preprocess_image(img_path)
    # Identify which COCO object contains the image ID
    for coco in coco_objects:
        if image_id in coco.getImgIds():
            img_info = coco.loadImgs(image_id)[0]
            img_height, img_width = img_info['height'],
img_info['width']
            mask = generate_segmentation_mask(coco, image_id,
img_height, img_width)
            mask = tf_image.convert_image_dtype(mask, "uint8")
            return input_img, mask
    raise ValueError(f"Image ID {image_id} not found in any annotation
files.")

# Dataset Function
def get_dataset(batch_size, img_paths, image_ids, preprocess_fn,
max_dataset_len=None):
    """Creates a dataset for images and COCO segmentation masks."""
    if max_dataset_len:
        img_paths = img_paths[:max_dataset_len]
        image_ids = image_ids[:max_dataset_len]
    dataset = tf_data.Dataset.from_tensor_slices((img_paths, image_ids))
    dataset = dataset.map(preprocess_fn,
num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)

# Main Execution
# Load all annotation files
coco_objects = load_coco_annotations(annotation_dir)

# Extract image IDs and paths
```

```
image_ids, input_img_paths = get_image_paths_and_ids(coco_objects,
input_dir)

# Validate that all referenced images exist
for img_path in input_img_paths:
    if not os.path.exists(img_path):
        raise FileNotFoundError(f"Image file not found: {img_path}")

# Random shuffling
val_samples = 1000
np.random.seed(1337)
shuffled_indices = np.random.permutation(len(image_ids))

# Split into train/val
train_indices = shuffled_indices[:-val_samples]
val_indices = shuffled_indices[-val_samples:]
train_img_paths = [input_img_paths[i] for i in train_indices]
train_image_ids = [image_ids[i] for i in train_indices]
val_img_paths = [input_img_paths[i] for i in val_indices]
val_image_ids = [image_ids[i] for i in val_indices]

# Training dataset
train_dataset = get_dataset(
    batch_size,
    train_img_paths,
    train_image_ids,
    preprocess_image_and_mask,
    max_dataset_len=1000
)

# Validation dataset
val_dataset = get_dataset(
    batch_size,
    val_img_paths,
    val_image_ids,
    preprocess_image_and_mask
)
```

load_img_masks(input_img_path, target_img_path)
This helper function processes a pair of input and target image file paths by performing the following:
Load the input image:
tf_io.read_file(input_img_path):
Reads the image file from the given path as a raw binary string.
tf_io.decode_png(input_img, channels=3):
Decodes the PNG file into a 3-channel (RGB) image tensor.
Resize the input image:
tf_image.resize(input_img, img_size):
Resizes the image to the dimensions specified in img_size.
Normalize the input image:
tf_image.convert_image_dtype(input_img, "float32"):
Converts the image's pixel values to the range [0, 1] by casting them to float32.

```
 import os
import random
import xml.etree.ElementTree as ET
import numpy as np
from tensorflow import data as tf_data
from tensorflow import image as tf_image
from tensorflow import io as tf_io

# Directory paths
input_dir = "images/"
annotation_dir = "annotations/xmls/"
img_size = (160, 160)
batch_size = 32

# Load input image paths
input_img_paths = sorted(
    [
        os.path.join(input_dir, fname)
        for fname in os.listdir(input_dir)
        if fname.endswith(".jpg")
    ]
)

# Load annotation paths
annotation_paths = sorted(
    [
        os.path.join(annotation_dir, fname)
        for fname in os.listdir(annotation_dir)
        if fname.endswith(".xml")
    ]
```

```
)

print("Number of samples:", len(input_img_paths))
for input_path, annotation_path in zip(input_img_paths[:10], annotation_paths[:10]):
    print(input_path, "|", annotation_path)

# Parse XML Annotations
def parse_xml(annotation_path):
    tree = ET.parse(annotation_path)
    root = tree.getroot()
    bboxes = []
    for obj in root.findall("object"):
        bbox = obj.find("bndbox")
        class_name = obj.find("name").text
        xmin = int(bbox.find("xmin").text)
        ymin = int(bbox.find("ymin").text)
        xmax = int(bbox.find("xmax").text)
        ymax = int(bbox.find("ymax").text)
        bboxes.append({
            "class": class_name,
            "bbox": [xmin, ymin, xmax, ymax]
        })
    return bboxes

# Example: Normalize bounding boxes and extract class IDs
def preprocess_bboxes(annotation_path, img_width, img_height):
    parsed = parse_xml(annotation_path)
    bboxes = []
    class_ids = []
    for item in parsed:
        # Normalize bounding box coordinates
        xmin, ymin, xmax, ymax = item["bbox"]
        bboxes.append([
            xmin / img_width, ymin / img_height,
            xmax / img_width, ymax / img_height
        ])
        # Example: Map class name to integer ID
        class_id = {"class1": 0, "class2": 1, "class3": 2}.get(item["class"], -1)
        class_ids.append(class_id)
    return np.array(bboxes, dtype=np.float32), np.array(class_ids, dtype=np.int32)

# Preprocessing Function for Dataset
def preprocess_images_and_annotations(input_img_path, annotation_path):
    # Preprocess image
    input_img = tf_io.read_file(input_img_path)
    input_img = tf_io.decode_png(input_img, channels=3)
    input_img = tf_image.resize(input_img, img_size)
```

```
    input_img = tf_image.convert_image_dtype(input_img, "float32")

    # Get original dimensions
    img_shape = tf_io.read_file(input_img_path)
    img_shape = tf_image.decode_png(img_shape, channels=3).shape[:2]
    img_height, img_width = img_shape

    # Preprocess bounding boxes and class IDs
    bboxes, class_ids = preprocess_bboxes(annotation_path, img_width, img_height)
    return input_img, (bboxes, class_ids)

# General Dataset Function
def get_dataset(
    batch_size,
    input_img_paths,
    annotation_paths,
    preprocess_fn,
    max_dataset_len=None,
):
    """Creates a TF Dataset for images and annotations."""
    if max_dataset_len:
        input_img_paths = input_img_paths[:max_dataset_len]
        annotation_paths = annotation_paths[:max_dataset_len]

    dataset = tf_data.Dataset.from_tensor_slices((input_img_paths, annotation_paths))
    dataset = dataset.map(preprocess_fn, num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)

# Random shuffling
val_samples = 1000
random.Random(1337).shuffle(input_img_paths)
random.Random(1337).shuffle(annotation_paths)

# Splitting into training and validation sets
train_input_img_paths = input_img_paths[:-val_samples]
train_annotation_paths = annotation_paths[:-val_samples]
val_input_img_paths = input_img_paths[-val_samples:]
val_annotation_paths = annotation_paths[-val_samples:]

# Create training dataset
train_dataset = get_dataset(
    batch_size,
    train_input_img_paths,
    train_annotation_paths,
    preprocess_images_and_annotations,
    max_dataset_len=1000,
)
```

```
# Create validation dataset
val_dataset = get_dataset(
    batch_size,
    val_input_img_paths,
    val_annotation_paths,
    preprocess_images_and_annotations,
    max_dataset_len=1000,
)
```

Step 1: Organize the Content

First, organize the content into sections with clear headings and subheadings. This will make it easier to read and understand.

Step 2: Use Markdown Formatting

Markdown is a lightweight markup language that you can use to format your text. Here's how you can structure the document:

markdown
Copy

```
# Going Modular: Turning Notebook Code into Reusable Python Files

## Introduction
The main concept of this section is to turn useful notebook code cells into reusable Python files. This will save us from writing the same code over and over again.

## Notebooks Overview
There are two notebooks for this section:

1. **05. Going Modular: Part 1 (cell mode)** - This notebook is run as a traditional Jupyter Notebook/Google Colab notebook and is a conden sed version of notebook 04.
2. **05. Going Modular: Part 2 (script mode)** - This notebook is the same as number 1 but with added functionality to turn each of the maj or sections into Python scripts, such as `data_setup.py` and `train.py`.

## Why Two Parts?
Because sometimes the best way to learn something is to see how it differs from something else. If you run each notebook side-by-side, you'l l see how they differ, and that's where the key learnings are.

## What We're Working Towards
By the end of this section, we want to have two things:

1. The ability to train the model we built in notebook 04 (Food Vision Mini) with one line of code on the command line: `python train.py`.
2. A directory structure of reusable Python scripts, such as:

going_modular/
├── going_modular/
│   ├── data_setup.py
│   ├── engine.py
│   ├── model_builder.py
│   ├── train.py
│   └── utils.py
├── models/
│   ├── 05_going_modular_cell_mode_tinyvgg_model.pth
│   └── 05_going_modular_script_mode_tinyvgg_model.pth
└── data/
    └── pizza_steak_sushi/
        ├── train/
        │   ├── pizza/
        │   │   ├── image01.jpeg
        │   │   └── ...
        │   ├── steak/
        │   └── sushi/
```

```
└─── test/
├─── pizza/
├─── steak/
└─── sushi/
```

Copy

```
## Things to Note
- **Docstrings**: Writing reproducible and understandable code is important. Each of the functions/classes we'll be putting into scripts has been created with Google's Python docstring style in mind.
- **Imports at the top of scripts**: All of the Python scripts we're going to create could be considered small programs on their own, so all of the scripts require their input modules to be imported at the start of the script.

## 0. Cell Mode vs. Script Mode
- **Cell Mode Notebook**: Such as 05. Going Modular Part 1 (cell mode), is a notebook run normally, each cell in the notebook is either code or markdown.
- **Script Mode Notebook**: Such as 05. Going Modular Part 2 (script mode), is very similar to a cell mode notebook, however, many of the code cells may be turned into Python scripts.

## 1. Get Data
Getting the data in each of the 05 notebooks happens the same as in notebook 04. A call is made to GitHub via Python's `requests` module to download a `.zip` file and unzip it.

```python
import os
import requests
import zipfile
from pathlib import Path

# 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")
```

Path orara

Opens a file in binary write mode ("wb") to save the downloaded content.
f.write(request.content)
Writes the binary content (request.content) of the downloaded file into the opened file.

"r": Opens the file in read mode.

2. Create Datasets and DataLoaders (data_setup.py)

Once we've got data, we can then turn it into PyTorch `Dataset`'s and `DataLoader`'s (one for training data and one for testing data).

We convert the useful `Dataset` and `DataLoader` creation code into a function called `create_dataloaders()`.
```

python
Copy

```
%%writefile going_modular/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, # don't need to shuffle test data
      num_workers=num_workers,
      pin_memory=True,
```

```
)

return train_dataloader, test_dataloader, class_names
```

3. Making a Model (model_builder.py)

Over the past few notebooks (notebook 03 and notebook 04), we've built the TinyVGG model a few times. So it makes sense to put the model into its file so we can reuse it again and again.

python
Copy

```
%%writefile going_modular/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.conv_block_2(self.conv_block_1(x))) # <- leverage the benefits of operator fusion
```

4. Creating train_step() and test_step() Functions and train() to Combine Them

We wrote several training functions in notebook 04:

- train_step() - takes in a model, a DataLoader, a loss function, and an optimizer and trains the model on the DataLoader.
- test_step() - takes in a model, a DataLoader, and a loss function and evaluates the model on the DataLoader.
- train() - performs 1. and 2. together for a given number of epochs and returns a results dictionary.

Since these will be the engine of our model training, we can put them all into a Python script called engine.py.

python
Copy

```
%%writefile going_modular/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
```

5. Creating a Function to Save the Model (utils.py)

Often you'll want to save a model whilst it's training or after training. Since we've written the code to save a model a few times now in previous notebooks, it makes sense to turn it into a function and save it to file.

python
Copy

```
%%writefile going_modular/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)
```

6. Train, Evaluate and Save the Model (train.py)

As previously discussed, you'll often come across PyTorch repositories that combine all of their functionality together in a train.py file. This file is essentially saying "train the model using whatever data is available".

In our train.py file, we'll combine all of the functionality of the other Python scripts we've created and use it to train a model. This way we can train a PyTorch model using a single line of code on the command line:

bash
Copy

```
python train.py
```

To create train.py, we'll go through the following steps:

1. Import the various dependencies, namely torch, os, torchvision.transforms, and all of the scripts from the going_modular directory, data_setup, engine, model_builder, utils.
2. Setup various hyperparameters such as batch size, number of epochs, learning rate, and number of hidden units (these could be set in the future via Python's argparse).
3. Setup the training and test directories.
4. Setup device-agnostic code.
5. Create the necessary data transforms.
6. Create the DataLoaders using data_setup.py.
7. Create the model using model_builder.py.
8. Setup the loss function and optimizer.
9. Train the model using engine.py.
10. Save the model using utils.py.

python
Copy

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

import os
import torch
import data_setup, engine, model_builder, utils

from torchvision import transforms

# Setup hyperparameters
NUM_EPOCHS = 5
BATCH_SIZE = 32
HIDDEN_UNITS = 10
LEARNING_RATE = 0.001

# Setup directories
train_dir = "data/pizza_steak_sushi/train"
test_dir = "data/pizza_steak_sushi/test"

# 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")
```

Now we can train a PyTorch model by running the following line on the command line:

```
bash
Copy
python train.py
```

Doing this will leverage all of the other code scripts we've created. And if we wanted to, we could adjust our `train.py` file to use argument flag inputs with Python's `argparse` module, this would allow us to provide different hyperparameter settings like previously discussed:

```
bash
Copy
python train.py --model MODEL_NAME --batch_size BATCH_SIZE --lr LEARNING_RATE --num_epochs NUM_EPOCHS
```

Conclusion

By modularizing our code into reusable Python scripts, we can significantly improve the efficiency and maintainability of our machine learning workflows. This approach not only saves time but also makes it easier to share and collaborate on projects.

Where to Get Help

- All of the materials for this course are available on GitHub.
- If you run into trouble, you can ask a question on the course GitHub Discussions page.
- And of course, there's the PyTorch documentation and PyTorch developer forums, a very helpful place for all things PyTorch.

Copy

```
### Step 3: Convert Markdown to PDF
You can use a tool like `pandoc` to convert the Markdown file to a PDF. Here's how you can do it:

1. **Install Pandoc**: If you don't have Pandoc installed, you can download it from [here](https://pandoc.org/installing.html).

2. **Convert Markdown to PDF**: Use the following command to convert your Markdown file to PDF:

   ```bash
   pandoc -o output.pdf input.md
```

Replace `input.md` with the name of your Markdown file and `output.pdf` with the desired output PDF file name.

Step 4: Review the PDF

Open the generated PDF to ensure that the formatting is correct and that all sections are properly organized.

Step 5: Share the PDF

Once you're satisfied with the PDF, you can share it with others or use it as a reference document for your project.

This process will give you a well-documented PDF that you can use to share your work or keep as a reference for future projects.
```